Designing, Debugging, and Deploying Configurable Computing Machine-based Applications Using Reconfigurable Computing Application Frameworks

Anthony Lynn Slade
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd
Part of the Electrical and Computer Engineering Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/58

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
DESIGNING, DEBUGGING, AND DEPLOYING CONFIGURABLE
COMPUTING MACHINE-BASED APPLICATIONS USING
RECONFIGURABLE COMPUTING APPLICATION FRAMEWORKS

by

Anthony Lynn Slade

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Brigham Young University

April 2003
BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Anthony Lynn Slade

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date

Brent E. Nelson, Chair

Date

Dan R. Olsen, Jr.

Date

Michael J. Wirthlin
As chair of the candidate’s graduate committee, I have read the thesis of Anthony Lynn Slade in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Brent E. Nelson
Chair, Graduate Committee

Accepted for the Department

A. Lee Swindlehurst
Graduate Coordinator

Accepted for the College

Douglas M. Chabries
Dean, College of Engineering and Technology
ABSTRACT

DESIGNING, DEBUGGING, AND DEPLOYING CONFIGURABLE COMPUTING MACHINE-BASED APPLICATIONS USING RECONFIGURABLE COMPUTING APPLICATION FRAMEWORKS

Anthony Lynn Slade

Department of Electrical and Computer Engineering

Master of Science

Configurable computing machines (CCMs) offer high-performance application acceleration with custom hardware. They are also dynamically reconfigurable and give significant internal visibility. Such features are useful throughout the design, debug, and deploy stages of CCM-based application development. However traditional, monolithic design tools do not offer adequate support for all of these development stages. This thesis describes a specification for a reconfigurable computing application framework (RCAF) which is more suitable for CCM application development. It also describes an implementation of such an RCAF. This RCAF improves the efficiency of application design and debugging. It also establishes an application architecture framework which helps to build up not only the hardware design, but also the application software and user interface. Applications built using this small, deployable RCAF may also perform significantly better due to the dynamic hardware reconfiguration features included with the RCAF.
ACKNOWLEDGMENTS

There are many people without whom I could not have completed this work. There is so much I would like to do for all of them to show my gratitude. This list of acknowledgments is but a beginning of that process.

First and foremost, I express my love and appreciation for my family members, Noni Rae, RaeLynn, and Joseph. Their love and devotion, and especially patience, are invaluable in all of my endeavors. Thank you, Rae, I love you. I also express my thanks to my parents, James and Karin Slade.

Much thanks goes to those who have helped me directly on this thesis. They include my thesis advisory committee members, Brent E. Nelson, Dan R. Olsen, Jr., and Michael J. Wirthlin. Others who have given great amounts of direct support and advice include M. Ryan Byrd, Adrian Evans, K. Scott Hemmert, Clint Hilton, Preston Jackson, Brian McMurtrey, Eric Roesler, K. Ann Tanner, Justin Tripp, and Robyn Voelckers.

Of course I cannot forget the extensive support of the many members of Brigham Young University’s Configurable Computing Laboratory with whom I have worked. They include Brad Hutchings, Brent Nelson, Michael Wirthlin, Greg Ahlquist, Dan Baker, Peter Bellows, Eric Blake, Ben Bullough, Ryan Byrd, Dan Carver, Rob Franklin, Russell Fredrickson, Paul Graham, Mike Halcrow, Scott Hemmert, Clint Hilton, Preston Jackson, Eric Johnson, Nathan Kitchen, Matthew Koecher, Wes Landaker, Anshul Malvi, Jared Martin, Brian McMurtrey, Steve Morrison, Devin Pratt, Eric Quist, Curtis Randall, Nathan Rollins, Eric Roesler, Michael Rytting, Aaron Stewart, Navanee Sundaramoorthy, Justin Tripp, Trent VanDenBerghe, Isaac Wagner, Tim Wheeler, Brett Williams, and Matt Young.
Contents

Acknowledgments vi

List of Tables xi

List of Figures xiv

1 Introduction & Motivation 1

1.1 Application Acceleration Using FPGA-based CCMs 1

1.2 CCM Application Production Stages 2

1.2.1 The Design Stage 3

1.2.2 The Debug Stage 3

1.2.3 The Deploy Stage 3

1.3 Limitations of Current CAD Tools for CCM Application Development 4

1.3.1 ASIC-oriented History of CAD Tools 4

1.3.2 Limitations Imposed by the ASIC-oriented Nature of CAD Tools 5

1.4 Features of CCM Systems Improve the Production of Applications 7

1.4.1 CCM Features Improve Application Debugging 7

1.4.2 CCM Features Permit More Design Exploration 9

1.4.3 CCM Features Improve Application Performance 9

1.5 The Significance of the Deploy Stage 11

1.6 Reconfigurable Computing Application Frameworks 14

2 Reconfigurable Computing Application Frameworks 17

2.1 RCAF Interface Between CCM Hardware and CCM Software 19

2.2 RCAF API For Application Interfaces 20
2.3 RCAF Support for (Dynamic) Circuit Design .................................. 21
2.4 Designing, Debugging, and Deploying CCM Applications ............... 21

3 JHDL: A Basis for a Reconfigurable Computing Application Framework 23
3.1 JHDL Design Fundamentals ....................................................... 23
3.1.1 The JHDL Design Data Model .............................................. 23
3.1.2 JHDL Application Programming Interface .............................. 25
3.2 Hardware Support in JHDL ....................................................... 33
3.3 JAB: The JHDL Circuit Browser ............................................... 35
3.3.1 Interactive Circuit Visualization ........................................... 35
3.3.2 Simulation and Execution Control ........................................ 36
3.3.3 JAB Customizability ......................................................... 39
3.4 JHDL and JAB: A Precursor to an RCAF System ......................... 41
3.4.1 Drawbacks to the JHDL and JAB System ............................... 41
3.4.2 Lessons From JAB ............................................................... 42

4 Circuit Visualization Tool (CVT) ................................................. 43
4.1 CVT Architecture ................................................................. 43
4.1.1 The CVT Model ............................................................... 44
4.1.2 The CVT Controller ......................................................... 44
4.1.3 The CVT View ................................................................. 46
4.1.4 Integrating CVT Model, View, and Controller ....................... 46
4.2 Generic Circuit Verification Tool .............................................. 48
4.3 Custom Circuit Interaction Components .................................... 49
4.4 Creating Full Custom CCM Applications by Extending CVT ............ 51
4.4.1 Custom CCM Application Model Component ....................... 51
4.4.2 Custom CCM Application View Components ....................... 51
4.4.3 Custom CCM Application Controller Component .................. 53
4.5 Application Deployment .......................................................... 53
4.6 Tradeoffs and Considerations in Application Deployment ............... 54
4.6.1 Framework and Application Code Execution .......................... 55
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6.2 CCM Board Support</td>
<td>55</td>
</tr>
<tr>
<td>4.6.3 Execution-time Hardware Design Modifications</td>
<td>56</td>
</tr>
<tr>
<td>4.6.4 Deployment Considerations Conclusion</td>
<td>58</td>
</tr>
<tr>
<td>5 Example Applications</td>
<td>59</td>
</tr>
<tr>
<td>5.1 Audio Processing Application</td>
<td>59</td>
</tr>
<tr>
<td>5.2 Edit Distance Application</td>
<td>61</td>
</tr>
<tr>
<td>5.2.1 Edit Distance Application Model</td>
<td>62</td>
</tr>
<tr>
<td>5.2.2 Edit Distance Application View</td>
<td>64</td>
</tr>
<tr>
<td>5.2.3 Edit Distance Application Controller</td>
<td>65</td>
</tr>
<tr>
<td>5.2.4 Completing the Edit Distance Application</td>
<td>65</td>
</tr>
<tr>
<td>5.3 Guessing Game Application</td>
<td>66</td>
</tr>
<tr>
<td>5.3.1 Guessing Game Application Model</td>
<td>67</td>
</tr>
<tr>
<td>5.3.2 Guessing Game Application View</td>
<td>68</td>
</tr>
<tr>
<td>5.3.3 Guessing Game Application Controller</td>
<td>69</td>
</tr>
<tr>
<td>5.4 Quantitative Analysis</td>
<td>70</td>
</tr>
<tr>
<td>5.5 Additional Examples</td>
<td>72</td>
</tr>
<tr>
<td>6 Conclusions and Future Work</td>
<td>77</td>
</tr>
<tr>
<td>6.1 Contributions of this Research</td>
<td>78</td>
</tr>
<tr>
<td>6.2 Future Work</td>
<td>80</td>
</tr>
<tr>
<td>A CVT Component APIs</td>
<td>85</td>
</tr>
<tr>
<td>A.1 CVT Controller APIs</td>
<td>85</td>
</tr>
<tr>
<td>A.2 CVT View APIs</td>
<td>89</td>
</tr>
<tr>
<td>B CVT Component Sizes</td>
<td>99</td>
</tr>
<tr>
<td>C Example Source Code</td>
<td>101</td>
</tr>
<tr>
<td>C.1 Custom Visualization Using JAB</td>
<td>101</td>
</tr>
<tr>
<td>C.2 Example Application: Audio Processing</td>
<td>105</td>
</tr>
<tr>
<td>C.2.1 Audio Processing Application Model</td>
<td>105</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>HWSystem methods for accessing circuit design cells</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>JHDL Cell methods for accessing circuit design structure</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>Wire class methods for accessing circuit design structure</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>JHDL TechMapper methods for writing a netlist file</td>
<td>29</td>
</tr>
<tr>
<td>3.5</td>
<td>HWSYSTEM methods for simulation control and feedback</td>
<td>30</td>
</tr>
<tr>
<td>3.6</td>
<td>The SimulatorCallback interface for simulator feedback</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>Wire class methods for stimulating wire objects</td>
<td>31</td>
</tr>
<tr>
<td>3.8</td>
<td>JHDL Wire methods for accessing circuit state</td>
<td>32</td>
</tr>
<tr>
<td>3.9</td>
<td>JHDL MemoryInterface methods</td>
<td>33</td>
</tr>
<tr>
<td>3.10</td>
<td>The UserDefinedSchematic interface for custom schematic nodes</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Broker methods for extending the CVT system</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>ColorPlacementListener method for obtaining guess events from</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>the guessing game viewer</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Sizes of sample CVT-based applications</td>
<td>72</td>
</tr>
<tr>
<td>B.1</td>
<td>Sizes of the CVT RCAF and its subcomponents</td>
<td>99</td>
</tr>
<tr>
<td>B.2</td>
<td>Sizes of various technology-specific libraries</td>
<td>100</td>
</tr>
<tr>
<td>B.3</td>
<td>Sizes of additional CVT components</td>
<td>100</td>
</tr>
<tr>
<td>B.4</td>
<td>Total size of all components listed in previous tables</td>
<td>100</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Configurable Computing Machine (CCM) Block Diagram.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Stages of CCM application production</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>CCM application production stages are typically split between hardware and software development</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Features of FPGA-based CCMs</td>
<td>8</td>
</tr>
<tr>
<td>1.5</td>
<td>FPGA LUT-based general-purpose comparator.</td>
<td>11</td>
</tr>
<tr>
<td>1.6</td>
<td>FPGA LUT-based constant comparator.</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>CCM benefits leveraged by RCAF.s to improve application development</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>The RCAF interfaces between the CCM hardware and application software</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>The RCAF provides a mapping between the hardware design and the physical CCM hardware</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Inheritance structure of JHDL’s base classes.</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Hierarchy of the JHDL system.</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>JHDL’s hardware systems exposes three types of APIs to external systems.</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>A design model of an FPGA-based circuit.</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>Representation of the physical implementation of an FPGA-based circuit.</td>
<td>34</td>
</tr>
<tr>
<td>3.6</td>
<td>JAB’s circuit hierarchy browser.</td>
<td>36</td>
</tr>
<tr>
<td>3.7</td>
<td>JAB’s design schematic viewer.</td>
<td>37</td>
</tr>
<tr>
<td>3.8</td>
<td>JAB’s memory viewer.</td>
<td>37</td>
</tr>
<tr>
<td>3.9</td>
<td>JAB’s waveform viewer.</td>
<td>37</td>
</tr>
<tr>
<td>3.10</td>
<td>JAB’s command line console.</td>
<td>38</td>
</tr>
<tr>
<td>3.11</td>
<td>Customizing JAB schematics with user-defined schematic nodes.</td>
<td>39</td>
</tr>
<tr>
<td>3.12</td>
<td>An example custom circuit visualization component using the SimulatorCallback interface.</td>
<td>40</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.1</td>
<td>The model-view-controller architecture for user interfaces.</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>CVT Architecture.</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Example communication in the CVT model-view-controller architecture.</td>
<td>47</td>
</tr>
<tr>
<td>4.4</td>
<td>CVT’s main circuit view.</td>
<td>48</td>
</tr>
<tr>
<td>4.5</td>
<td>CVT provides several default viewers with intuitive interfaces.</td>
<td>50</td>
</tr>
<tr>
<td>4.6</td>
<td>Communications to and from CVT viewers.</td>
<td>52</td>
</tr>
<tr>
<td>5.1</td>
<td>Custom viewer for an audio processing application.</td>
<td>60</td>
</tr>
<tr>
<td>5.2</td>
<td>The \texttt{EditDistance} class creates the circuit model for the edit distance application.</td>
<td>63</td>
</tr>
<tr>
<td>5.3</td>
<td>The graphical user interface for an edit distance application.</td>
<td>64</td>
</tr>
<tr>
<td>5.4</td>
<td>The \texttt{EDBroker} class extends the default RCAF controller for the edit distance application.</td>
<td>66</td>
</tr>
<tr>
<td>5.5</td>
<td>The \texttt{tbEditDistance} class creates and integrates the various application components.</td>
<td>67</td>
</tr>
<tr>
<td>5.6</td>
<td>The graphical user interface for a guessing game application.</td>
<td>68</td>
</tr>
<tr>
<td>5.7</td>
<td>The \texttt{EDBroker} class extends the default RCAF controller for the edit distance application.</td>
<td>70</td>
</tr>
<tr>
<td>5.8</td>
<td>Hybrid CPU/FPGA system debugger.</td>
<td>74</td>
</tr>
<tr>
<td>5.9</td>
<td>Debugger for hardware synthesized from high-level code.</td>
<td>75</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction & Motivation

1.1 Application Acceleration Using FPGA-based CCMs

Field-programmable gate arrays (FPGAs) are reconfigurable, general-purpose hardware devices. It is somewhat common to use FPGAs as replacements for very large and complex glue logic in digital systems. FPGAs are also useful for creating hardware prototypes of ASIC (application-specific integrated circuit) designs. FPGAs can help decrease development time by providing a hardware platform in which ASIC designs may be verified. Where a very quick time to market is required for the success of new digital hardware systems, FPGAs can even be easy and economical replacements for ASICs altogether[1].

Most recently, FPGAs have been seen as viable core technologies for serious computing applications. Configurable computing machines (CCMs) are FPGA-based processing and computing platforms. Figure 1.1 shows the components of a typical CCM architecture. CCMs provide a versatile and powerful computing platform that may be integrated with a host system. The FPGAs in a CCM can be configured by the host system to perform virtually any type of computation or process. This makes CCMs attractive computing platforms with flexibility comparable to that of microprocessor-based systems and performance comparable to that of ASIC designs[2].

Because CCM computation and processing occurs in hardware, applications targeted to CCMs may be able to run several times faster than if they were executed in software running on a processor. In some cases, CCM-based applications may be as much as an order of magnitude faster than alternative implementations[2]. CCMs are also capable of improving the performance-to-price ratio in many applications. For example, Sun
Microsystems and TimeLogic Corporation recently teamed up to create relatively inexpensive CCM systems that can outperform large and expensive server farms performing bio-informatics algorithms[3].

1.2 CCM Application Production Stages

Production of CCM-based application occurs in three stages. These stages are design, debug, and deploy. These stages are depicted in Figure 1.2 and described below.
1.2.1 The Design Stage

The design stage of CCM application development includes the processes required to specify the application features. It also includes the process of actually implementing the application. A major part of the design involves creating a structural or behavioral description of the application hardware configuration. The application also includes software that supports the hardware and acts as an interface to the user or to other external systems. In some cases, this software component may be just as significant as the hardware configuration design.

1.2.2 The Debug Stage

The debug stage takes the design and exercises it to verify that it functions correctly. Traditional FPGA debugging uses a simulation model created by a hardware synthesizer. This model approximates or emulate the execution behavior of the FPGA hardware as if it were configured with the hardware design. The debugging process may expose errors or weaknesses in the design. Developers may use this information to revisit portions of the design stage to modify the application design in order to remove the problems or minimize their effects. This is often an iterative process between the design and debug stages.

1.2.3 The Deploy Stage

The CCM application is prepared for use by the end user in the deploy stage. This process may involve finalizing the integration of the hardware and software components of the application. The result of the deploy stage is an application that is in a form which may be distributed to end users.

An important part of the deploy stage is preparing a system that will enable the application to execute on the final target system. This includes establishing an application infrastructure and runtime environment. This may involve specifying drivers and other tools that the user must have in order to execute the application properly. In some cases, these tools may actually accompany or be integrated with the application itself.
1.3 Limitations of Current CAD Tools for CCM Application Development

State-of-the-art computer-aided design (CAD) tools for developing CCM-based applications are inadequate to support all of the stages of development described in Section 1.2. These CAD tools tend to be very good for the design stage. However, their support for the debug stage of CCM applications is usually only marginal. And in general, these CAD tools do not support the deployment stage of CCM application development at all. Even the design support provided by these tools is only useful for the hardware portion of the whole CCM application; it ignores the application software development needs altogether.

1.3.1 ASIC-oriented History of CAD Tools

Hardware design that targets FPGAs has long been constrained by a design flow that was originally developed for the production of ASICs. This ASIC-oriented tendency is the natural product of the history of the development of the FPGA market. In order to quickly generate a new market for their products, FPGA manufacturers have relied heavily on an existing electronic design automation (EDA) and CAD tool infrastructure. Rather than build a new set of tools to support design creation, synthesis, and simulation of FPGA designs, these manufacturers have leveraged the capabilities of existing software to support their products. FPGA manufacturers have done this in two ways: providing libraries for use with major CAD tools, or by providing sufficient information and specifications to enable others to more easily provide such libraries. These libraries contain information about the various primitive cells supported by the back-end synthesis tools that are used to generate configuration images for specific FPGA devices. Designers could then use their favorite EDA/CAD tools to create designs that target these libraries in order to synthesize them to the FPGA device of choice.

The strategy of supporting existing EDA/CAD tools has assisted FPGA manufacturers in successfully building up a large market for their products. However, this success has come at a fairly significant cost for the development of configurable computing machine-based applications. At issue is the nature of the existing EDA/CAD tools used to target FPGAs. For the most part, these tools (e.g. Mentor, Cadence, PowerView, etc.) are geared
mainly toward ASIC design\cite{4}. The ASIC-oriented nature of CAD tools has been adequate for creating FPGA designs that serve as custom glue logic or ASIC replacements. However, these tools typically cannot support the full process of designing, debugging, and deploying both the hardware and software of a CCM application. The limitations found in these tools are described in the following section.

1.3.2 Limitations Imposed by the ASIC-oriented Nature of CAD Tools

The ASIC-oriented tendency of EDA/CAD tools assumes that the design may only be put into hardware form after extensively simulating it in order to fully verify its functionality. Verification via simulation is typically a time- and resource-intensive part of the design process. Relying exclusively on simulation to verify hardware designs retards the debug stage of the development of CCM applications. While simulation does have its place in designing for FPGA hardware, reducing reliance on simulation is possible if the target FPGA hardware is used during the design process. The application developer may execute the design in the FPGA hardware and analyze its operation. In-hardware debugging of FPGA designs is something not possible in the ASIC design process, since the very purpose of that process is to produce the final hardware itself. A notable exception is the use of programmable hardware devices, such as FPGAs, to approximate the execution behavior of ASIC designs. In-hardware debugging of FPGA designs allows the designer to bypass great amounts of the simulation that would otherwise be crucial to an ASIC design process.

A CCM is coupled with an external host through software running on that host. This software forms a part of the CCM application itself. It not only provides a data and control path between the host and the CCM, it also provides an interactive interface to the end user of the application. This software is crucial to the final deployment of the application. However, CAD tools usually only focus on the hardware design of such applications. The result is a split development process as shown in Figure 1.3. This separation of hardware and software components hinders the development of CCM applications in two ways. First, the disjoint, often sequential, development of the hardware design and the software design extends the time necessary to develop the whole CCM application. Concurrent, integrated development of these application components would streamline the production of
the application and reduce the complexity of the deployment stage. Second, separation of the development of these two components may prevent the developers from attaining added insight into the application that could only come from an integrated design process. Such an integrated process would offer the designers added flexibility to shift functionality between the software and the hardware portions of the application. Therefore, the integrated process would help with the partitioning problem associated with hardware-software co-design. Debugging the whole application, rather than separated software and hardware components, would also help expose and eliminate design flaws that may only be detected in an integrated system.

![Diagram](image.png)

Figure 1.3: CCM application production stages are typically split between hardware and software development.

Because typical CAD tools are primarily ASIC-oriented, they usually do not include support for runtime support of CCM hardware. This means that they do not have the capability of linking into drivers and interface APIs that are used to establish execution-time support for CCM applications for a given CCM platform. Many configurable computing machine platforms come with board support packages that expose APIs that may be used to configure and execute hardware designs within the hardware resources provided by the CCM. Because common CAD tools do not have the capability of integrating with these board support packages, CCM application developers may not be able to use the CAD tools to help them build up the runtime environment for the final application to be deployed.
In summary, traditional EDA/CAD tools typically have three main weaknesses related to CCM application development. First, the use of these tools hinders the progress of application development by relying exclusively on slow simulation for debugging. Second, they unnecessarily separate the development of the hardware and software components of the CCM application, thereby extending and complicating the development process. Finally, modern CAD tools lack flexible hardware and software development environments to support the complete deployment of CCM applications.

1.4 Features of CCM Systems Improve the Production of Applications

Development of CCM applications may be improved by leveraging the features of FPGAs and FPGA-based CCM platforms. As noted by Hutchings and Nelson[5], three major features of CCMs are:

- Availability of the hardware at design time
- Programmability and re-programmability of the hardware
- Visibility into the hardware

To this list, Graham[6] adds the following:

- Controllability of the hardware state
- System execution controllability

These features are listed again in Figure 1.4, except that the two controllability items are combined into one generic controllability feature. Two major benefits come as a result of these features of CCM systems: a reduction in the need for simulation for hardware design verification, and tighter integration of the software and the hardware components of CCM applications.

1.4.1 CCM Features Improve Application Debugging

CCM features listed in Figure 1.4 allow for significant improvements to the debug stage of application development. The ASIC design approach must rely solely on simulation to verify the design. Software-based simulation of hardware designs can easily become
a bottleneck in the hardware design process. But the same hardware that is meant to accelerate the operation of the final application can also be used to accelerate the verification and debugging of the application design itself. In-hardware execution and analysis of the design, made possible by the availability of the hardware even before the hardware configuration design is complete, can reduce or eliminate the reliance on simulation. Because FPGA hardware is re-programmable, the designer may use a design cycle that involves design, compilation, and hardware execution and test. This design cycle, which parallels exactly the process of producing software applications, may be repeated as often as necessary until the design is completed. Using FPGA target hardware as a replacement for full simulation is possible due to the extraordinary internal visibility made available by the hardware.

A knowledgeable designer may manually make use of these features even without specific support for their utilization from within the principle development tools of the design. However, using these features in a disjoint combination may still not lead to a more effective design process. An EDA tool that integrates the use of available FPGA hardware during the design process will improve the efficiency of that process by allowing the designer to concentrate design efforts on just one tool or development environment. The result is an integrated development environment (IDE) that may closely resemble IDEs found in the software development community. Such systems have long been seen as useful development tools for software engineers. They are beneficial in improving the cyclic process of designing and debugging software applications. The programmer may use an

---

1 For example, Xilinx FPGAs offer a “readback” capability that gives the current state of all register and memory elements in the FPGA device.
IDE to create a software program and then execute it in an integrated debugger that reveals the program state at each line of executed code. The analogy extends to hardware design when a debugging environment is able to back annotate the design or a representation of the design (perhaps a schematic or other view of the circuits) to then give the designer a view of the hardware at each significant execution state.

1.4.2 CCM Features Permit More Design Exploration

It is difficult to overemphasize the significance of the potential benefits of utilizing available target hardware during the design process for FPGA-based designs. In-hardware execution is typically several orders of magnitude faster than simulation\[^6\]. Integrating the information from in-hardware execution of incremental and final builds of the hardware design within the development environment allows the designer to verify the design in less time\[^5\]. It may also give the designer superior insights into the operation of the design. The designer may, with improved efficiency, build up and instrument the design in various fashions to test alternate implementations. Therefore, leveraging the benefits of the presence of target hardware during design may help to reduce the amount of time required to develop the design as well as help produce a more optimized design than might be possible using only design methods used in ASIC design.

The integration of CCM hardware with the design environment may also benefit application developers during the deployment stage of application production. The programmability, visibility, and controllability features of CCMs allow designers to treat the CCM platforms as if they were extensions to the software part of the application. Concurrent and integrated development of the software and hardware portions of the application gives a more useful view of the CCM application as a whole computing system, rather than two separate pieces. This enables further design exploration that helps to determine the optimal partition between the hardware and software components of the application.

1.4.3 CCM Features Improve Application Performance

Perhaps one of the greatest benefits of an integrated design process is the potential for improved application performance. Performance enhancements in a CCM application
are often made by embedding specific data from the problem being solved by the application into the hardware design. This concept is similar to constant folding, which is often performed in the compilation of software systems. Constant folding improves the performance of software by embedding constant values in the assembled machine code of the software. This approach avoids the extra overhead that comes from treating constants like any other variables that must be stored into and loaded from memory. Besides reducing memory accesses, constant folding can speed up the execution of software by utilizing optimized ways of executing operations. For example, division or multiplication by a constant value that is a power of two may be replaced by much faster logical shifts. Smart compilers may favor such optimizations over more processor-intensive ways of executing the code. The added complexity of the compiler may be easily made up in performance of the software, especially when the optimization is made in an inner loop of code that is executed many times. Corresponding optimizations for hardware design have been called data folding[7] or constant propagation[8]. Some examples of hardware constant propagation include:

- replacing complex combinational logic with faster lookup tables

- instead of large, slow, general-purpose multipliers, using more efficient constant-coefficient multipliers

- creating a custom state flow for controllers made with finite state machines

- etc.

For a graphical example of constant propagation, see Figure [1.5 and Figure [1.6. Figure 1.5 shows a four-bit comparator implemented with two FPGA lookup tables (LUTs) and an AND gate. Figure 1.6 shows a four-bit comparator with one of the values, a known constant, embedded in the configuration of the LUT. The constant comparator is more efficient in both area and latency.

With high-performance CCM applications, when the data changes, so does the problem to be solved. Integrating the development of the application interface software with the development of the hardware design makes it possible to integrate user-provided
data with the hardware design at the time of execution. Unfortunately, traditional design tools and practices do not provide the components necessary to carry out this integration, which includes linking the application interface and user input data path with features that may enable dynamic reconfiguration of the hardware design.

![FPGA LUT-based general-purpose comparator.](image1)

**Figure 1.5:** FPGA LUT-based general-purpose comparator.

![FPGA LUT-based constant comparator.](image2)

**Figure 1.6:** FPGA LUT-based constant comparator.

### 1.5 The Significance of the Deploy Stage

The deploy stage of development (described in Section 1.2.3) is particularly important for CCM-based applications. This stage of development determines the ultimate
characteristics of the application as it is used by the end user. These characteristics may or may not fully take advantage of the features of FPGA-based CCMs, which are described in Section 1.4, particularly the features described in Section 1.4.3.

In traditional digital logic design, including FPGA and CCM design, the CAD/EDA tool does not play a role in the application development past the debug stage. However, the EDA tool is typically the only means of realizing significant modifications to the hardware configuration design. Even if a CAD/EDA tool did contain specialized features to support hardware-enhanced design and debugging, as introduced in Sections 1.4.1 and 1.4.2, those unique and powerful capabilities would normally be thrown away once the hardware design is complete and the deployment stage of development begins. Removing the CAD tool from the application after the debug stage inhibits the application’s ability to offer flexibility and performance advantages to the end user. For example, hardware optimizations based on input from the user, as introduced in Section 1.4.3, are not feasible under traditional application development processes as dictated by the nature of the CAD/EDA tools themselves.

Consider, for example, an application that compares two strings of characters to identify similarities or differences between them. Such an example will also be presented in greater detail in Section 5.2. This application uses comparators like the comparator shown in Figure 1.6. An electronic design automation tool is used to determine the configuration of the LUT in order to embed the information for a specific target string into the configuration of the hardware design. The typical configuration process requires the EDA tool to generate a circuit netlist. This netlist is then interpreted by placement and routing tools specific to the FPGAs in the target CCM. However, without the availability of the EDA tool, the configuration for the comparators must remain static. This means that the design must either remain applicable to only one set of data, or that it must regress to use general-purpose comparators like the comparator shown in Figure 1.5.

This example application shows how traditional ASIC development methods inhibit the potential that CCM-based applications may have. What such applications need is a way to continue to enable the dynamic capabilities of the underlying FPGA hardware in the deploy stage and beyond. The FPGAs by themselves are powerless to provide this
dynamic capability. They rely on external, usually software-based systems to determine their configurations. If a CCM-based application is to maintain dynamic characteristics to offer application flexibility or increased performance, then all or part of the capabilities of the CAD/EDA tools used in the design and debug stages of development must continue to be a part of the application software in the deploy stage and beyond.

To demonstrate the importance of retaining the CAD tool capabilities beyond the debug stage, consider the way in which the above example application might be executed by the final user. The end user might not be a hardware designer at all. For example, suppose the string pattern matching application is used by a biologist who analyzes the patterns in large database of DNA (deoxyribonucleic acid) or protein information. The string comparison application may provide an interface that allows the biologist to key in a target string for comparison. Then the biologist may tell the application to configure itself for comparisons with that string. The interface which the biologist uses simply appears like any other software application on a typical workstation. The application does not need to reveal that the underlying computation will be performed in custom-configured hardware. The application will produce a custom netlist of the hardware design, complete with the data of the target string embedded in the hardware design so that comparisons are optimized for it. Placement and routing tools specific to the FPGA hardware in the CCM then process that netlist to generate a configuration image for the FPGAs. This process may take anywhere from several minutes to a few hours. At this point, the hardware configuration is verified to ensure that it fits within the timing and resource constraints of the given CCM system. Once the configuration is verified and loaded into the FPGAs, the software portions of the application may then proceed to obtain values from biologist’s database and pass them to the CCM hardware to perform the comparisons. Simultaneously, the application software collects the output from the hardware and either passes it on to another system for analysis or stores it in a separate database for deferred analysis. Therefore, although the biologist does not necessarily have experience with digital hardware design, he or she may benefit from an application that speeds up the database comparisons so that they may finish in just a few hours instead of several months. Much of the application speedup was due to the inherit speed advantage that the hardware has over alternative software implementations.
However, if the application offers execution-time optimizations as in this example, the application performance may increase even more, making the CCM system a more attractive computing solution for the biologist.

1.6 Reconfigurable Computing Application Frameworks

Many modern CAD systems are excellent tools for ASIC development. However, the design methodology for ASICs is far different than that for good design of full CCM applications. The unique needs of CCM application design demand unique solutions. The purpose of this thesis is to present an alternative to typical CAD tools for CCM application development. The proposed solution to overcoming the limitations of CAD tools is not to use a monolithic application for development at all. The needs of CCM applications can be better addressed with application frameworks rather than large and complex CAD systems.

Such frameworks would be better suited to take advantage of the special features available in CCMs. Using these features from an application framework that supports them will help designers to easily develop custom design and debug environments that are specific to any given application. The preparations for deployment of each of those applications will occur simultaneously as the whole application, including both the hardware and software components, is built up in an integrated fashion. This thesis will refer to such frameworks as reconfigurable computing application frameworks, or RCAFs.

A reconfigurable computing application framework performs a number of roles. These roles help to fulfill the needs of the design, debug, and deploy stages of development for CCM applications. The RCAF may be used as a development tool for the hardware and software designs of a CCM application. It is a flexible tool that may be customized to create an application-specific development environment suitable for designing and debugging CCM-based applications. It is well-suited to provide support for the design and verification of the application hardware design and simultaneous development of the software, complete with integrated communications and data flow. This allows the RCAF to provide a framework for building up the software data path and user interface for the final CCM application to be deployed. Finally, the RCAF also includes added support for the
deploy stage of application development by establishing a framework for runtime execution support, including dynamic hardware configuration, as described in Section 1.5.

This thesis establishes a high-level specification for a reconfigurable computing application framework. This specification, given in Chapter 2, outlines the general features of the architecture and composition of an RCAF. Chapter 2 also give the justification for the design decisions in that architecture based on the needs of application development and the features of CCMs. Research associated with this thesis led to the implementation of such an RCAF. Chapter 3 describes the pre-existing foundation for that implementation: JHDL, an object-oriented hardware description language for FPGA design. JHDL acts as the principle hardware model and hardware interface for the other components in the RCAF implementation. Chapter 4 describes the RCAF implementation, called CVT, as well as the role that JHDL plays in it. Chapter 4 also contains details about the ways in which the CVT implementation of an RCAF supports the complete deployment of a CCM-based application. Chapter 5 contains a number of example applications built using this RCAF implementation. Finally, conclusions and recommendations based on this research are outlined in Chapter 6.
Chapter 2

Reconfigurable Computing Application Frameworks

The custom hardware configuration of ASIC designs enables them to perform specific tasks at very high speeds. However their functionality is limited to the purposes for which they are designed. Even ASIC-like FPGA designs are configured with “firmware” that is usually left untouched once the design is completed. Updating the firmware of such designs is often relatively difficult, if it is possible at all.

Contrasting with ASIC and ASIC-like FPGA designs, configurable computing machines are general-purpose computing platforms that may be used as the processing cores for a wide variety of applications. Loading a new hardware configuration into the FPGAs of a CCM changes its functionality\(^1\) Configuring the FPGA hardware of a CCM is much like loading in a new software program on a microprocessor-based machine. And much like software loading, CCM configuration and reconfiguration can be dynamic and interactive. This is made possible by the unique features of CCM hardware. These features were introduced in Chapter[1] and are repeated in Figure[2.1].

- Availability of the hardware at design time
- Programmability and re-programmability of the hardware
- Visibility into the hardware
- Controllability of the hardware

Figure 2.1: CCM benefits leveraged by RCAFs to improve application development

\(^{1}\) Some CCMs even support partial hardware reconfiguration. Partial reconfiguration only changes the functionality of parts of the FPGA or FPGAs, leaving the rest of the hardware configuration intact.
Figure 2.2: The RCAF interfaces between the CCM hardware and application software

An *Reconfigurable computing application framework* (RCAF) performs the following roles:

- It is a bridge between the CCM hardware and the CCM application software running on the host system. This relationship is demonstrated in Figure 2.2.
- The RCAF is a foundation and framework for both the hardware and software designs of CCM-based applications. That is, the RCAF is a source of reusable code and essential APIs to simplify building up significant CCM-based applications.
- The RCAF establishes a communications architecture foundation for the various components of CCM applications.
- The RCAF enables dynamic circuit configuration beyond the design and debug stages of application development. Even the final application deployed to the end user may retain the ability to adapt to user-provided data to optimize performance.

RCAFs support the dynamic, software-like features and operation of CCMs. Applications based on the RCAF may offer custom interfaces that match the needs of the individual applications, rather than the abilities or limitations of the the CAD tool. These applications may also remain dynamic and flexible to adapt to the changing needs of the end user. The RCAF interface model for interactions between CCM hardware and software enables a new type of development process for CCM applications. Because this design process is intended to enable developers to quickly design, debug, and deploy whole CCM
applications, it is more suitable to CCM application design than the process made possible by traditional CAD tools for FPGA design. A given application may customize and extend the RCAF architecture framework to quickly and easily build up the full application, including the hardware configuration, the supporting software data path, and the software user interface.

2.1 RCAF Interface Between CCM Hardware and CCM Software

An RCAF offers simple and accessible interfaces to all of the important features of targeted CCM platforms. The RCAF takes care of the low-level details of configuring and operating the CCM hardware. Ideally, an RCAF will work with virtually any CCM hardware platform. The same interface is available to the application developer, no matter what the target platform may be. This could present certain challenges where different platforms do not offer identical sets of features and controls. But when subsets of the features are identical or comparable, the RCAF APIs exposed to the developer are consistent across all CCM platforms. For most CCM platforms this includes operations such as configuring the FPGAs, cycling the main system clock, applying data to the CCM, and querying the hardware state of the CCM.

In this way, the RCAF functions much like conventional board-support software packages such as are provided with commercially available CCM platforms. However, the RCAF is not only a resource of reusable code for interacting with CCM hardware. It provides much more in its application programming interfaces (APIs). The RCAF also includes a system that aids in the interpretation of the hardware state as it relates to the original hardware design. It maintains a reference mapping between the circuit design components and their corresponding circuit elements in hardware. (See Figure 2.3) That is, the RCAF takes care of the overhead necessary to give the application developer a transparent view of the circuit structure and state while the circuit is executing in hardware.
2.2 RCAF API For Application Interfaces

Because the hardware view of the circuit provided by the RCAF is the same as the design view, circuit interaction components may be consistent across the design, debug, and even deployment stages of application development. These circuit interaction components may include schematic viewers, waveform viewers, and other traditional CAD circuit visualization components, as well as circuit execution controls, and especially custom circuit visualization components. Beyond even that, the RCAF contains APIs which aid in establishing a simple architecture to support application communications. This architecture creates a communications network that allows interactive components to send and receive messages to and from other interactive components as well as the hardware execution and state controllers. This architecture helps maintain a modular and scalable application system for easy maintenance and updating.

Because the RCAF is already integrated with the target hardware system, custom application interfaces may be tested and verified at full hardware speed. These interfaces are also useful for designing and debugging the hardware configuration of the application because they can provide intuitive means of interacting with the circuit and interpreting results.

Figure 2.3: The RCAF provides a mapping between the hardware design and the physical CCM hardware
2.3 RCAF Support for (Dynamic) Circuit Design

The RCAF is more than just an interface layer in a CCM-based application. It is also a circuit design tool for the CCM hardware configuration. The circuit design is built up programmatically and then translated into a format that can be accepted by technology-specific tools that prepare it to configure the CCM hardware.

The RCAF also has the ability to receive directives from the application to dynamically create a custom version of the CCM hardware configuration design. As explained in section 1.4, this enables CCM applications to optimize the hardware configuration of the CCM based on parameters and data supplied by the user. The RCAF itself is deployed with the final application. This gives even the end user access to dynamic, interactive circuit creation capabilities to significantly improve the performance of the application.

As mentioned in Section 2.1, the RCAF maintains a design entry-level model of the hardware configuration. The same RCAF model serves the purposes of design entry, hardware configuration, and hardware execution and simulation. This means that the dynamically established hardware design and the application interface software can maintain a consistent relationship through the APIs exposed by the hardware design model. This helps minimize the amount of code required to establish relationships between the hardware design and the software interfaces. This is a tremendous improvement over traditional CAD tools which usually convert a design-entry model into entirely different models for simulation and hardware configuration.

2.4 Designing, Debugging, and Deploying CCM Applications

RCAF features allow designers to improve the process of designing, debugging, and deploying CCM-based applications. Application development stages proceed within the same environment; the RCAF is an integral part of the application from design to debug to deployment. The final application also remains in the same environment. This continuity reduces the risk of having to restart the development process at various stages, as might occur when transferring a development project from one tool to another. This is because the application behaves the same across all boundaries between development stages. In
general, projects created using an RCAF are easier to develop and maintain than projects created with multiple development tools and environments.
Chapter 3

JHDL: A Basis for a Reconfigurable Computing Application Framework

The initial motivation for the creation of a reconfigurable computing application framework was based on the need to update and improve an existing circuit design tool. This tool, JHDL, already contained precursors of many features of an RCAF, such as those described in Chapter 2. However, as a continuing work in progress, JHDL had a few shortcomings, and JHDL was not previously viewed or utilized in an RCAF context. Goals and plans to correct JHDL’s shortcomings quickly evolved into an RCAF development project. This chapter describes the state of JHDL before the RCAF project began.

3.1 JHDL Design Fundamentals

3.1.1 The JHDL Design Data Model

JHDL’s data model is a Java-based structural representation of digital logic circuits. Although several different languages could have been used to help build up JHDL data models, the Java programming language offered a number of features that made it a good choice. Among those features are an object-oriented architecture that does not require the designer to be concerned with memory management, and Java reflection, which allows for dynamic creation, modification, and analysis of object classes. At the low level, a JHDL design is made up of instances of Java objects which represent primitive logic cells available in typical FPGA technologies. These primitive JHDL cells have behavioral models for simulation purposes. JHDL cell objects are subclasses of the Node or Cell classes.

1 For more information on JHDL and developments related to it, see [9, 10, 11, 5, 6].
depicted in the inheritance UML diagram shown in Figure 3.1. All JHDL cells have ports associated with them. Wire objects (instances of subclasses of the Wire class) are connected to cell ports to form interconnect between cells in a given level of hierarchy and between a cell and its parent or its children. This hierarchical build-up of cells helps not only create a modular design, but also permits the cells to be easily parameterized based on values provided by their parent cells. The structural approach of the JHDL design model may be used to create libraries of simple primitives as well as complex parameterized modules to help support high level design which targets FPGA devices. When simulating JHDL designs, aggregate cells may also have a single behavioral model to execute instead of the multiple behavioral models of the various leaf cells that make up its hierarchical structure.

![Figure 3.1: Inheritance structure of JHDL’s base classes.](image)

JHDL designs can either be built up by explicitly creating instances of primitive and compound cells and connecting them together or by invoking platform-independent methods of the generic JHDL Logic class. The platform-independent methods allow a design to be targeted to multiple technologies without any changes to the design. A technology-specific logic mapper is plugged into the back end of the Logic class. This mapper will translate the technology-independent Logic methods to methods that create instances of primitive cells and modules associated with the current technology. However, because the design entry through the Logic class is platform independent, this method of circuit design may not result in the optimal design for a given technology.
A JHDL design is created from within a JHDL testbench. The testbench typically determines the target technology of any technology-independent components of the circuit design. The testbench is contained within the JHDL hardware system. (JHDL’s hardware system is represented by the HWSystem class.) The hardware system acts as the gateway for all access to the circuit hierarchy, circuit state, and circuit control. It also creates a circuit simulator and establishes a relationship between the simulator and the JHDL circuit design. The relationships between the various components of the JHDL system are shown in Figure 3.2.

Figure 3.2: Hierarchy of the JHDL system.

3.1.2 JHDL Application Programming Interface

The hierarchy and relationships shown in Figure 3.2 are typically self-contained. However, the JHDL hardware system also exposes a simple application programming interface (API) that gives access to the circuit design, as depicted in Figure 3.3. JHDL designs are created using a general purpose programming language (as opposed to a closed hardware description language meant only for creating simulation models and netlists). Because of this, components external to the base JHDL system may easily be integrated with JHDL.

---

2 The use of the name testbench comes from a historical context. In some situations, i.e. when a JHDL design is in final form and executing in hardware, the name testbench is a misnomer.
through the API exposed by JHDL’s hardware system. These APIs are described in more detail below.

Figure 3.3: JHDL’s hardware systems exposes three types of APIs to external systems.

**JHDL’s Circuit Structure API**

The HWSystem class offers methods to access its child cell, as well as any arbitrary subcomponent in the circuit design. These methods are shown in Table 3.1. The `getTestBench` method simply returns a reference to the only design element child of the hardware system: the testbench. The `findNamed` method has a return type of `Nameable`. The `Nameable` class is the super class for all nodes and wires in a JHDL circuit design. The `findNamed` method may be used to acquire a pointer to any wire or cell of the design, even if it doesn’t happen to be explicitly connected into the circuit. The `Nameable` class ensures that all JHDL nodes have unique identifiers, or names, that can be used to obtain a pointer to them through the `findNamed` method.
Table 3.1: HWSystem methods for accessing circuit design cells

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Arguments</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>getTestBench</td>
<td></td>
<td>Cell</td>
</tr>
<tr>
<td>findNamed</td>
<td>String name</td>
<td>Nameable</td>
</tr>
</tbody>
</table>

JHDL cells in general also have a set of methods that can be used to access the circuit structure of the design. These methods are shown in Table 3.2. The methods getParents, getChildren, and getDescendents, may be used to obtain hierarchy information (but not connectivity and port information) about the JHDL design. The methods getPortRecords, getPortRecord, getAttachedPort, getAttachedWire, getSinkWires, and getSourceWires, can be used to determine information about the wires connected to the cell at the ports. The methods getWires, getCellNetlist, and getFlatNetlist can be used to obtain pointers to the wires that connect the children cells of a given cell. Finally, the method getSystem provides a convenient method of obtaining a pointer to the top-level JHDL system from any cell.

Table 3.2: JHDL Cell methods for accessing circuit design structure

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Arguments</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>getParent</td>
<td></td>
<td>Node</td>
</tr>
<tr>
<td>getChildren</td>
<td></td>
<td>NodeList</td>
</tr>
<tr>
<td>getDescendents</td>
<td></td>
<td>CellList</td>
</tr>
<tr>
<td>getPortRecords</td>
<td></td>
<td>PortRecordList</td>
</tr>
<tr>
<td>getPortRecord</td>
<td>String portName</td>
<td>PortRecord</td>
</tr>
<tr>
<td>getAttachedPort</td>
<td>Wire w</td>
<td>String</td>
</tr>
<tr>
<td>getAttachedWire</td>
<td>String portName</td>
<td>Wire</td>
</tr>
<tr>
<td>getSinkWires</td>
<td></td>
<td>WireList</td>
</tr>
<tr>
<td>getSourceWires</td>
<td></td>
<td>WireList</td>
</tr>
<tr>
<td>getWires</td>
<td></td>
<td>WireList</td>
</tr>
<tr>
<td>getCellNetlist</td>
<td></td>
<td>Netlist</td>
</tr>
<tr>
<td>getFlatNetlist</td>
<td></td>
<td>Netlist</td>
</tr>
<tr>
<td>getSystem</td>
<td></td>
<td>HWSystem</td>
</tr>
</tbody>
</table>
The \texttt{Wire} class also has a set of methods that may be used to obtain circuit structure information from a JHDL design. These methods are shown in Table \ref{tab:wire-class-methods}. The first six methods return references to the sink or source cells of a given wire object. The \texttt{getWidth} method returns the number of bits that make up a given wire object. The \texttt{getWire} and \texttt{range} methods offer convenient means of obtaining references to one or multiple bits of a multi-bit wire. The \texttt{getParent} method may be used to determine the owner of a given wire. The \texttt{getSystem} method of the \texttt{Wire} class is another convenience method for obtaining a reference to the top-level system object for the JHDL design.

\begin{table}[h]
\centering
\caption{\texttt{Wire} class methods for accessing circuit design structure}
\begin{tabular}{|l|l|l|}
\hline
\textbf{Method Name} & \textbf{Arguments} & \textbf{Return Type} \\
\hline
\texttt{getSinkCell} & & \texttt{Cell} \\
\texttt{getSinkCells} & & \texttt{CellList} \\
\texttt{getAllSinkCells} & & \texttt{CellList} \\
\texttt{getSourceCell} & & \texttt{Cell} \\
\texttt{getSourceCells} & & \texttt{CellList} \\
\texttt{getAllSourceCells} & & \texttt{CellList} \\
\texttt{getWidth} & \texttt{int index} & \texttt{Wire} \\
\texttt{getWire} & \texttt{int highInd, int lowInd} & \texttt{Wire} \\
\texttt{range} & & \texttt{Node} \\
\texttt{getParent} & & \texttt{HWSystem} \\
\hline
\end{tabular}
\end{table}

The subclasses of the abstract \texttt{TechMapper} class can write the structure of a design to a netlist file. Table \ref{tab:techmapper-methods} shows the methods to generate these netlists. The output format depends on the particular \texttt{TechMapper} used. For example, the \texttt{TechMapper} classes for the Xilinx FPGAs generate EDIF (electronic design interchange format) netlists for use by the Xilinx back-end synthesis tools. \texttt{TechMapper} classes also exist to create netlists in VHDL format (very high-speed integrated circuit hardware description language) and Opt format (used for the Teramac CCM produced by Hewlett-Packard).
### JHDL’s Simulation API

JHDL simulation is set up and managed by the JHDL hardware system and its corresponding simulator. The default simulator is a statically-scheduled simulator. This makes simulation significantly faster than what is possible with an event-driven simulator. However, it has the disadvantage of limiting the types of circuits that may be simulated; specifically, circuits that contain asynchronous loops may not be simulated with JHDL’s default simulator. JHDL simulation is simply performed by executing the code that makes up the circuit design. Each circuit element contains simulation code within either a clock or propagate method. The cell objects themselves determine their own behavior when these methods are invoked by the simulator. Typically, each cell will read the values on its input wires and make decisions about what values to put on its output wires. The order of execution of simulation code is determined up front by the simulator schedule. The simulator schedule is divided into steps. The number of steps depends on the number of clocks in the circuit design, as well as their duty cycles and relative frequencies. A full simulation schedule, or complete set of steps, is a simulation cycle.

The JHDL hardware system makes a simulation execution control API available. This API allows the simulator to be “stepped” and “cycled.” The relevant methods (cycle, step, and skip) are shown in Table 3.5. Table 3.5 also shows the simulator callback methods in the hardware system. These methods allow any class which implements JHDL’s SimulatorCallback interface, shown in Table 3.6, to receive feedback from the simulator. The feedback comes in the form of notification of the completion of significant simulator events. The simulatorReset method of all registered SimulatorCallback objects will be invoked whenever the simulator is reset. The simulatorUpdate method will be invoked after each step of the simulator. The user or testbench may tell the simulator
to step or cycle for multiple clock steps or cycles. The simulatorRefresh method will be invoked only after all of the steps or cycles of a given call to the step or cycle method have completed. Therefore, the simulatorRefresh method is useful for classes that do not need to observe the circuit after every simulator event. For example, waveform viewers should receive notification of every simulation event to update their database of wire value histories, but circuit state-annotated schematic viewers, which typically only show a single state at a time, only need to perform their updates after a simulation run of multiple steps or cycles has completed. The skip method of HWSystem acts just like the step method, except the notification of SimulatorCallback objects is temporarily disabled for the duration of the number of steps indicated by the given argument. The forceSimulatorCallbackRefresh method is useful for updating SimulatorCallback objects after using the skip method.

Table 3.5: HWSystem methods for simulation control and feedback

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycle</td>
<td>int no_clocks</td>
</tr>
<tr>
<td>step</td>
<td>int no_steps</td>
</tr>
<tr>
<td>skip</td>
<td>int no_steps</td>
</tr>
<tr>
<td>addSimulatorCallback</td>
<td>SimulatorCallback sc</td>
</tr>
<tr>
<td>removeSimulatorCallback</td>
<td>SimulatorCallback sc</td>
</tr>
<tr>
<td>forceSimulatorCallbackRefresh</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.6: The SimulatorCallback interface for simulator feedback

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>simulatorReset</td>
<td></td>
</tr>
<tr>
<td>simulatorRefresh</td>
<td>int cycle, int step</td>
</tr>
<tr>
<td>simulatorUpdate</td>
<td>int cycle, int step</td>
</tr>
</tbody>
</table>
The `Wire` class includes methods for stimulating wires during simulation. These methods are shown in Table 3.7. The methods with `boolean`, `int`, and `long` arguments are used to apply values to wires up to one, thirty-two, and sixty-four bits wide respectively. The methods with `int[]` and `BV` arguments may be used to stimulate wires of arbitrary widths. The JHDL `BV` class is a general-purpose bit vector class that also has a number of extra features convenient for binary number representation. Wire drivers may use the `putTriState` methods to indicate that they are not driving the wire at a given time. The source `Cell` for these methods must be a valid driver for the wire. For top-level input port wires, the `TestBench` is typically the only valid driver. However, the `TestBench` may employ other special-purpose instances of the `Cell` class to drive wires according to specialized behavioral models. See, for example, the `Stimulator` class described in Section A.1.

### Table 3.7: Wire class methods for stimulating wire objects

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>put</td>
<td><code>Cell source, int val</code></td>
</tr>
<tr>
<td>putA</td>
<td><code>Cell source, int[] val</code></td>
</tr>
<tr>
<td>putB</td>
<td><code>Cell source, boolean val</code></td>
</tr>
<tr>
<td>putBV</td>
<td><code>Cell source, BV val</code></td>
</tr>
<tr>
<td>putL</td>
<td><code>Cell source, long val</code></td>
</tr>
<tr>
<td>putTriState</td>
<td><code>Cell source, int val</code></td>
</tr>
<tr>
<td>putTriStateA</td>
<td><code>Cell source, int[] val</code></td>
</tr>
<tr>
<td>putTriStateBV</td>
<td><code>Cell source, BV val</code></td>
</tr>
<tr>
<td>putTriStateL</td>
<td><code>Cell source, long val</code></td>
</tr>
</tbody>
</table>

**JHDL’s Circuit State API**

The state of a JHDL design during simulation is usually accessed through its wire objects. The JHDL `Wire` class has a number of methods that return the current value on the wire. Each of these methods requires as an argument an object that implements the JHDL `Browser` interface. The `Browser` interface has no methods; it is merely a tagging
interface. A class that implements the `Browser` interface is identified as being a legitimate wire state inquirer. Besides `Browser` objects, only valid sink cells are allowed to query the state of a wire object. The wire state accessor methods, shown in Table 3.8, offer a variety of return types. The `boolean`, `int`, and `long` return types are meant to give the states of wires up to one, thirty-two, and sixty-four bits wide respectively. The `int[]` and `BV` return types may be used to support wires of arbitrary widths. The `getX` and `getZ` methods in Table 3.8 return special bit masks that indicate bits of the wire that are currently in an undetermined or high-impedance state respectively.

Table 3.8: JHDL Wire methods for accessing circuit state

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Arguments</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>Browser b</td>
<td>int</td>
</tr>
<tr>
<td>getA</td>
<td>Browser b</td>
<td>int[]</td>
</tr>
<tr>
<td>getB</td>
<td>Browser b</td>
<td>boolean</td>
</tr>
<tr>
<td>getBV</td>
<td>Browser b</td>
<td>BV</td>
</tr>
<tr>
<td>getBV</td>
<td>Browser b, BV bv</td>
<td>BV</td>
</tr>
<tr>
<td>getL</td>
<td>Browser b</td>
<td>long</td>
</tr>
<tr>
<td>getX</td>
<td>Browser b</td>
<td>int</td>
</tr>
<tr>
<td>getXA</td>
<td>Browser b</td>
<td>int[]</td>
</tr>
<tr>
<td>getXB</td>
<td>Browser b</td>
<td>boolean</td>
</tr>
<tr>
<td>getXBV</td>
<td>Browser b</td>
<td>BV</td>
</tr>
<tr>
<td>getXBV</td>
<td>Browser b, BV bv</td>
<td>BV</td>
</tr>
<tr>
<td>getXL</td>
<td>Browser b</td>
<td>long</td>
</tr>
<tr>
<td>getZ</td>
<td>Browser b</td>
<td>int</td>
</tr>
<tr>
<td>getZA</td>
<td>Browser b</td>
<td>int[]</td>
</tr>
<tr>
<td>getZB</td>
<td>Browser b</td>
<td>boolean</td>
</tr>
<tr>
<td>getZBV</td>
<td>Browser b</td>
<td>BV</td>
</tr>
<tr>
<td>getZBV</td>
<td>Browser b, BV bv</td>
<td>BV</td>
</tr>
<tr>
<td>getZL</td>
<td>Browser b</td>
<td>long</td>
</tr>
</tbody>
</table>

JHDL also provides a means of extracting state information from memory elements of a JHDL design. JHDL cells that perform the role of memory elements implement
MemoryInterface (Table 3.9) or other related interfaces. The methods of these interfaces are used to obtain information about the dimensions and contents of a memory cell.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Arguments</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSize</td>
<td>long</td>
<td></td>
</tr>
<tr>
<td>getMemoryWidth</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>getMemoryElement</td>
<td>int address</td>
<td>BV</td>
</tr>
<tr>
<td>getMemoryRange</td>
<td>int startAddr, int elements</td>
<td>BV[]</td>
</tr>
</tbody>
</table>

### 3.2 Hardware Support in JHDL

JHDL has been used as a tool to unify the designing and debugging of FPGA-based hardware designs with in-hardware execution support, as shown by [11, 5, 6]. Thus far, in-hardware execution support in JHDL has been made possible by utilizing hardware readback capabilities found in many programmable logic devices produced by Xilinx Corporation. The readback data from Xilinx FPGAs includes the state of all memory and register components in the hardware, as well as several key combinational logic values. With the assumption of a circuit design that does not contain any asynchronous loops, the memory and register state information is sufficient to discern the state of any signal of interest in the design. Consider, for example, the model of a circuit design shown in Figure 3.4. In an FPGA, the combinational logic of this design would most likely be mapped to a single lookup table, as in Figure 3.5. The readback information from the FPGA hardware would likely only include the output values of the three registers. This information is enough to interpolate not only the value of the output signal of the circuit, but also all of the intermediate values. Even if the readback information did include the value of the output wire, intermediate values, which may be of interest when debugging the circuit, would be lost without this interpolation technique. This technique, therefore, offers full circuit visibility to the designer.
Figure 3.4: A design model of an FPGA-based circuit.

Figure 3.5: Representation of the physical implementation of an FPGA-based circuit.
In JHDL, interpolation of intermediate logic values is done using the same hardware system model and simulator that are used during simulation without hardware execution. This means that the system architecture shown in Figure 3.2 is the same whether the circuit is being simulated in software, or executed in hardware. The only difference is that in simulation mode, register and memory elements update their own values, but in hardware mode, those elements are updated by the simulator with data provided by the hardware readback. However, correlating physical hardware components with the circuit design model in software is not necessarily trivial. Graham[6] showed that matching the software model with the hardware readback data can be accomplished by carefully tracing the mapping of the original design through FPGA vendor tools to the final hardware configuration. The result is a hardware symbol table comparable to symbol table data produced by software compilers. This hardware symbol table provides a map between the elements of the circuit model and their corresponding values in the readback data. With this map in place, the JHDL simulator may derive the information it needs from the hardware readback to build up an image of the circuit state.

With the same system model for both simulation and hardware execution, many of the APIs shown in Figure 3.3 also remain the same. In particular, the circuit structure and circuit state APIs are identical. Only the simulation control API may change slightly in hardware mode. However, for the most part, this still allows the same design and debug interface to be used whether the design is being simulated or executed in hardware.

3.3 JAB: The JHDL Circuit Browser

3.3.1 Interactive Circuit Visualization

To facilitate circuit design and debug, JHDL includes an interactive and graphical tool called JAB (Just Another Browser). JAB provides an environment from which the designer may verify the structure and operation of a JHDL design. JAB’s main graphical viewer (shown in Figure 3.6) provides both a hierarchical view of the circuit (presented in a familiar tree format) and a table of wire and port information for the currently-selected cell. This main viewer also offers menus that give access to other helpful graphical viewers such
as schematic viewers (Figure 3.7), memory viewers (Figure 3.8), and waveform viewers (Figure 3.9).

Figure 3.6: JAB’s circuit hierarchy browser.

To furnish a system layer from which input to the design may be provided and output analyzed, all JHDL designs must be contained in a testbench. To ease the designer’s burden of having to provide a testbench for all designs, especially for small and simple designs, JAB provides a dynamic testbench that can be used as the parent of most designs. This testbench, called TbOne, can be used to create top-level wires for the design and provide stimuli for the design inputs and basic output verification. Although it has some limitations, TbOne provides a convenient means of quickly and easily verifying the operation of most small JHDL designs.

3.3.2 Simulation and Execution Control

JAB also includes a command-line interpreter for executing commands that perform operations not offered by the graphical viewers as well as a few commands that are
Figure 3.7: JAB’s design schematic viewer.

Figure 3.8: JAB’s memory viewer.

Figure 3.9: JAB’s waveform viewer.
found in the viewers. The JAB console (shown in Figure 3.10) shows the history and feedback of commands. Textual commands may be stored in and executed from log files for convenience in scripting repetitive simulation actions.

![JAB's command line console](image)

Figure 3.10: JAB’s command line console.

Simulation control in JAB is achieved with both textual commands and graphical controls. Specifically, cycle and step commands are accessible from both the graphical viewers (typically as buttons) and the command line interpreter. When using Tbone as the design testbench, additional text commands may be used to control the input values to the circuit design.
3.3.3 JAB Customizability

In designing and debugging many projects, it is useful to customize the development environment to suit the needs of the design and the style of the developer. JAB provides a few different means of customizing its interactive environment. For example, Figure 3.11 shows a schematic of a design that has additional code to define custom schematic nodes. In this case, the appearance of the flip-flops changes according to their present state. To create custom schematic nodes, a JHDL design element needs only to implement the two methods of the UserDefinedSchematic interface shown in Table 3.10. The UserDefinedNode class that is passed to these methods has a number of methods to draw arbitrary shapes and text to create virtually any appropriate visual representation of the design element. One particularly useful application of the UserDefinedSchematic interface is in displaying the current state of control units with a textual label rather than cumbersome numerical outputs. Such custom visualizations of the circuit during simulation may allow the developer to process feedback about the circuit operation more quickly.

Table 3.10: The UserDefinedSchematic interface for custom schematic nodes

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>initUserDefinedNode</td>
<td>UserDefinedNode udn</td>
</tr>
<tr>
<td>paint</td>
<td>UserDefinedNode udn</td>
</tr>
</tbody>
</table>
JAB also allows the developer to customize its command line interpreter. The user can register aliases for commonly used commands. For example, if a particular design only generates meaningful output after a certain number of clock cycles, the user may register a simple command such as `c4` to perform the same function as `cycle 4`. The command line interpreter will also automatically execute a JAB initialization file (`.jabrc`) if it is found in the user’s home directory. This file can be used to easily execute common startup commands and to initialize favorite aliases.

JHDL designs may also use additional custom components that are not necessarily related to JAB. For example, custom circuit visualization components can be linked into the JHDL hardware system through the `SimulatorCallback` interface. A simple example of such a component is shown in Figure 3.12. This custom component provides an alternative view of the same circuit shown in Figure 3.11. However, this view of the circuit provides a more concise visualization of the state of the flip-flops. The TestBench which contains the JHDL design was used to create an instance of this custom visualization component. The testbench also registered the component with the JHDL hardware system as a `SimulatorCallback` (see Table 3.6). With feedback of major simulation events through the `SimulatorCallback` interface, and circuit state information accessed through the testbench, this custom component is able to generate its custom view and update it as simulation proceeds. Such custom views can improve circuit verification dramatically by programmatically obtaining and analyzing circuit state information to present to the user. This automated circuit visualization saves the designer time and effort that would be required to navigate and interpret conventional tools such as schematics and waveforms.

![Custom View](image)

Figure 3.12: An example custom circuit visualization component using the `SimulatorCallback` interface.
Other means of customizing the circuit verification of JHDL designs can be done through the testbench. Because the testbench is also written in the same general-purpose programming language as the design itself, it is natural and straightforward to implement custom components to interact with the simulation process. For example, custom components created by the testbench could manipulate the inputs to a circuit design. This could be done programmatically or interactively.

3.4 JHDL and JAB: A Precursor to an RCAF System

Features of JHDL and JAB described above created the original inspiration for the notion of reconfigurable computing application frameworks. Indeed, many of the characteristics of an RCAF described in Chapter 2 were already present in JHDL and JAB. In particular, JHDL and JAB can offer an interface between the hardware design model and the actual hardware, relatively simple circuit execution control and feedback, and dynamic circuit design and creation.

3.4.1 Drawbacks to the JHDL and JAB System

However, the JHDL/JAB system has a few characteristics and shortcomings that prevent it from being used as a good RCAF. JHDL itself works well as a base design model. However, to use JHDL alone would still require building up an entirely new system on top of it for every new application.

JAB is an inappropriate framework for the following reasons. JAB is generally a monolithic software system, and as such does not scale well as would be required by a good application framework. JAB was developed with an ad hoc style that created an extensive set of interdependences between its subcomponents. This makes JAB less modular and more difficult to maintain and update. JAB is also mostly a closed system. Outside of the few customizability features presented in Section 3.3.3, JAB offers no means of directly linking in extra components to offer custom application features and interactive elements.

Using JAB with JHDL is an excellent means of designing and debugging hardware configurations for FPGAs and CCMs. However, it is not adequate to perform the role of an RCAF in designing, debugging, and deploying complete CCM-based applications. That is
because JAB is basically a stand-alone application with minimal support for customizing it. Also, JAB is a large, monolithic tool that would require a great deal of unnecessary overhead for most applications.

### 3.4.2 Lessons From JAB

Despite its shortcomings, JAB provided a number of lessons in the creation of an RCAF. With its network of internal component dependences, JAB was difficult to maintain. Updating, adding, or removing any one component was difficult to do without upsetting several other parts of the system. JAB also lacked a centralized control unit; instead, components simply acted individually and inconsistently to accomplish tasks and communicate with each other.

The difficulty in maintaining JAB as well as the lack of a centralized control unit for the system highlighted the necessity of creating a simpler and more modular architecture. The new architecture should be scalable to fit a variety of applications. It should also allow for simpler integration of a wide variety of custom circuit and application interaction components. Finally, the new architecture should be flexible and small enough to easily be deployed along with complete CCM applications.
Chapter 4

Circuit Visualization Tool (CVT)

The JHDL-based implementation of a reconfigurable computing application framework is called Circuit Visualization Tool, or CVT. CVT borrows a great deal of code from the JAB tool. However, because the CVT architecture and its usage model are radically different from those of JAB, it is a distinct and unique tool with unique features that constitute not only a significant improvement over JAB, but also over current tools for CCM application development in general.

4.1 CVT Architecture

The CVT architecture is based on sound principles of user interface design such as those described by Olsen[12] and used in significant user interface toolkits such as Java Swing[13]. This user interface architecture is based on distinct components that act as the model, view, and controller of the user interface. The model is the fundamental data being manipulated by the system or the computation being performed. The view is the system component that determines how data is presented to the user. The controller is the portion of the system that receives user input and interprets it to perform system tasks. These tasks may perform manipulations of the data or they may change the way in which the view displays information about the data. The model-view-controller architecture, shown in Figure 4.1, divides the functionality of a system along boundaries that allow for simple system production and maintenance.

The model-view-controller architecture also establishes a natural communications network between its three components. The implementation for the CVT internal communication network is patterned after the listener model commonly used in user interface
architectures, notably in the Java Swing classes. The CVT version of the model-view-controller architecture is shown in Figure 4.2. The following three sections describe how the components in this figure relate to the model-view-controller architecture.

4.1.1 The CVT Model

The underlying model for CVT is implemented by JHDL. The CVT RCAF simply uses the default JHDL system without any modifications. The various APIs described in Section 3.1 and depicted in Figure 3.3 offer sufficient access to information about the model in order to support the model view and the controller. Manipulations to the model come in the form of stepping the circuit clock and applying values to the circuit inputs.

4.1.2 The CVT Controller

To receive and interpret user input and execute corresponding system-wide tasks, CVT has a centralized controller. The CVT controller is contained in a class named Broker. The Broker class gets its name from its role as an intermediary between the other major components of the CVT RCAF. The Broker is the only RCAF component with a system-wide view of all the other components. This centralized location of system
information is an improvement over the network of component references found in the JAB system described in Section 3.3. This system simplifies CVT and makes it easier to maintain. This also makes CVT modular and scalable. That is because only one class needs to be modified to change the set of features and components included in CVT to support a given application.

The Broker class contains all of the methods to perform major system tasks. Such tasks include creating and disposing viewers and managing communication between viewers. The Broker also performs manipulations to the application model; i.e. it invokes the reset, step, and cycle commands to control circuit execution and it applies values to the circuit inputs.

The Broker class works closely with a command interpreter. The command interpreter accepts and processes textual commands from the application user. The result is the invocation of corresponding methods of the Broker class. The set of commands available is therefore determined by the Broker class. In graphical environments, the Broker creates listeners which translate low-level events from the viewers (e.g. button presses, mouse clicks, and key strokes) into text commands to give to the command interpreter. All executed text commands are stored in a history log. This log may be retained, modified as

Figure 4.2: CVT Architecture.
necessary, and executed again later, making CVT scriptable. The system of textual commands also helps make CVT modular and extensible. The textual commands separate the Broker from any specific application interface element. A given interface element may be exchanged for any other element that happens to be able to generate the same types of text commands. This helps to minimize the maintenance required for the Broker class.

4.1.3 The CVT View

CVT viewers offer application-specific presentations of the circuit design. They also offer a portal through which the user may apply inputs to be passed to the CVT controller. A default CVT configuration for general-purpose circuit visualization includes a standard set of viewers (e.g. schematic viewers, circuit hierarchy viewers, memory viewers, etc.) Each unique CCM application may easily add to or remove from this default set of viewers as appropriate for the application. Each custom viewer merely needs to follow CVT’s method for generating events to work with the Broker’s listener model for communication. Each viewer should provide a listener interface for these events. Implementations of the listener interface may be used to translate action events into commands for the command interpreter. The CVT system imposes no other restrictions on custom viewers. It is up to the developers to decide on implementation details related to custom viewers; although the model-view-controller architecture does provide a good standard to follow on the viewer level as well.

4.1.4 Integrating CVT Model, View, and Controller

Figure 4.3 shows an example interaction between the various components of the RCAF architecture. In this example, an application consists of the JHDL circuit model, the CVT Broker class, the command interpreter, and three application viewers. The Broker has already created an instance of a listener that can receive events from the schematic viewer on the right. This listener translates events from the schematic viewer into textual commands for the command interpreter. In this example scenario, the user double-clicks with a mouse pointer on a portion of the schematic viewer that displays a wire in the circuit design. This generates an event which is picked up by the GUI listener
registered with the schematic viewer. The event includes the type of action as well as the object of that action. Because the **Broker** has a total-system view, it created a listener with the capability to translate this event into the textual command that is most appropriate for the application. The **Broker** registered the listener with the viewer so that it would receive the double-click event, perform the translation, and give the textual command to the command interpreter. When the command interpreter receives the command, it determines the appropriate **Broker** method to call as well as the correct arguments. The method called in this case tells the **Broker** to display appropriate views of the wire of the given name. The **Broker** knows of two other viewers which should be affected by that command and tells them to update their output to include the new wire. Those viewers then query the necessary information from the circuit model. The model then returns a reference to that wire, including its state information as determined by the simulator or CCM hardware. The two viewers then use that information to update their displays.

In general, the **Broker** holds the central position in the CVT system. Modifications to the **Broker** change the set of available CVT viewers. Since the **Broker** knows which viewers may exist in the system, and their relationships to itself and to each other, it knows how to configure the command interpreter and the viewer listeners. Having just
this one class with information that makes up a system-wide view makes CVT easier to
maintain and update. The single system controller makes removing and adding viewers to
the system and managing system communication simple.

4.2 Generic Circuit Verification Tool

CVT’s original purpose was to replace the JAB tool as a default circuit verifica-
tion tool. A circuit can be loaded into CVT and then simulated or executed much like in
JAB. As with JAB, CVT has a hierarchical tree view, a port and wire table, and a com-
mand interpreter console. These features are contained in the main CVT viewer shown in
Figure 4.4.

![CVT's main circuit view.](image)

CVT is not a feature-for-feature duplication of JAB. CVT’s improved architecture
led to an improved interaction with the viewers. Many features in JAB were only accessible
through menu items. CVT’s viewer listener model, however, allows many actions to be
more intuitive. For example, the user may wish to use the mouse to double-click on an item in the circuit hierarchy. This causes an event to be dispatched which is caught by a listener object. That listener then translates the event into a textual command to send to the command interpreter part of the controller. If the item clicked is a normal cell element of the design, the listener generates a command to create a schematic view of that cell; if the item is a memory, a command to create a memory viewer is generated instead. Similarly, the user may add wires to a waveform viewer, open new viewers by clicking on items in schematic viewers, and perform simulation operations by clicking on step and cycle buttons. The details of how this works are described in the example in Section 4.1.4. This system was set up to enable the user to “feel” around the circuit visualization tools to find the desired information without having a great deal of prior information. This helps to eliminate what Olsen calls the “gulf of execution”[12]. A gulf of execution occurs when a user interface is such that a user has difficulty determining how to perform a particular action to obtain a desired result from the system. In CVT, the user simply performs actions within the viewers that appear to make sense to someone with minimal prior knowledge. The listeners and Broker then perform a task that makes the most sense for each action. The CVT system also allows for multiple means to accomplish the same task. For example, many actions may be performed with mouse clicks, menu commands, keyboard shortcuts, or commands typed directly in a console associated with the command interpreter.

4.3 Custom Circuit Interaction Components

Since CVT is written in a general-purpose programming language, there are many ways to extend its capabilities. All that is required is a reference to the circuit design from which all circuit structure and state information may be extracted. Indeed, the CVT architecture is so simple and small, it would not take too much to recreate entirely the functionality of the default viewers and controller. Should an application developer desire to do that, the CVT system can still provide various resources to help build up an application.

CVT provides a toolbox of resources useful for building up custom circuit interaction environments for circuit design, debug, and execution. These resources include the command interpreter, generic viewer frames, wire stimulators, and specific viewer frames
Figure 4.5: CVT provides several default viewers with intuitive interfaces.

such as those shown in Figure 4.5. They also include corresponding modular tools in the form of panels. Such panels may be included and arranged as necessary in custom frames, tabbed panes, and even Java Applets (such as in [14]). This is a convenient way of mixing existing default tools with custom tools in an arrangement that is customized and appropriate for a given application. Various CVT components that may be used in other applications are listed in Appendix A.

Using CVT components in this way makes the framework into a kind of grab bag of resources for fast application prototyping. This is a good way to quickly build up custom design and debug environments for FPGA hardware designs. This enables a great deal of code reuse, although it may not be particularly well-suited to creating large, robust CCM-based applications.
4.4 Creating Full Custom CCM Applications by Extending CVT

Creating significant and solid CCM-based applications with CVT is best done by extending the default model-view-controller architecture established by the Broker class and related CVT components. This architecture is specifically designed to be scalable[12]. With the CVT framework as a basis for a CCM application architecture, developers may more easily create a system that is more maintainable and modular.

4.4.1 Custom CCM Application Model Component

The model for a CVT-based custom CCM application might be nothing more than a typical JHDL circuit design. However, many applications may wish to include extra features in the circuit model to accommodate application requirements or platform-specific features or needs. As instances of Java classes, JHDL designs may include extra fields and methods to serve whatever purpose necessary in the CCM application. For example, methods may exist to assist viewers in locating key circuit components or interpreting encoded values. As an example of this, see the edit distance application in Section 5.2. JHDL circuit models may also include design features that reflect the architecture of a specific CCM platform. Several projects related to JHDL have done this by creating “board models” for various CCM architectures that include such features as FIFO and memory interfaces. With these interface elements in place, an application-specific design may be “dropped in” the board model. This allows the same board model to be reused for various applications, thus giving application designers a head start in making a particular design work on the given target platform.

4.4.2 Custom CCM Application View Components

Application viewers may be of any form necessary to implement the application specifications. However, viewers that follow the pattern established by the CVT architecture help build up the application in a modular, scalable manner. This pattern contains three communication points for each viewer:

- model queries
• outward inter-component communication

• inward inter-component communication

These communication points are shown in Figure 4.6. Viewer model queries are direct communications between the viewer and the application model (circuit design). Viewers may maintain references to the circuit model as necessary to query circuit structure and state to garner information to be displayed in the viewer. However, this direct communication should not perform model manipulations, such as clock stepping or wire value manipulations. Viewer controls that are intended to perform such model manipulations or to communicate with other viewers should do so through the outward communication point. This communication point directs events and messages to viewer listeners, which then translate the messages into textual commands for the command interpreter in the CVT controller. The viewer receives messages from other CVT component through the Broker. Because the Broker is the only class that has a reference to the viewer, it is easier to exchange the viewer for another or to update its communication API, since only the Broker class needs to be updated to accommodate the change.

Figure 4.6: Communications to and from CVT viewers.
4.4.3 Custom CCM Application Controller Component

The custom application controller may be implemented by extending the Broker class. Extending the Broker class involves overriding the two methods shown in Table 4.1. The constructor method is a convenient place to create viewers known to the Broker and to perform other initialization. The registerCLICommands method gives the custom Broker an opportunity to create custom textual commands for the command interpreter (CLInterpreter). The default Broker also contains several other methods which perform the various tasks found in the default CVT system. (See Section A.1 in Appendix A for a listing of these methods.) The needs of the new textual commands and the set of application viewers may be accommodated by overriding these methods or by creating new methods.

Table 4.1: Broker methods for extending the CVT system

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>new Broker (constructor)</td>
<td>_HWSystem system, CLInterpreter interp</td>
</tr>
<tr>
<td>registerCLICommands</td>
<td>_HWSystem system, CLInterpreter interp</td>
</tr>
</tbody>
</table>

4.5 Application Deployment

As described in Section 1.2.3, deploying a CCM-based application involves preparing the application for use by the end user. This includes establishing the relationships between the various components of the application system as they will be at runtime for the final user as opposed to the application developer. For CVT this means executing the CVT and other application code to build up the hardware model, the controller component, and the application viewers, and creating the communications system among them. Application deployment also means establishing the relationship between the hardware model and the CCM board support software. This implies the existence of a minimum runtime environment sufficient to support the needs of the application. CVT design considerations related to establishing the runtime environment are discussed in Section 4.6. The remainder
of this section discusses the means of creating the model, view, and controller components and integrating them at execution time.

Any Java class may be used to create and combine the various components of the CCM application and act as a point of application execution. Typically, the TestBench portion of the circuit design acts as this deployment class; however, the application execution is performed in static methods, so it does not really matter in which class it happens, as long as the class has access to the appropriate methods. Application integration and execution consists of creating the circuit design (by instancing a JHDL HWSystem and the design TestBench) and creating an instance of the Broker. This is the point at which user-provided data may be used to configure the circuit design to include data folding and other optimizations, as described in Section 1.4. Some applications may wish to also create one or more viewers in the deployment methods, but this may also be performed in the constructor of the Broker as described in Section 4.4.3 Once the viewers have been created, the flow of execution passes to them and the application initialization is complete.

The final application only needs to include the features of the CVT RCAF that are necessary. For example, JHDL contains libraries that target a variety of FPGA technologies (e.g. XC4000 series, Virtex, CSRC, etc.), but if a CCM application is only intended for use on a particular platform, the extraneous libraries may be excluded. Likewise, default viewers not used in the application may be left out. The current implementation of the CVT RCAF scales down to only two to four megabytes compressed. This includes all of the necessary JHDL classes to build up the circuit design model, the JHDL hardware system and simulation environment, netlisting tools, all of the controller classes, and all of the viewers necessary for the system. Appendix B shows the sizes of the various groups of components that may be included in a configuration of CVT. When compared to the tens or hundreds of megabytes for traditional, monolithic CAD tools, this is a small application foundation that may reasonably be deployed to users.

4.6 Tradeoffs and Considerations in Application Deployment

A number of considerations related to the deployment of CVT-based applications must be made. The requirements for the runtime environment needed to support CCM
applications based on CVT required tradeoffs to offer the best support for runtime flexibility and application development simplicity. The important issues related to this include ensuring that the framework and application code can execute on the host. They also include ensuring that the CVT framework can adequately control the CCM hardware. Finally, deployment considerations include establishing a means of modifying the hardware configuration at execution time should the application require such a feature in order to optimize the application performance.

4.6.1 Framework and Application Code Execution

The first deployment consideration is the runtime execution environment for the CVT framework code and for the application code built on top of it. CVT is written in Java for ease of development and for maximum compatibility with JHDL. A significant disadvantage of using Java is that code execution is slower (because Java is an interpreted language rather than native code). Java applications also require the Java Runtime Environment, or JRE. In addition to the JRE, if a given application also requires the end user to compile Java source code (a rare condition) then the Java Software Development Kit, or Java SDK, is also required. However, Java applications have the advantage of being very portable. As long as a standard JRE exists on a given platform, the CVT framework can run on it. Also, many modern workstations include a JRE, or users may obtain one for free from a number of different sources, particularly directly from Sun Microsystems.

4.6.2 CCM Board Support

CVT uses the JHDL Simulator API to operate the CCM hardware. This gives features to CVT which are much like board support packages that are available for many CCM platforms. However, the Simulator API does not perform its actions directly with the CCM hardware. It still relies on an underlying board support package. JHDL includes interfaces that may be used to create classes which integrate with these drivers and packages, even if they are not written in Java. These interfaces include means of configuring the FPGA hardware and manipulating the hardware state through cycling the system clock. They also
include methods for obtaining runtime hardware state to back annotate the circuit design model.

### 4.6.3 Execution-time Hardware Design Modifications

The RCAF specification requires that the CCM application be able to modify the hardware configuration based on user-supplied data at execution time. In implementing this feature, CVT represents a compromise between two extremes. On one end is a low level, yet complex means of manipulating existing hardware configurations. On the other end is the use of massive EDA tools to synthesize and regenerate the hardware configuration from design inputs in standardized and widely-used formats.

The low-level, complex solution may be implemented with such systems as Xilinx’s JBits software development kit[15]. The JBits tool provides APIs which allow a Java program to perform very low-level manipulations to an existing configuration bitstream for the Virtex family of Xilinx FPGAs. Thus, optimizations based on user data, such as the type demonstrated in Section 1.4.3, could be performed on a bitstream that was produced just once. This bitstream is created by Xilinx-specific tools based on a hardware design netlist from the CAD tool. This bitstream could be generated during the deploy stage of the application development process and then included with the final application. The principle advantage of the JBits-style approach to execution-time modification of the hardware design is that it can be performed very quickly compared to other means of modification. JBits is also a relatively small tool with the ability to perform virtually any type of modification necessary.

The JBits approach has significant disadvantages. The JBits approach will only work with Xilinx Virtex FPGAs, thus limiting the set of CCMs for which it could be used. The JBits API also demands a very low-level understanding of the FPGA hardware being targeted. The modifications would therefore either be very complex to carry out or be constrained to not require very extensive modifications (e.g. limiting modifications to LUT configuration changes and excluding more complex modifications that would require routing changes).
A large, traditional EDA tool could also be used to modify an existing hardware design. With a full EDA tool, the hardware design entry may be in a variety of widely-used and familiar formats, such as VHDL, Verilog, and more. The execution-time design parameters may be included with these design entry components. The EDA tool then processes the design entry to create a circuit netlist. This netlist is then processed by technology-specific tools to perform placement and routing of the circuit components to generate low-level configurations for the FPGA devices in the CCM. This approach is flexible to meet the needs and preferences of the hardware designer. It also allows the designer to create a high-level, behavioral design of the hardware if desired.

The full EDA tool approach also has disadvantages. The very size of the tool would prevent it from being easily deployed along with a CCM application. Such tools often require hundreds of megabytes of storage space for the program code. Licenses for such tools are also typically very costly. Another problem is the requirement of the technology-specific tools to perform placement and routing of the design. These tools also require a significant amount of resources on the host system.

CVT offers a compromise between these two extremes by taking advantage of the features of JHDL. JHDL may be used to regenerate circuit netlists of re-parameterized designs. As with traditional EDA tools, this netlist must then be processed by placement and routing tools specific to the FPGAs being targeted. However, JHDL itself is a very small component. A sufficient subset of JHDL may only be as small as two to three megabytes in size. This approach still requires that the runtime environment include the placement and routing tools. However, this requirement is not entirely impractical. The size of these tools is not altogether too high. And they are sometimes available for free (such as Xilinx’s ISE WebPACK tool) or they might come bundled with a given CCM. Although such tools might not be deployed along with CVT-based application, it is still reasonable to expect the end user to have them available should a particular application require them. Although design entry formats for JHDL are not as varied as with many full-featured EDA tools, it still offers tremendous flexibility and breadth of usability due to its programmatic

---

1See Appendix B for a listing of various sizes. Minimum JHDL sizes include the size of the model listed in Table B.1 as well as at least one technology library listed in Table B.2
approach to circuit description using the Java programming language. CVT enhances this even more by establishing a full application framework that may be used to communicate user directives to the JHDL design.

Other means of runtime optimizations and hardware configuration modification also exist. Some of these include techniques such as design-level scan\cite{17} and direct modification of the configuration bitstream\cite{18}. The former introduces additional hardware overhead in the configuration but is relatively straightforward to implement and requires no additional tools. The latter method is similar to a JBits-style method, but may require less overhead and less knowledge of the internal FPGA architecture, as long as the tools to perform the modification abstract it sufficiently. However, this method is also quite closely tied to a specific FPGA architecture.

4.6.4 Deployment Considerations Conclusion

In considering deployed applications based on CVT, the additional overhead for Java code execution, CCM board support, and runtime hardware design modification are generally ignored. These additional resources are typically assumed to be readily available to the end user. In analyzing deployment issues related to CVT-based applications, such as in the quantitative analysis presented in Section 5.4, these additional tools are not explicitly discussed. However, the reader should note that they still play a role in the overall CVT system and that the deployment stage of the development process must include considerations based on these requirements.
Chapter 5

Example Applications

Several applications demonstrate the use of CVT as a reconfigurable computing application framework. A sample set of applications is presented in this chapter. These example applications demonstrate the wide variety of applications possible as well as the simplicity of building up CCM-based applications based on CVT. The first application, an audio processing application, demonstrates the usefulness of internal hardware visibility made possible by the JHDL application model. The next application, a string comparison application, demonstrates the advantages of an application that is dynamically reconfigurable to obtain optimal application performance. The next application, a simple game, demonstrates the use of the complete application communication architecture in CVT. An analysis of the size of these three example applications gives insight into the amount of development effort required to produce applications built on top of CVT. Finally, another two applications, a hybrid CPU/FPGA system debugger, and a hardware-synthesized code debugger, add more examples of the utility of the CVT system as an RCAF.

5.1 Audio Processing Application

The first example application performs custom processing on a two-channel audio stream. This application demonstrates a method that may be used to include a custom circuit debugging and/or application state view with the default CVT system. The approach used by this application does not extend any of the CVT classes themselves. Instead, it creates a custom circuit viewer which works independently of the default CVT viewers. For this particular application, such a setup works well because the continuous streams of input and output data do not require interactive control beyond the default controls offered
by CVT. Therefore the custom viewer is not fully integrated with the CVT communications architecture; it simply provides visual feedback to the application user.

Figure 5.1 shows the custom viewer for the audio processing application. This viewer is created in the TestBench which creates the hardware design for the audio processing circuit. The only connection between the custom viewer and the rest of the application is a reference to the JHDL HWSystem. Four panels make up the custom view. The two panels on the left show the history of raw audio samples as they came into the system, one panel for each channel of audio. The right two panels show a corresponding history of audio samples that exited the system.

![Custom viewer for an audio processing application.](image)

Each audio sample in this application is represented with twenty-bit buses inside the FPGA. The audio streams enter and exit the FPGA hardware serially, according to the specifications of a particular stereo CODEC (the Asahi Kasei stereo delta-sigma CODEC [AK4520A]). The serial interface reduces the number of pins required for communication...
between the FPGA and the CODEC. This savings in I/O bandwidth would normally mean that data visibility could only be obtained with the added complexity needed to de-serialize and de-multiplex the data. JHDL simplifies this data visibility problem by providing full internal visibility to the custom viewer. The data samples displayed in the custom viewer come from values read from the wires which are internal to the hardware design. The viewer obtains a reference to these wires using the findNamed method in Table \[ 3.1 \]. The get method in Table \[ 3.8 \] returns the wire values to the viewer. The viewer buffers these values to form the data sets from which the audio sample history panels derive the information displayed. Within an execution environment that supports hardware readback, this same sampling method works when the application is executed in hardware as well as in simulation.

### 5.2 Edit Distance Application

The next example application computes the edit distance between two strings. Edit distance is the number of character changes (insertions, deletions, and substitutions) required to transform one string (the source) into another string (the target). Edit distance calculations are useful in a number of different applications. One of the most common uses is the comparison of DNA (deoxyribonucleic acid), RNA (ribonucleic acid) or protein strings. The bases of the DNA, RNA, or protein strings are represented by characters. These characters are run through the edit distance calculator to determine how similar or dissimilar two strings are. Such applications help biological and pharmaceutical researchers determine such things as the effectiveness of new drugs. Because databases of information related to such work is growing at a rate faster than performance improvements in general-purpose computing, application-specific, CCM-base systems give these researchers a cost effective way to keep pace with the data[3].

The following four subsections describe this application according to the model-view-controller architecture described in Section \[ 4.1 \]. This shows how the various application components correspond with the architectural components of the RCAF. It also shows how the development of the application may be subdivided into easy-to-manage portions.
Thus a group of designers may easily develop various parts of a significant application concurrently.

5.2.1 Edit Distance Application Model

The edit distance application described here uses a four-character alphabet suitable for comparing DNA strings. The four characters are A, C, G, and T, representing the bases adenine, cytosine, guanine, and thymine, respectively. The calculator also allows for a wildcard character. Each character is represented by four bits. (Three bits would normally suffice for this application. However, many FPGAs are built with four-bit lookup tables (LUTs). Therefore, a four-bit character representation allows the alphabet to grow to nearly four times its size with only a little extra routing overhead.)

The model for this edit distance calculator application is a parameterizable JHDL design built up with a linear systolic array. Lipton and Lopresti introduced a generic ASIC systolic array implementation which accepted any two strings for comparison[19]. Later, Hoang demonstrated that embedding one of the strings into the hardware design of an FPGA can greatly improve the speed of the string comparisons[20]. The design for this example application is based on the designs of these three researchers. Another report from Bland and Megson demonstrates how dynamically reconfigurable systolic array processors in FPGA designs can dramatically improve performance of other similar applications[21].

Figure 5.2 shows a portion of the code for the EditDistance class which builds the circuit model. The parameters that determine the circuit structure and function are the width of the output wire distanceOut and the contents of the string targetStr. The output wire is connected to a counter which tallies the total number of operations needed to convert the source string to the target string. Therefore, the width of this wire determines the maximum edit distance value the circuit may detect. The user provides the desired target string parameter when starting the application. This means that the circuit will be optimized for the data at deployment time, even if it changes from one use of the application to the next. Although this specific example only demonstrates the RCAF system in simulation mode, the deployment-time optimizations for in-hardware execution could be performed by netlisting again or by modifying previously generated hardware.
configuration images. If the target string length and the counter width remain constant, then the hardware configuration image can simply be modified, saving the added time that would be required to perform placement and routing again. The method of updating optimizations between execution runs could be determined by extensions to CVT, making the whole process transparent to the user.

```java
public class EditDistance extends Logic {

    public static CellInterface[]
    cell_interface = {
        in("charIn",4),
        out("distanceOut",PARAM_WIDTH),
        param("outputWidth",INTEGER),
        param("targetString",STRING),
    };

    public EditDistance( Node parent, Wire charIn,
            Wire distanceOut, String targetStr ) {
        // constructor code ...
    }

} // end class EditDistance
```

Figure 5.2: The EditDistance class creates the circuit model for the edit distance application.

The JHDL model for this application also has additional convenience methods to support both the creation of the circuit structure and the interpretation of the circuit state. These methods are found in the EDAlphabet class as well as the EditCellInput, EditCellInner, and EditCellOutput classes which make up the processors of the systolic array. The EDAlphabet class contains methods to convert between ASCII characters and binary representations of characters. The processor classes provide methods that indicate their state; the custom viewer described in Section 5.2.2 uses these methods to conveniently obtain information to output.
5.2.2 Edit Distance Application View

The view for the edit distance application (Figure 5.3) provides a simple interface that allows the user to input a source string, control the execution of the circuit, and view the progress of the string as it cycles through the processors of the systolic array. The inputs are in the form of text fields. The circuit visualization shows information about each processor in the circuit. On the top of the view of each processor is displayed the character which is embedded in that processor. Below that is a symbol which indicates whether the current source character in that processor is equal or not. Below that is the character from the source string. Finally, on the bottom is the output value from the processor, which is passed down the array to the accumulator connected to the circuit output.

![Graphical User Interface](image)

Figure 5.3: The graphical user interface for an edit distance application.

Between cycles of the system clock, the viewer updates the values displayed. The values are obtained by using methods shown in Tables 3.1 and 3.8. References to the circuit wires that contain the values of interest are obtained with the `findNamed` method. The actual values are then obtained through the `get` method.
Each of the processor views in the edit distance application is a separate, parameterizable GUI panel. This GUI subpanel was developed along with the hardware design for an individual processor. This custom view of a single processor was used to help debug the hardware design. When the processor design was verified, both the processor and GUI subpanel were duplicated in parameterizable arrays in the hardware design and complete custom viewer respectively. Thus the development of the custom viewer not only built up the final application interface, it also aided in the hardware design and debug from the beginning of application development.

5.2.3 Edit Distance Application Controller

The controller for the edit distance application is built by extending the features of the default CVT Broker class. Additional, custom commands are also added to the CVT command interpreter. The code for the custom Broker class is shown in Figure 5.4. A new custom edit distance viewer is created in the EDBroker constructor on line 5. Line 11 creates and registers an instance of the EDCLICommand class. This class implements a command for the CVT command interpreter (the CLIInterpreter class). This command is tied to the text input field in the custom viewer in Figure 5.3. Entering a new source string in the text field causes the custom command to be generated and sent to the command interpreter, and hence the Broker. This results in an updated data schedule for the source string input values that are streamed into the system. This example of custom interactive circuit input and control shows how the deployed application can be built in such a way that is most intuitive to the final user, not just the hardware designer.

5.2.4 Completing the Edit Distance Application

The edit distance application is executed by a JHDL testbench class. This class is the principle execution point of the application; in this class, the system components (the circuit model, the viewers, and the controller) are built and finally integrated into the complete CCM application. The code for this class is shown in Figure 5.5. It contains a main method which receives user data on lines 13 and 14. That information is then used to create the testbench, and hence the circuit design itself, on line 16. This is the point
public class EDBroker extends Broker {

    public EDBroker(HWSystem system, CLInterpreter interp) {
        super(system, interp);
        new EDFrame(system, interp);
    }

    protected void registerCLICommands(HWSystem system, CLInterpreter interp) {
        super.registerCLICommands(system, interp);
        EDCLICommand edc = new EDCLICommand(interp);
    }

} // end class EDBroker

Figure 5.4: The EDBroker class extends the default RCAF controller for the edit distance application.

at which the RCAF dynamic circuit generation features are activated to create the optimal circuit configuration for the user. Finally, the application creates an EDBroker on line 18 to start up the interactive components (i.e. the custom viewer and the controller elements).

This example application shows the great potential of dynamic, in-field reconfiguration for CCM-based applications. The custom application interface provides a means for end users to easily configure the system and manipulate values sent to it. The users don’t need to have knowledge of the specialized APIs of the underlying RCAF. Instead, an application-specific, user-friendly interface facilitates user interactions with the runtime-custom hardware application accelerator. The CVT RCAF allows the application developers to easily combine the dynamic capabilities of the CCM hardware with software interface solutions that match the problem to be solved from the user’s point of view.

5.3 Guessing Game Application

The next example application demonstrates a system that follows the more involved communications architecture of the CVT model, view, and controller. The application is an implementation of a guessing game with similarities to the game Mastermind®. The purpose of the game is to guess the colors and order of four “pegs” which make up the “code” to break. The player has a set number of guesses to break the code, and after each
Figure 5.5: The tbEditDistance class creates and integrates the various application components.

guess receives feedback that helps to plan the next guess. As with the previous example, the application components associated with the model, view, and controller are discussed separately in the following sections.

5.3.1 Guessing Game Application Model

The guessing game application model is a JHDL design made up of an array of cells which represent each guess from the player. For simplicity, the hardware design is static, although it could easily be parameterizable to create a dynamic design. (e.g. the number of elements in the code, the maximum number of guesses, and the set of possible colors could all be parameterized.)

The JHDL circuit has four input buses plus a control input which the application uses to register each guess from the player. Another set of signals allow the application to query the game state. Thus all of the circuit input and output is performed on the external port level, allowing sufficient circuit visibility even if the readback capability is not available.
5.3.2 Guessing Game Application View

Figure 5.6 shows the view for the guessing game application. It is composed of a custom panel, representing the game board. The player uses a pointer to drag colored squares to appropriate positions on the board and then clicks on a submit button to register each guess. This triggers an event that includes information about the guess submitted. Other classes may receive these events by implementing the ColorPlacementListener interface and then registering themselves with the game view (the GameCanvas class). The game viewer is independent of any given game model or controller. It is simply an interface component that may be included with any application that needs it. It may also easily be exchanged for alternate application interface, thus helping to keep the game application modular.

![Figure 5.6: The graphical user interface for a guessing game application.](image-url)
Table 5.1: ColorPlacementListener method for obtaining guess events from the guessing game viewer

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>colorsSubmitted</td>
<td>GameCanvas source, int row, int[] guess</td>
</tr>
</tbody>
</table>

5.3.3 Guessing Game Application Controller

The controller for the guessing game application is also built by extending the features of the default CVT Broker class. However, this Broker extension involves more of the features of the CVT communications architecture as described in Section 4.1. That is, the controller includes not only the Broker extension, but also a viewer listener (an implementation of the ColorPlacementListener interface) and three custom textual commands for the CVT command interpreter.

The three textual commands are used to tell the controller to start a new game, to register a guess from the player, or to query the circuit state and update the game view with appropriate feedback for the player. Each of these custom commands simply maps to a corresponding method of the custom Broker class. Those methods perform the tasks of manipulating circuit inputs and game view parameters appropriately. Using these textual commands to manage the flow of major system tasks helps to create a manageable history of commands for later execution.

The textual commands also create a convenient point in which to place a bridge between graphical user interface input and the application controller. An implementation of the ColorPlacementListener interface performs this bridging role. This class receives information from the application viewer about the player’s guess. It translates this information into one or more of the text commands described above. The command interpreter then logs those commands in its history and performs the appropriate method calls in the Broker class.

Figure 5.7 shows the portion of the custom Broker class which creates the various components described above and establishes the relationships between them. Lines 14, 15, and 16 create the three custom textual command objects and register them with the CVT
command interpreter. Line 20 creates the application view. Then line 21 creates a listener to receive events from the application view. This is the listener which performs the translation of GUI events to text commands. Thus, in just a few lines of a single method, the main communications system of the application is established. This demonstrates the ease with which a developer may integrate various components to build up significant applications using the CVT RCAF.

```java
public class PGBroker extends Broker {

    public PGBroker( tbPG parent, HWSystem system ) {
        super(system, new CLInterpreter());
        this.parent = parent;
        openGame(); // causes the game view to display
    }

    protected void registerCLICommands( HWSystem system, CLInterpreter interp ) {
        super.registerCLICommands(system, interp);
        // registers put and get commands:
        Stimulator.registerCLICommands(interp);
        new CommandNewGame(this, interp);
        new CommandGuess(this, interp);
        new CommandFeedback(this, interp);
        // remember the interpreter
        this.interp = interp;
        // create game view in this method for convenience only
        canvas = new GameCanvas();
        canvas.addListener(new ViewListener(interp));
    }

    /* ... remaining PGBroker methods and fields ... */
}
```

Figure 5.7: The EDBroker class extends the default RCAF controller for the edit distance application.

5.4 Quantitative Analysis

This section discusses metrics for determining the value of an RCAF. The bulk of the functionality of an RCAF-based application is intended to be in the RCAF itself.
Therefore, it is natural that the application-specific code layered on top of the RCAF should be relatively small. Hence one useful metric to evaluate the usefulness of the RCAF is the code size of the non-RCAF code. The smaller the size of the non-RCAF application code, the better. That is because the small code size indicates that the RCAF establishes an API and application framework that is not overly complex to use. The less code the RCAF requires the application developer to write, the less likely the application will have errors and the more likely it is to be easily-maintained. The size of the RCAF itself is also important, because it will be deployed along with the application.

This section lists the various sizes of the three sample applications described above. Sizes are given in both line counts and compiled code size. These measurements don’t give any real indication about the size of any RCAF-based application. However, they may offer information to help develop a “feel” for the amount of effort that may be necessary to develop a full CCM application based on the CVT RCAF. Therefore, true analysis of this data is a combination of both objective and subjective observations. The raw numbers must be weighed against the capabilities and limitations of the corresponding applications.

EDIF (electronic design interchange format) netlist sizes are also given to help gain an understanding for how much work the JHDL component of CVT does to support the application. The EDIF netlist size correlates directly with the number of data structures that JHDL must maintain during the simulation or in-hardware execution of the hardware design. Naturally, this also corresponds to the amount of physical FPGA hardware resources required to host the design. Even very simple designs can generate very large netlists. For example, the edit distance application may be scaled to very large sizes by embedding more target characters, and hence more processors, in the hardware design. CVT manages the scaling of such designs for the application, thus removing a great deal of the burden for application support from the developer’s application design while still maintaining the full capabilities of the acceleration hardware provided by the CCM.

The sizes for the three applications are given in Table 5.2. The size of the CVT is from two to four megabytes in size, depending on the features included. Note that this leaves out the sizes of other runtime components which may be required for the application,

---
1See Appendix B for a listing of CVT component sizes.
such as a Java virtual machine and CCM-specific hardware drivers. Such components are assumed to be available on a system which includes a given target CCM, and are therefore not part of the CCM application distribution.) The size of the application code layered on top of CVT is relatively small—as much as two orders of magnitude smaller than CVT itself. Given the features that could be included with the application—dynamic circuit model generation, netlisting, platform control, etc.—the result is a small (relative to most other full-featured CAD tools) simple, deployable application. The small sizes listed in Table 5.2 also imply that the development effort for a given application is relatively small. This adds to other, less quantitative benefits of CVT, such as rapid, in-hardware debugging, reusable code, and a solid application architecture framework.

Table 5.2: Sizes of sample CVT-based applications.

<table>
<thead>
<tr>
<th>Application</th>
<th>Audio Processing</th>
<th>Edit Distance</th>
<th>Guessing Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Lines Model</td>
<td>2,157</td>
<td>1,315</td>
<td>2,523</td>
</tr>
<tr>
<td>Model</td>
<td>1,570</td>
<td>696</td>
<td>1,588</td>
</tr>
<tr>
<td>View</td>
<td>587</td>
<td>503</td>
<td>614</td>
</tr>
<tr>
<td>Controller</td>
<td>n/a</td>
<td>116</td>
<td>321</td>
</tr>
<tr>
<td>Compiled Size Model</td>
<td>48,161 bytes</td>
<td>37,960 bytes</td>
<td>65,192 bytes</td>
</tr>
<tr>
<td>View</td>
<td>15,811 bytes</td>
<td>18,627 bytes</td>
<td>38,825 bytes</td>
</tr>
<tr>
<td>Controller</td>
<td>n/a</td>
<td>15,695 bytes</td>
<td>16,278 bytes</td>
</tr>
<tr>
<td></td>
<td>11,089 bytes</td>
<td>3,638 bytes</td>
<td>10,089 bytes</td>
</tr>
<tr>
<td>EDIF Size (ZIP Compressed)</td>
<td>27,139 bytes (43.6% deflated)</td>
<td>21,852 bytes (42.4% deflated)</td>
<td>35,102 bytes (46.2% deflated)</td>
</tr>
</tbody>
</table>

5.5 Additional Examples

Two other example applications demonstrate the utility of the CVT RCAF architecture. Some of the user interface elements of the first application are shown in Figure 5.8.
Eric Roesler of the Configurable Computing Laboratory at Brigham Young University created this application using CVT[22]. The application is used for debugging hybrid FPGA systems that include a CPU (central processing unit) core in the FPGA configuration. This is useful not only for debugging new CPU design cores for FPGAs, but also for verifying the functionality of the CPU core and the software executing it as they work alongside other custom logic included in the same FPGA configuration. This application is unique because of its multiple simultaneous views of the same circuit model. Each view offers completely different types of insights into the execution state of the hardware. Figure 5.8 depicts a view of the major components of the CPU (arithmetic logic unit, instruction register, the register file, etc.) and a view that shows a view of the original source code of the software being executed in the CPU core. Another custom viewer not shown here is a view of the assembly code version of the software. These views may be used simultaneously with traditional schematic and waveform views to help designers verify the operation of the hardware and software in the FPGA. The model-view-controller architecture used by CVT helps to keep the various views separated from the model to enable each of them to display independent and unique versions of the hardware state.

Figure 5.9 shows the interface to another example application. This application is used to debug high-level software code that is synthesized to structural hardware components[23]. As in the previous example, this example gives multiple independent views of the hardware system. This example also uses a more complex controller system to manage not only the JHDL model, but also the synthesis and debug system from which the JHDL model is derived. This system divides the controller responsibilities between two classes. The first class is the same broker class in the CVT system; this class basically provides the same functionality of the default CVT system. The other class manages the hardware synthesis and source code views for the debugging application. This separated controller keeps the CVT system, including the JHDL circuit model, independent of the synthesis and debug environment. This allows either system to be used just as well with any other system of equal or comparable functionality. The result is a highly modular and easily-maintained application.
Figure 5.8: Hybrid CPU/FPGA system debugger.
Figure 5.9: Debugger for hardware synthesized from high-level code.
Chapter 6

Conclusions and Future Work

Existing computer-aided design (CAD) and electronic design automation (EDA) tools for hardware designs targeting FPGAs are inadequate to support the development needs for configurable computing machine-based applications. While well-suited for ASIC design and some static FPGA design, these tools lack the support for dynamic reconfiguration and in-hardware debugging that are among the greatest advantages offered by configurable computing machines (CCMs). Also, because they focus only on the hardware design, these tools compel applications designers to divide the development process into separate hardware and software development processes. These two processes typically only meet in the deployment stage of the application development. Combining these development processes into unified design, debug, and deploy stages would allow for greater design exploration, better integration between hardware and software, improved custom debugging capabilities and possibly better, more intuitive application user interfaces.

The conclusion of this research is that developers may use a reconfigurable computing application framework (RCAF) for creating CCM-based applications to overcome the limitations of traditional, monolithic CAD/EDA tools. An RCAF offers a unique interface layer between the CCM application hardware and software components. This layer does not merely control the configuration and execution of the CCM hardware. It also maintains a hardware model that gives the application extraordinary visibility into the hardware state based on the original design entry model. The RCAF also establishes an architecture model on which significant applications may be built. This allows for excellent hardware and software integration which gives the user unprecedented access to and control over
the hardware. CCM applications may also use RCAF features to provide dynamically-configurable hardware designs which may improve in-field performance by optimizing the hardware design based on user input.

6.1 Contributions of this Research

The main contributions of this research are the specification and implementation of a reconfigurable computing application framework. The design and implementation of this RCAF (called CVT) are built on top of existing technology and prior research. This prior research provided the following principles for CCM application development:

- Hardware designs for FPGAs can be built up in simply and programmatically using general purpose, object-oriented programming languages[9, 10].

- The software simulation and hardware execution of a hardware design can and should be performed from within the same design environment for debugging[6, 11, 5].

- Circuit design and debug visualization can be done using custom viewers and circuit control mechanisms that best reflect the usage and requirements of each individual application.

- FPGA hardware designs may easily be modified to include optimizations based on data provided at execution time[7, 8].

This solid foundation helped me create a system with the following features:

- CVT creates a design and debug approach for CCM application development that integrates the development of the hardware and software components of CCM-based applications. This gives the designer more freedom to explore whole-application design alternatives. This integrated development also shortens the application development time by preparing for final application deployment throughout the design and debug stages of development.

- The CVT application model (as in the model-view-controller architecture) is consistent with the original design-entry format. The model state is consistent with the
corresponding state of the actual CCM hardware. This gives the applications complete visibility into the hardware design structure and state.

- CVT allows for custom hardware interaction and control. This control may be based on the specifications of a given application rather than of the RCAF or the target CCM. The application interface is built on top of the RCAF architecture, which takes care of the low-level details of interacting with the CCM-specific interfaces. The same application-specific interfaces may be used for both simulation and in-hardware execution of the hardware design component of the application.

- Final CCM applications built on CVT may include dynamically-configured hardware designs. This allows the applications to incorporate user-provided data to improve the design performance.

- CCM applications based on CVT are simple to develop, relative to the complexity of building up an entire application that may take advantage of the features of CCMs. Resulting applications have a small, deployable form.

Individually, the above features do not necessarily represent significant new contributions. However, the RCAF specification and the CVT implementation integrate these features in such a way that does present a new and significant contribution to the CAD/EDA community. This integration is built on top of sound architectural principles that help to create a system that is modular, scalable, and deployable. This system has already been used to develop significant applications. The CVT framework allows these applications to take advantage of CCM features that would otherwise only be available with a significant increase in overall application development cost, particularly in time.

The pre-existing technology on which CVT is based consists of the JHDL system. JHDL performs the role of the application model within the CVT system. This subsystem performs a number of critical functions related to the RCAF specification given in Chapter 2. These include interacting with and maintaining an image of the state of the circuit within actual CCM hardware. These functions also include the ability to programmatically and dynamically build up a hardware configuration. CVT works with JHDL through
the circuit structure, circuit state, and simulator APIs which JHDL makes available. CVT creates a controller and viewer system to work with these JHDL APIs. It also establishes the communications system to be used by these systems. This is in accordance with the commonly used model-view-controller architecture for user interface design in software systems.

The CVT system based on JHDL is not only an improvement over the previous system to use JHDL (i.e. the JAB system described in Section 3.3). It also enables an approach to CCM application development that allows the design and debug stages of development to integrate the development of both the hardware and software of the CCM application. This approach also permits better application design and debug by giving developers the ability to easily generate custom development environments that match the needs of each application. The new and unique feature of this development process which CVT provides is that the features which enhance the design and debug stages of development may persist in the application in the deploy stage and beyond. The final, deployed application will continue to be flexible and portable because it inherits the same characteristics of the CVT RCAF itself, including the characteristics listed in the bulleted items above. The CVT system is an improvement over traditional CAD/EDA tools for CCM application development in that it allows applications to retain their dynamic characteristics, which, in a normal application development process, are typically discarded after the debug stage.

6.2 Future Work

Future work on the CVT RCAF system may include additional features, improvements to existing capabilities, and continued architecture exploration. This section lists an example for each of these categories.

The circuit model for CVT applications based on JHDL designs is a structural circuit model. While the circuit description may contain behaviorally modeled cells, such components are typically only for simulation purposes. However, behavioral descriptions of circuits can help shorten development time by eliminating the need to translate algorithms from their original programmatic descriptions to structural circuit models. CVT could be improved by including features that support such higher-level circuit descriptions.
Work being carried out by Tripp[24] and Hemmert[23] related to synthesis of high-level behavioral code to structural designs and corresponding debugging support may be used to create a system that better integrates behavioral synthesis. Hemmert’s system does already include CVT as a foundation for its application architecture, however, it carries out all interactions with Tripp’s synthesis tools independently of CVT. Integrated support could include a specialized API within CVT to facilitate the inclusion of synthesized circuit components within the CVT application model.

Current hardware support in CVT is dependent on individual CCM board models. These board models are typically built on foundation APIs provided by JHDL. However, board model creators have a lot of design freedom. Therefore, each board model may differ significantly from the next. This means that application views and controls typically must conform to the APIs of a specific board model. CVT hardware support may be improved by providing platform-independent APIs to support all board models adhering to the APIs. These APIs may be created by extending the current board model framework in JHDL. Existing board models may be modified to fit this API. With a unified API for hardware control, application developers need only to concentrate on the one set of APIs provided by CVT, and such applications will work for any specific CCM board which has a JHDL/CVT model.

The architecture specified and implemented for the CVT system is merely one way to create an RCAF. Further architecture experimentation and exploration may continue to produce systems with different characteristics and features. Although the current implementation is built on solid and well-known principles, alternate implementations may prove to have qualities which are better-suited to particular classes of applications. For example, CVT may be augmented to include a suite of varying controllers to fulfill the needs of different types of applications: one may be well-suited for traditional digital logic design, another may be useful for image-processing applications, another for pattern matching applications, etc. Developers may select the framework components which are the best fit to the problem they are solving. Such architectural experimentation will likely be driven by application development needs as they arise.
Appendix
Appendix A

CVT Component APIs

This appendix contains a listing of many application programming interfaces (APIs) of various CVT components. Many of these components may be used within other applications as noted in Section 4.3. This is not a complete list of APIs, as can be found in the documentation on the JHDL web site [25]. In particular, the vast number of APIs associated with the JHDL tool are excluded from this listing. Many JHDL APIs were already listed in Section 3.1.2. This appendix focuses on the APIs for the controller and viewer components of the default CVT system. The purpose of this appendix is to give the reader a view of the capabilities of CVT as well as the requirements for extending CVT to create custom applications.

A.1 CVT Controller APIs

The following classes are components of the CVT controller. As described in Section 4.1, the controller is the application component that receives user input and performs corresponding manipulations to the model.

- Broker
- CLICommand
- CLInterpreter
- SimulatorThread
- Stimulator
The Broker Class

The Broker class is the main component of the CVT controller. This class receives messages through the CLInterpreter to determine actions to perform on the CVT model (the JHDL circuit design). The most important methods below are the constructor and the registerCLICommands methods. All other methods can also be overridden and supplemented with methods to suit the needs of a given CCM application.

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broker</td>
<td>Broker (constructor)</td>
<td>HWSSystem, CLInter-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>preter</td>
</tr>
<tr>
<td>CLInterpreter</td>
<td>getInterp</td>
<td>void</td>
</tr>
<tr>
<td></td>
<td>registerCustomViewer</td>
<td>Window</td>
</tr>
<tr>
<td></td>
<td>cycleSimulator</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>stepSimulator</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>resetSimulator</td>
<td>boolean</td>
</tr>
<tr>
<td></td>
<td>newSchematicView</td>
<td>Cell</td>
</tr>
<tr>
<td></td>
<td>print</td>
<td>Cell</td>
</tr>
<tr>
<td>boolean</td>
<td>addWavesWatch</td>
<td>Wire</td>
</tr>
<tr>
<td></td>
<td>removeWavesWatch</td>
<td>Wire</td>
</tr>
<tr>
<td></td>
<td>closeViews</td>
<td></td>
</tr>
</tbody>
</table>

The CLInterpreter Class

The CLInterpreter class receives text commands from the user or from a class that acts listens to events from interactive components and translates them to text commands. These commands are interpreted and corresponding methods of the Broker class are invoked. Any custom CLICommand may be registered with the CLInterpreter.
The CLICommand Interface

Implementations of the CLICommand interface may be registered with the `CLInterpreter`. These commands usually correspond to methods found in the CVT controller. The CLIException is thrown by the execute method of instances of CLICommand when there is a problem with the command given to the `CLInterpreter`. 

![Table](https://via.placeholder.com/150)

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>execute</td>
<td>CLInterpreter, String[]</td>
</tr>
<tr>
<td>String</td>
<td>getHelpText</td>
<td>String</td>
</tr>
<tr>
<td>String</td>
<td>getHelpType</td>
<td>String</td>
</tr>
<tr>
<td>String</td>
<td>getUsageText</td>
<td>String</td>
</tr>
</tbody>
</table>
The SimulatorThread Class

Many manipulations to the CVT model, such as executing a large number of cycles of the hardware clock, take a great deal of time, especially in simulation mode. When such operations take place in an interactive environment, they normally take over the main thread of the interactive components. This may make the application appear to hang or crash. The SimulatorThread is used to conveniently create a queue of threads separate from the main interactive thread to allow these events to execute without disrupting the rest of the application. The SimulatorThreadListener interface may be used to tell other application components when a particular SimulatorThread has completed.

<table>
<thead>
<tr>
<th>byucc.jhdl.apps.util.SimulatorThread</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Return Type</strong></td>
</tr>
<tr>
<td>SimulatorThread</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td>void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>byucc.jhdl.apps.util.SimulatorThreadListener</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Return Type</strong></td>
</tr>
<tr>
<td>void</td>
</tr>
</tbody>
</table>

The Stimulator Class

The Stimulator class manipulates values on input wires to a JHDL circuit. It partners with the Broker to perform manipulations to the JHDL circuit model. A set of Stimulator-specific commands are registered with the registerCLICommands method. These commands may be used from other custom commands. Testbenches should create Stimulators and register wires with them as needed.
<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stimulator</td>
<td>Stimulator (constructor)</td>
<td>Node</td>
</tr>
<tr>
<td>void</td>
<td>addWire</td>
<td>Wire</td>
</tr>
<tr>
<td>void</td>
<td>addWire</td>
<td>Wire, String</td>
</tr>
<tr>
<td>void</td>
<td>put</td>
<td>String, String</td>
</tr>
<tr>
<td>void</td>
<td>registerCLICommands</td>
<td>CLInterpreter</td>
</tr>
<tr>
<td>void</td>
<td>removeWire</td>
<td>String</td>
</tr>
</tbody>
</table>

A.2 CVT View APIs

The following classes are components of the default CVT view. As described in Section 4.1, the view is the application component that gives the user a view of the state of the application model.

- MemoryViewer
- SchematicViewerPanel
- SimControlPanel
- TreeBrowserPanel
- ViewerFrame
- WavesWirePanel
- WiresTablePanel
- cvtFrame

The MemoryViewer Class

The MemoryViewer displays a table containing the data stored in a memory device. The MemoryViewerFrame is a subclass of the ViewerFrame class that contains only a MemoryViewer.
The SchematicViewerPanel Class

The SchematicViewerPanel displays a schematic view of a cell at a given level of hierarchy in a JHDL circuit model. This view shows the cell ports, wires, and subcells. The ports and wires are annotated with values from the simulator. Other classes may use the SchematicActionListener and SchematicCanvasListener interfaces to obtain information about user interactions with the SchematicViewerPanel. The SchematicViewerFrame is a subclass of the ViewerFrame class that contains only a SchematicViewerPanel.
### byucc.jhdl.apps.Viewers.Schematic.SchematicActionEvent

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SchematicActionEvent</td>
<td>SchematicActionEvent (constructor)</td>
<td>Object, int, String, Wire</td>
</tr>
<tr>
<td>SchematicActionEvent</td>
<td>SchematicActionEvent (constructor)</td>
<td>Object, int, String, Cell</td>
</tr>
<tr>
<td>SchematicActionEvent</td>
<td>SchematicActionEvent</td>
<td>Object, int, String</td>
</tr>
<tr>
<td>Wire</td>
<td>getWire</td>
<td></td>
</tr>
<tr>
<td>Cell</td>
<td>getCell</td>
<td></td>
</tr>
</tbody>
</table>

### byucc.jhdl.apps.Viewers.Schematic.SchematicActionListener

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>schematicActionPerformed</td>
<td>SchematicActionEvent</td>
</tr>
</tbody>
</table>

### byucc.jhdl.apps.Viewers.Schematic.SchematicCanvasEvent

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SchematicCanvasEvent</td>
<td>SchematicCanvasEvent</td>
<td>Object, int, String</td>
</tr>
<tr>
<td>SchematicCanvasEvent</td>
<td>SchematicCanvasEvent</td>
<td>Object, int, String, String</td>
</tr>
<tr>
<td>SchematicCanvasEvent</td>
<td>SchematicCanvasEvent</td>
<td>Object, int, String, double</td>
</tr>
<tr>
<td>SchematicCanvasEvent</td>
<td>SchematicCanvasEvent</td>
<td>Object, int, String, int</td>
</tr>
<tr>
<td>String</td>
<td>getStringValue</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>getDoubleValue</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getIntegerValue</td>
<td></td>
</tr>
</tbody>
</table>

### byucc.jhdl.apps.Viewers.Schematic.SchematicCanvasListener

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>schematicCanvasEventPerformed</td>
<td>SchematicCanvasEvent</td>
</tr>
</tbody>
</table>
The SimControlPanel Class

The SimControlPanel provides in interactive interface directly tied to the CLInterpreter class. This panel includes buttons and textfields that correspond directly to specific text commands for the CLInterpreter. It also has a general-purpose text field for entering in arbitrary text commands. Other classes may use the SimControlActionListener interface to obtain information about user interactions with the SimControlPanel.
The TreeBrowserPanel Class

The TreeBrowserPanel displays a hierarchical view of a circuit design. The view format is a tree similar to tree views used in many file system viewers. Other classes may use the TreeBrowserActionListener interface to obtain information about user interactions with the TreeBrowserPanel. The TreeBrowserFrame is a subclass of the ViewerFrame class that contains only a TreeBrowserPanel.

### byucc.jhdl.apps.Viewers.TreeBrowser.TreeBrowserPanel

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeBrowserPanel</td>
<td>TreeBrowserPanel (constructor)</td>
<td>Cell</td>
</tr>
<tr>
<td>void</td>
<td>addTreeBrowserActionListener</td>
<td>TreeBrowserActionListener</td>
</tr>
<tr>
<td>void</td>
<td>removeTreeBrowserActionListener</td>
<td>TreeBrowserActionListener</td>
</tr>
<tr>
<td>Cell</td>
<td>getSelectedCell</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>loadClass</td>
<td>Cell</td>
</tr>
</tbody>
</table>

### byucc.jhdl.apps.Viewers.TreeBrowser.TreeBrowserActionEvent

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeBrowserActionEvent</td>
<td>TreeBrowserActionEvent</td>
<td>Object, int, String, Cell</td>
</tr>
<tr>
<td>TreeBrowserActionEvent</td>
<td>TreeBrowserActionEvent</td>
<td>Object, int, String</td>
</tr>
<tr>
<td>Cell</td>
<td>getCell</td>
<td></td>
</tr>
</tbody>
</table>

### byucc.jhdl.apps.Viewers.TreeBrowser.TreeBrowserActionListener

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>treeBrowserActionPerformed</td>
<td>TreeBrowserActionEvent</td>
</tr>
</tbody>
</table>
## The ViewerFrame Class

The ViewerFrame class provides a standard set of functionality for viewer frames used in CVT. It provides utilities for creating menus, building subpanels, and closing views.

### byucc.jhdl.apps.Viewers.ViewerFrame

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ViewerFrame</td>
<td>ViewerFrame</td>
<td></td>
</tr>
<tr>
<td>ViewerFrame</td>
<td>ViewerFrame</td>
<td>String</td>
</tr>
<tr>
<td>boolean</td>
<td>setSystemExitOnAll-ViewerFramesClosed</td>
<td>boolean</td>
</tr>
<tr>
<td>void</td>
<td>dispose</td>
<td></td>
</tr>
<tr>
<td>Container</td>
<td>buildContentPanel</td>
<td></td>
</tr>
<tr>
<td>JMenuBar</td>
<td>buildMenuBar</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>buildAndShowFrame</td>
<td></td>
</tr>
</tbody>
</table>

## The WavesWirePanel Class

The WavesWirePanel displays a set waveforms of wires. Interactions with the WavesWirePanel typically go through the WavesWireManager class. The WiresTableFrame is a subclass of the ViewerFrame class that contains only a WavesWirePanel.
The WiresTablePanel Class

The WiresTablePanel simply displays the set of wires owned by a particular cell in a circuit model of the CVT system. This view shows the name, type, and state of each wire. Other classes may use the WiresTableActionListener interface to obtain information about user interactions with the WiresTablePanel. The WiresTableFrame is a subclass of the [ViewerFrame](#) class that contains only a WiresTablePanel.
The cvtFrame Class

The cvtFrame class is the principal viewer for the CVT system. This viewer contains a hierarchical view of the circuit model and a table of wires from the currently selected circuit cell. This view also contains a panel that is used for user input directly to the CLIInterpreter.
<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvtFrame</td>
<td>cvtFrame (constructor)</td>
<td>Cell</td>
</tr>
<tr>
<td>cvtFrame</td>
<td>cvtFrame</td>
<td>Cell, CLInterpreter</td>
</tr>
<tr>
<td>cvtFrame</td>
<td>cvtFrame</td>
<td>Cell, Broker</td>
</tr>
<tr>
<td>cvtFrame</td>
<td>cvtFrame</td>
<td>Cell, CLInterpreter, Broker</td>
</tr>
<tr>
<td>CLInterpreter</td>
<td>getInterp</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>newCell</td>
<td>Cell</td>
</tr>
</tbody>
</table>
Appendix B

CVT Component Sizes

This appendix includes the sizes of the various components of the CVT RCAF system. This reference is useful to estimating the size of the RCAF to be deployed with a given application according to the features present.

Table B.1 contains sizes of the CVT system alone. The model subcomponent of CVT is basically just the JHDL tool for describing and simulating digital logic designs. If at least one of the technology-specific libraries from Table B.2 is included, the CVT set of tools is sufficient for simulation purposes only. The “hardware” software listed in Table B.3 is the component of the CVT system that is required for interfacing the JHDL simulator with CCM specific drivers to control hardware execution and obtain readback information. The “modules” software listed in Table B.3 is an extra set of parameterizable design modules that may be helpful in building up hardware designs. This optional toolkit includes such things as multipliers, finite state machines, floating-point tools, etc.

In many of the tables that follow, groups of tools are also subdivided into separate components. Many of these subcomponents are not not mutually exclusive, therefore, summing the sizes of subcomponents separately will often result in a value larger than the whole group combined.

Table B.1: Sizes of the CVT RCAF and its subcomponents

<table>
<thead>
<tr>
<th>Component</th>
<th># Lines</th>
<th>Source Size</th>
<th>Compressed</th>
<th>Compiled Size</th>
<th>Compressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>52,810</td>
<td>2,408 KB</td>
<td>373 KB</td>
<td>1,448 KB</td>
<td>405 KB</td>
</tr>
<tr>
<td>View</td>
<td>31,402</td>
<td>1,352 KB</td>
<td>286 KB</td>
<td>1,700 KB</td>
<td>487 KB</td>
</tr>
<tr>
<td>Controller</td>
<td>11,007</td>
<td>592 KB</td>
<td>118 KB</td>
<td>588 KB</td>
<td>160 KB</td>
</tr>
<tr>
<td>CVT Total</td>
<td>93,519</td>
<td>4,264 KB</td>
<td>761 KB</td>
<td>3,692 KB</td>
<td>1,040 KB</td>
</tr>
</tbody>
</table>
Table B.2: Sizes of various technology-specific libraries

<table>
<thead>
<tr>
<th>Library</th>
<th># Lines</th>
<th>Source Size</th>
<th>Compressed</th>
<th>Compiled Size</th>
<th>Compressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtex</td>
<td>136,183</td>
<td>6,860 KB</td>
<td>1,238 KB</td>
<td>5,100 KB</td>
<td>1,809 KB</td>
</tr>
<tr>
<td>Virtex2</td>
<td>149,664</td>
<td>7,608 KB</td>
<td>1,316 KB</td>
<td>5,472 KB</td>
<td>1,940 KB</td>
</tr>
<tr>
<td>XC4000</td>
<td>92,923</td>
<td>4,764 KB</td>
<td>863 KB</td>
<td>3,656 KB</td>
<td>1,261 KB</td>
</tr>
<tr>
<td>XC9000</td>
<td>84,497</td>
<td>4,972 KB</td>
<td>1,062 KB</td>
<td>3,872 KB</td>
<td>1,527 KB</td>
</tr>
<tr>
<td>Total Xilinx</td>
<td>451,588</td>
<td>23,496 KB</td>
<td>4,349 KB</td>
<td>17,440 KB</td>
<td>6,357 KB</td>
</tr>
<tr>
<td>CSRC</td>
<td>31,128</td>
<td>1,736 KB</td>
<td>296 KB</td>
<td>1,340 KB</td>
<td>474 KB</td>
</tr>
<tr>
<td>Teramac</td>
<td>31,719</td>
<td>2,164 KB</td>
<td>549 KB</td>
<td>780 KB</td>
<td>777 KB</td>
</tr>
<tr>
<td>Grand Total</td>
<td>514,435</td>
<td>27,396 KB</td>
<td>5,192 KB</td>
<td>20,532 KB</td>
<td>7,605 KB</td>
</tr>
</tbody>
</table>

Table B.3: Sizes of additional CVT components

<table>
<thead>
<tr>
<th>Component</th>
<th># Lines</th>
<th>Source Size</th>
<th>Compressed</th>
<th>Compiled Size</th>
<th>Compressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>50,277</td>
<td>2,496 KB</td>
<td>298 KB</td>
<td>1,296 KB</td>
<td>413 KB</td>
</tr>
<tr>
<td>Modules</td>
<td>40,641</td>
<td>1,544 KB</td>
<td>326 KB</td>
<td>1,036 KB</td>
<td>331 KB</td>
</tr>
</tbody>
</table>

Table B.4: Total size of all components listed in previous tables

<table>
<thead>
<tr>
<th>Component</th>
<th># Lines</th>
<th>Source Size</th>
<th>Compressed</th>
<th>Compiled Size</th>
<th>Compressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Size</td>
<td>698,812</td>
<td>35,700 KB</td>
<td>6,574 KB</td>
<td>26,550 KB</td>
<td>9,386 KB</td>
</tr>
</tbody>
</table>
Appendix C

Example Source Code

This appendix contains source code from a number of examples used in the main text of this thesis. All of the source code is written in the Java programming language. All code in this appendix is compatible with versions 1.3.x and 1.4.x of Java and the versions of JHDL, JAB, and CVT at the time of publication of this thesis.

C.1 Custom Visualization Using JAB

This section contains code to demonstrate the customizability features in JAB. This is the code used to generate the examples shown in Section 3.3. Some of this code comes from the JHDL documentation[25].

This section includes the following source code files:

- myffd.java
- Ring.java
- RingObserver.java
- tbRing.java

myffd.java

```java
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;
import byucc.jhdl.apps.Viewers.Schematic.*;

/** Example code from JHDL documentation. This class shows
 * how to use the UserDefinedSchematic interface to implement
 * a custom schematic view of the circuit. */
public class myffd extends Logic
    implements UserDefinedSchematic{
```
public static CellInterface[] cell_interface = {
    in("d",1),
    out("q",1),
};

private Wire q_out;

/** Constructor */
public myffd(Node parent, Wire d, Wire q) {
    super(parent);
    connect("d",d);
    q_out = connect("q",q);
    reg_o(d,q); //Make a d flip-flop with an implicit clock.
    //Don’t forget there is an implicit clock.
}

public static final int X_SIZE=25, Y_SIZE=40;

/** this is to set up the schematic size and the ports */
public void initUserDefinedNode(UserDefinedNode udn){
    udn.setSize(X_SIZE, Y_SIZE);
    udn.addInPort(0,10,"d");
    // This is the implicit clock port.
    udn.addInPort(0,20,"c");
    udn.addOutPort(X_SIZE - 5+5, Y_SIZE-7,"q");
}

/** Draw everything else--be creative with the */
/** circuit’s state. */
public void paint(UserDefinedNode udn){
    udn.drawRect(0,0,X_SIZE,Y_SIZE);
    //Check to see if the output is one
    if (q_out.get(this) != 0 ) {
        //Draw grid lines on Cell if output one.
        for(int i=0; i < X_SIZE; i+=8)
            udn.drawLine(i,0,i,Y_SIZE);
        for(int i=0; i < Y_SIZE; i+=8)
            udn.drawLine(0,i,X_SIZE,i);
    }
}

} // end class myffd

Ring.java

import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

/** A simple demo showing how custom schematics are useful. */
public class Ring extends Logic{
    public static final int RING_LENGTH = 8;
    public static CellInterface[] cell_interface = {

public Ring(Node parent, Wire o) {
    super(parent);
    connect("o", o);

    Wire inv_in = o;
    Wire inv_out = not(inv_in);

    Wire ff_in = inv_out;
    Wire ff_out = wire(1, "ffOut0");

    for (int i=0; i<RING_LENGTH; i++) {
        new myffd(this, ff_in, ff_out);
        // connect this output to the next input.
        ff_in = ff_out;

        // if the next one is the last one--
        if (i == (RING_LENGTH - 2))
            // connect it’s output to the inverter.
            ff_out = inv_in;
        else
            // otherwise make a new wire to connect to
            // the new output.
            ff_out = wire(1, "ffOut"+(i+1));
    }
}

public class RingObserver extends JFrame implements Browser, SimulatorCallback {
    /** Constructor */
    public RingObserver(Wire[] flopOutputs) {
        this.flopOutputs = flopOutputs;
    }
    // end class Ring

    /** This class implements SimulatorCallback and Browser to observe the state of the outputs of the flip-flops in the ring design. The result is a row of buttons that shows whether each flop is on or off (1 or 0). * @author Anthony L. Slade */
    public class RingObserver extends JFrame implements Browser, SimulatorCallback {
        /** Constructor */
        public RingObserver(Wire[] flopOutputs) {
            this.flopOutputs = flopOutputs;
        }
    }
}
buildGUI();

/***** SimulatorCallback methods: *****

public void simulatorReset() {
    setStates();
}

public void simulatorRefresh(int cycle, int step) {
    setStates();
}

public void simulatorUpdate(int cycle, int step) {
    // ignore this method, it’s too fine-grained for GUIs
}

/** This method sets the states of the output visuals. */
private void setStates() {
    for (int bi = 0; bi < 8; ++bi)
        try {
            buttons[bi].setSelected(flopOutputs[bi].get(this) == 1);
        } catch (WireValueException wve) {
            // something funny happened.
            buttons[bi].setSelected(false);
        }
}

/** This method just initializes the contents of this
 * graphical viewer. */
private void buildGUI() {
    buttons = new AbstractButton[8];
    JPanel panel = new JPanel(new GridLayout(1,8));
    panel.setBorder(BorderFactory.createTitledBorder("Custom View"));
    for (int bi = 0; bi < 8; ++bi) {
        AbstractButton button = new JRadioButton();
        button.setEnabled(false);
        panel.add(button);
        buttons[bi] = button;
    }
    getContentPane().add(panel);
    // show the new frame!
    setTitle("Custom View"); pack(); show();
}

/** Array of the output buttons */
private AbstractButton[] buttons;

/** Maintain references to the wires of interest */
private Wire[] flopOutputs;
} // end class RingObserver

### tbRing.java

```java
import byucc.jhdl.base.Browser;
import byucc.jhdl.base.HWSystem;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.TestBench;
import byucc.jhdl.base.Wire;
import byucc.jhdl.base.WireValueException;
```
```java
import byucc.jhdl.Logic.Logic;
/** This class acts as the TestBench for the ring design.
* It also creates an instance of RingObserver to show a
* custom graphical display of the state of the design.
* @author Anthony L. Slade */
public class tbRing extends Logic implements TestBench {
    /** Constructor */
    public tbRing(Node parent) {
        super(parent,"tbRing");
        // build ring design
        Wire output = wire(1,"output");
        new Ring(this, output);
        // build up array of flop outputs
        Wire[] ffOutputs = new Wire[8];
        final String namePrefix = "tbRing/Ring/ffOut";
        for (int oi = 0; oi < 7; ++oi )
            ffOutputs[oi] = (Wire)getSystem()
                .findNamed(namePrefix+oi);
        ffOutputs[7] = output;
        // create the custom viewer and
        // register it as a SimulatorCallback
        getSystem()
            .addSimulatorCallback(new RingObserver(ffOutputs));
    }
}
```

### C.2 Example Application: Audio Processing

This section contains the code for the example application in Section 5.1. The code for the application model (Section C.2.1) and view (Section C.2.2) is included, as well as additional code to support simulated data streams (Section C.2.3). In each subsection, the classes are listed in alphabetical order.

#### C.2.1 Audio Processing Application Model

This section includes the following source code files:

- AudioProc.java
- ClockDivider.java
- EchoGenerator.java
- EchoGeneratorBlockRAM.java
- StereoCodecIF.java
AudioProc.java

```java
package model;

import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;
import byucc.jhdl.Xilinx.Virtex.bufg;
import byucc.jhdl.Xilinx.Virtex.ibufg;
import byucc.jhdl.Xilinx.Virtex.obuf;

/**
 * This circuit implements a very basic echo function for the stereo CODEC on the XSV board. This is also meant to be a very simple demonstration application to show some of the nice features of the CVT system.
 * 
 * Note that this demonstration circuit assumes a clock frequency of 50 MHz. Set the XSV board frequency to 50 MHz for propery operation of this demo.
 * 
 * The file AudioProc.ucf is appropriate for using this design with the XSV board.
 * 
 * @author Mike Wirthlin, Anthony Slade
 */

public class AudioProc extends Logic {

    public static final int ECHO_DELAY = 5000;
    public static final int ECHO_ATTENUATION = 4;

    public static CellInterface cell_interface[] = {
        in("clk",1),
        in("sdout",1),
        out("mclk",1),
        out("lrlck",1),
        out("sclk",1),
        out("sdin",1),
        //out("testport",5),
        param("echoDelay", INTEGER ), // # cycles for echo delay
        //param("echoAttenuation", INTEGER ),
    };

    public AudioProc(Node parent, Wire clk, Wire sdout,
                        Wire mclk, Wire lrlck, Wire sclk,
                        Wire sdin
                        /*,Wire testport*/) {
        this(parent,clk,sdout,mclk,lrlck,sclk,sdin,/*testport,*/
```
public AudioProc(Node parent, Wire clk, Wire sdout,
          Wire mclk, Wire lrclk, Wire sclk,
          Wire sdin,
          int echoDelay) {

   // call constructor in Logic
   super(parent,"AudioProcDemo");

   // Connect calls.
   connect("clk",clk);
   connect("sdout",sdout);
   connect("mclk",mclk);
   connect("lrclk",lrclk);
   connect("sclk",sclk);
   connect("sdin",sdin);
   //connect("testport",testport);
   bind("echoDelay",echoDelay);
   //bind("echoAttenuation",echoAttenuation);

   // Create Clock Wire.
   Wire internal_clk = wire(1,"internal_clk");
   Wire internal_ibufg_clk = wire(1,"internal_clk");
   new ibufg(this,clk,internal_ibufg_clk);
   new bufg(this,internal_ibufg_clk,internal_clk);
   setDefaultClock(internal_clk);

   // Wires
   Wire sampleData = wire(1,"sampleData");
   Wire rdatain = wire(20,"rdatain");
   Wire ldatain = wire(20,"ldatain");
   Wire rdataout = wire(20,"rdataout");
   Wire ldataout = wire(20,"ldataout");
   Wire internal_testport = wire(5,"internal_testport");

   // Create the internal module that actually performs the
   // echo generation.
   new EchoGenerator(this, internal_clk,
                    rdatain,ldatain,
                    sampleData,
                    rdataout,ldataout);

   // Interface to StereoCodec
   new StereoCodecIF(this,
                     internal_clk,// fast clock
                     ldataout, // output sample (left)
                     rdataout, // output sample (right)
sdout, // Serial in pin
ldatina, // Left Data in
rdatina, // Right data in
mclk, // mclk pin
lrclk, // lrclk pin
sclk, // sclk pin
sdin, // sdin pin (serial out)
sampleData, // increment signal
4//, // divide 50 MHz by 4 (for mclk)
/*internal_testport*/);

// add testport pins
//for (int i=0;i<internal_testport.getWidth();i++)
// new obuf(this,internal_testport.gw(i),
// testport.gw(i));

}

ClockDivider.java

package model;

import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;
import byucc.jhdl.Xilinx.Virtex.*;
import byucc.jhdl.Xilinx.Virtex.Modules.upcnt;

/**
 * This class creates a programmable clock divider. The
 * output is a regular signal (not a clock buffer). A clock
 * can be created by attaching a clock buffer to the output
 * of this circuit. Several of the XSV I/O components
 * require custom clocks. This circuit is used to facilitate
 * the creation of clock dividers for these resources.
 * <br><br>
 * This clock divider can divide the current clock by any
 * power of 2.
 * *
 * @author Anthony L. Slade
 **/
public class ClockDivider extends Logic {

public static CellInterface cell_interface[] = {
in("clk",1),
out("cout",1),
};

/**
 *
/**
 * public ClockDivider(Node parent, Wire clk, Wire cout,
 * int divide) {
 * super(parent);
 * this.clk = connect("clk", clk);
 * this.cout = connect("cout", cout);
 * setDefaultClock(this.clk);
 *
 * if ( divide == 1 ) {
 * buf_o(clk, cout);
 * return;
 * }
 *
 * Wire fout = cout;
 * Wire fin = null;
 * for ( int shift = 0; (1&(divide>>shift)) == 0;
 * ++shift ) {
 * fin = wire(1);
 * reg_o(fin, fout);
 * fout = fin;
 * } not_o(cout,fin);
 * }
 *
 * Wire clk;
 * Wire cout;
 */

EchoGenerator.java

package model;

import byucc.jhdl.base.BuildException;
import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;
import byucc.jhdl.Xilinx.Virtex.Modules.delay;
import byucc.jhdl.Xilinx.Virtex.Modules.upcnt;

/** The internal echo generation circuit for the AudioProc *
 * @author Anthony L. Slade */
public class EchoGenerator extends Logic {

public static final int NUM_DELAY_SRL = 1536;

public static CellInterface[] cell_interface = {
    in("clk", 1), // The clock for this circuit
    in("rightDataIn", 20), // Input data for right channel
    in("leftDataIn", 20), // Input data for left channel
    in("sampleData", 1), // Signals to sample the data
out("rightDataOut",20), // Output data for right channel
out("leftDataOut",20), // Output data for left channel
//param("echoDelay", INTEGER ), // # delay cycles
//param("attenuation", INTEGER )// shift amount

/**
 * @param parent The parent Cell that creates this circuit
 * @param clk the clock for this circuit
 * @param rDataIn Right channel input
 * @param lDataIn Left channel input
 * @param sampleData Sample enable signal
 * @param rDataOut Right channel output
 * @param lDataOut Left channel output
 * @param echoDelay Number of cycles to delay for echo
 * @param attenuation Number of bits to shift delayed signal to attenuate the echo signal -->
 */

public EchoGenerator(Node parent,
    Wire clk,
    Wire rDataIn,
    Wire lDataIn,
    Wire sampleData,
    Wire rDataOut,
    Wire lDataOut
) {
    super(parent,"EchoGenerator");

    // connect calls
    Wire clk_i = connect("clk",clk);
    Wire rDIn_i = connect("rightDataIn",rDataIn);
    Wire lDIn_i = connect("leftDataIn",lDataIn);
    Wire sample_i = connect("sampleData",sampleData);
    Wire rDOut_i = connect("rightDataOut",rDataOut);
    Wire lDOut_i = connect("leftDataOut",lDataOut);
    //bind("echoDelay",echoDelay);
    //bind("attenuation",attenuation);

    setDefaultCloseOperation(clk_i);

    Wire inputAddress = wire(12,"inputAddress");
    Wire outputAddress = wire(12,"outputAddress");
    new upcnt(this,clk_i,// clock wire
        sample_i,// clock enable
        and(inputAddress.gw(11),// counter load
            inputAddress.gw(10),
            inputAddress.gw(8)),
        gnd(12),// counter load data
        inputAddress,// counter output

    // Call super
    super(parent,"EchoGenerator");

    // connect calls
0L, //counter reset state
"inputAddressCounter");

new upcnt(this, clk_i, //clock wire
  sample_i, //clock enable
  and(outputAddress.gw(11), //counter load
    outputAddress.gw(10),
    outputAddress.gw(8)),
  gnd(12), //counter load data
  outputAddress, //counter output
  1L, //counter reset state
  "outputAddressCounter");

Wire toDelayDataR = rDataIn.range(19,4,"toDelayDataR");
Wire toDelayDataL = lDataIn.range(19,4,"toDelayDataL");
Wire delayedDataRAMR = wire(16,"delayedDataRAMR");
Wire delayedDataRAML = wire(16,"delayedDataRAML");

new EchoGeneratorBlockRAM(this,
  clk_i,
  toDelayDataR,
  sample_i,
  inputAddress,
  outputAddress,
  delayedDataRAMR,
  "RightChannelDelayRAM");

new EchoGeneratorBlockRAM(this,
  clk_i,
  toDelayDataL,
  sample_i,
  inputAddress,
  outputAddress,
  delayedDataRAML,
  "LeftChannelDelayRAM");

Wire delayedDataL = wire(16,"delayedDataR");
Wire delayedDataR = wire(16,"delayedDataL");

new delay(this,
  NUM_DELAY_SRL,
  clk_i,
  delayedDataRAMR,
  sample_i,
  delayedDataR,
  "srlDelayR");

new delay(this,
  NUM_DELAY_SRL,
  clk_i,
  delayedDataRAML,
  sample_i,
  delayedDataL,
  "srlDelayL");

add_o(rDIn_i,concat(gnd(4),delayedDataR),rDOut_i);
add_o(lDIn_i,concat(gnd(4),delayedDataL),lDOut_i);
}
package model;

import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;
import byucc.jhdl.Logic.Modules(DECODERS);
import byucc.jhdl.Xilinx.Virtex.RAMB4Dual;
import byucc.jhdl.Xilinx.Virtex.Modules.mux41;
import byucc.jhdl.Xilinx.Virtex.Modules.mux81;

/** The internal echo generation circuit delay using Block
 * RAMs for the AudioProc design.
 * @author Anthony L. Slade */

public class EchoGeneratorBlockRAM extends Logic {

    public static CellInterface[] cell_interface = {
        in("clk",1),
        in("dataIn",16),
        in("sample",1),
        in("addressIn",12),
        in("addressOut",12),
        out("dataOut",16),
    };

    public EchoGeneratorBlockRAM(Node parent, Wire clk, Wire dataIn, Wire sample, Wire addressIn, Wire addressOut, Wire dataOut, String name) {
        super(parent,name);

        Wire clk_i = connect("clk",clk);
        setDefaultClock(clk_i);
        Wire dataIn_i = reg(connect("dataIn",dataIn),"dataIn");
        Wire sample_i = reg(connect("sample",sample),"sampleData");
        Wire inputAddress = reg(connect("addressIn",addressIn),"inputAddress");
        Wire outputAddress = reg(connect("addressOut",addressOut),"outputAddress");
        Wire delayedData = connect("dataOut",dataOut);

        int[] zeroContents = new int[128];
        for ( int zi = 0; zi < zeroContents.length; ++zi ) {
            zeroContents[zi] = 0;
            //zeroContents[zi] = zi;
        }

        Wire[] inputRAMEnable = new Wire[16];
Wire[] delayedDataArray = new Wire[14];
for ( int wi = 0; wi < 14; ++wi ) {
    inputRAMEnable[wi] = wire(1,"inputRAMEnable"+wi);
    delayedDataArray[wi] = wire(16,"delayedData"+wi);
}
for ( int wi = 14; wi < 16; ++wi ) {
    inputRAMEnable[wi] = wire(1,"inputRAMEnable"+wi);
}

Wire inputRAMSelect
    = inputAddress.range(11,8,"inputRAMSelect");
Wire outputRAMSelect
    = outputAddress.range(11,8,"outputRAMSelect");
DECODERS.decoder(this, inputRAMSelect, inputRAMEnable, "inputRAMEnable");

Wire inputAddressLow
    = inputAddress.range(7,0,"inputAddressLow");
Wire outputAddressLow
    = outputAddress.range(7,0,"outputAddressLow");
for ( int br = 0; br < 14; ++br ) {
    new RAMB4Dual(this,//parent
        clk_i,clk_i,//clocks
        gnd("rstA"),//rstA
        vcc("enA"),//enA
        dataIn_i,//diA
        and(sample_i,inputRAMEnable[br]),//weA
        inputAddressLow,//addrA
        wire(16,"doA"),//doA
        gnd("rstB"),//rstB
        vcc("enB"),//enB
        gnd(16,"diB"),//diB
        vcc("weB"),//weB
        outputAddressLow,//addrB
delayedDataArray[br],//doB
        zeroContents,//contents[]
        "delayRam");//name
}

Wire[] delayedDataArrayReg = new Wire[14];
for ( int wi = 0; wi < 14; ++wi ) {
    delayedDataArrayReg[wi] = reg(delayedDataArray[wi],
        "delayedDataArrayReg"+wi);
}

Wire mux81Out0 = wire(16,"mux81Out0");
Wire mux81Out1 = wire(16,"mux81Out1");
Wire outputRAMSelectHi
    = outputRAMSelect.range(3,3,"outputRAMSelectHi");
Wire outputRAMSelectLow
    = outputRAMSelect.range(2,0,"outputRAMSelectLow");
new mux81(this,
    delayedDataArrayReg[0],
    delayedDataArrayReg[1],
delayedDataArrayReg[2],
delayedDataArrayReg[3],
delayedDataArrayReg[4],
delayedDataArrayReg[5],
delayedDataArrayReg[6],
delayedDataArrayReg[7],
outputRAMSelectLow,  
mux81Out0,"delayedDataMux0");

new mux81(this,
delayedDataArrayReg[8],
delayedDataArrayReg[9],
delayedDataArrayReg[10],
delayedDataArrayReg[11],
delayedDataArrayReg[12],
delayedDataArrayReg[13],
gnd(16,"emptyUpperMuxWire14"),
gnd(16,"emptyUpperMuxWire15"),
outputRAMSelectLow,  
mux81Out1,"delayedDataMux1");
mux_o(mux81Out0,
mux81Out1,  
outputRAMSelectHi,  
delayedData,  
"delayedDataMux");

StereoCodecIF.java

package model;

import byucc.jhdl.base.Wire;
import byucc.jhdl.base.SimulatorCallback;
import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.Logic.Logic;
import byucc.jhdl.Xilinx.Virtex.ofd;
import byucc.jhdl.Xilinx.Virtex.ifd;

/**
 * This class is a simple interface for the Asahi Kasei
 * stereo delta-sigma CODEC (AK4520A) used on the XSV board.
 * See Sheet 7 (page 52) on the XSV schematic drawings for
 * details of the Codec wiring. Also, see the AK4520A
 * datasheet for device specific
 * interfacing/operation.<br><br>
 * The XSV board has hard-wired the Codec in the following
 * manner:<br>
 * <li> DIF0 is hardwired to GND
 * <li> DIF1 is hardwired to VDD
 * <li> DEM0 is hardwired to VDD
 */
* DEM1 is hardwired to GND
* CMODE is hardwired to GND

These constraints force the chip into the following modes:

* MCLK must be set to 256 * fs (sampling frequency) (See Table 1 in the AK4520A datasheet)
* The serial data mode is hard wired to Mode 2 (10). This requires the SDTO data to be MSB justified and the SDTI data to be MSB justified. L/R = H/L and SCLK >= 40 * fs. (See Table 2 in the AK4520A datasheet)
* The de-emphasis filter is disabled.

This codec can sample data anywhere from 16kHz to 54 kHz. The actual sample rate is determined by the user of this module. The sample rate is specified in this module by setting the <tt>clkdivide</tt> parameter in the port interface. This parameter is used to create a new internal clock (mclk) that specifies the sample rate. See the javadoc comment on the <tt>clkdivide</tt> parameter for more information about setting this parameter.

For this module, the sampling clock rate is the frequency of MCLK divided by 256 (mclk/256). This forces the input clock to have a frequency between 4.096 MHz and 13.824 MHz.

This module consumes about 58 slices.

Five external I/O pins are needed to support this module. The pin locations for these nets can be forced by using the following UCF File snippet:

*NET "sdout_pin" LOC = "P7";
*NET "mclk_pin" LOC = "P3";
*NET "sclk_pin" LOC = "P5";
*NET "lrclk_pin" LOC = "P4";
*NET "sdin_pin" LOC = "P6";
</pre>

@author Mike Wirthlin

**/

public class StereoCodecIF extends Logic implements SimulatorCallback {

  public static CellInterface cell_interface[] = {
    in("clk",1),  // Module Clock
    in("ldatain",20),  // Left Data In
    in("rddatain",20),  // Right Data In
    in("sdout_pin",1),  // Codec Serial Data (from Codec)
    out("ldataout",20),  // Left Data Out
out("rdataout",20), // Right Data Out
out("mclk_pin",1), // Codec Master Clock Input
out("lrclk_pin",1), // Codec In/Out Channel Clock Pin
out("sclk_pin",1), // Codec Audio serial data clock pin
out("sdin_pin",1), // Codec Serial Data (to Codec)
out("newsample",1), // New sample (pre-sampling cycle)
param("clkdivide", INTEGER ), // clock divide
}

/**
* @param parent The parent Cell that creates this circuit
*
* @param clk The input clock to the module. This should be
* the system clock operating throughout the rest of the
* design. This clock will be divided internally to
* generate the appropriate internal clock signals. This
* module has been successfully tested with a 50 MHz
* clock.
* 
* @param ldatain The 20-bit left channel data digitized by
* the codec. This data is sampled on the clock cycle
* following the cycle when "newsample" is asserted.
* 
* @param rdatain The 20-bit right channel data digitized
* by the codec. This data is sampled on the clock cycle
* following the cycle when "newsample" is asserted.
* 
* @param sdout_pin The 1 bit serial input signal from the
* CODEC (it is an output on the CODEC). The user of this
* circuit does not need drive this signal - a pin with the
* proper location properties is created for this signal
* within this module.
* 
* @param ldataout The 20-bit left sample quantized by the
* CODEC (data digitized from the Audio In port on the
* board). This data should be sampled on the clock cycle
* following the "newsample" signal.
* 
* @param rdataout The 20-bit right sample quantized by the
* CODEC (data digitized from the Audio In port on the
* board). This data should be sampled on the clock cycle
* following the "newsample" signal.
* 
* @param mclk_pin This is the master clock used by the
* Codec. This clock must be 256 * fs. If the sampling
* frequency is 48 kHz, this clock should be 12.288
* MHz. This clock is generated by dividing the input clock
* by the integer value specified by the clkdivide
* parameter. This module will create the logic to
* generate the correct MCLK signal and will create the
* output buffer for the signal. The user of this module
* should not use this signal.
* 
* @param lrclk_pin This is the left/right clock
*/
* signal. This proper timing for this signal is generated by this module. This signal is connected to the appropriate IOB within this module and should not be used by the user circuit.

* @param sclk_pin The serial data clock going to the Codec. This clock will operate at 64 * fs. The IOB for this signal is created within this module.

* @param sdin_pin This is the signal that will drive the serial input to the CODEC. The logic and IOB for this signal is created within this module.

* @param newsample When this signal is asserted, the user design should sample the rdataout, ldataout signals and should provide a new sample for the analog out. This signal is asserted for one clock cycle every 1024 system clocks.

* @param clkdivide The number of clock cycles to divide the incoming clock signal into the MCLK signal. Since the MCLK must be 256 * fs (sampling frequency), the clkdivide value can be calculate as follows: clkdivide = sysclk / (fs * 256), where sysclk is the frequency of the incoming clock signal. For example, if the incoming clock has a frequency of 50 MHz and a sampling frequency of 48 kHz is desired, the clkdivide value should be set to: 50 MHz / (48k * 256) = 4.069 or 4. This value does not need to be a power of two.

**/


// Call super
super(parent,"StereoCodecIF");

// connect calls
clk_i = connect("clk",clk);
ldatain_i = connect("ldatain",ldatain);
rdatain_i = connect("rdatain",rdatain);
sdout_i = connect("sdout_pin",sdout_pin);
ldataout_i = connect("ldataout",ldataout);
rdataout_i = connect("rdataout",rdataout);
mclk_i = connect("mclk_pin",mclk_pin);
lrclk_i = connect("lrclk_pin",lrclk_pin);
sclk_i = connect("sclk_pin",sclk_pin);
sdin_i = connect("sdin_pin",sdin_pin);
newsample_i = connect("newsample",newsample);
bind("clkdivide",clkdivide);
Although all the logic in this circuit is clocked by the system clock (whatever is default at this point), this circuit changes its values at the internal "MCLK" frequency. An internal "MCLK" signal (not clock) is generated by dividing the input clock by the integer parameter "clkdivide".

The signal "new_mclk" is used in the design to signal the logic that the MCLK will be going high on the next clock signal. Internal variables will use this signal to make sure that things only change when MCLK is going high.

```verbatim
Wire pre_internal_mclk = wire(1,"pre_internal_mclk"); new ClockDivider(this,clk,pre_internal_mclk,clkdivide); internal_mclk = reg(clk,pre_internal_mclk,"internal_mclk");

Wire new_mclk;
if (clkdivide == 1)
// If clkdivide == 1, there is a new MCLK every cycle
new_mclk = vcc("new_mclk");
else
// If clkdivide != 1, there is a new MCLK only when
// pre_internal_mclk ==1 and internal_mclk == 0
new_mclk = and(pre_internal_mclk,not(internal_mclk),"new_mclk");
```

Create the output buffer and flip-flop for this signal. Note that all output and input IOBs that communicate with the StereoCodec use IOB flip-flops that are clocked by the higher-frequency system clock.

```verbatim
// Create the output buffer for
new ofd(this,"mclk_pin",clk,internal_mclk,mclk_i);
// mclk_i.addProperty(this,"LOC","P3");
```

Create a 256-bit counter that is incremented each time new_mclk is asserted. This 256-bit MCLK Counter defines the complete sequence of reading and writing a left and write sample from/to the CODEC. This is essentially a 256-sequence statemachine.

All output logic will use this 256-bit counter to determine their values. (i.e. the output forming logic uses the counter).

```verbatim
Wire mclk_counter = wire(8,"mclk_counter");
Wire mclk_counter_plus_1 = add(mclk_counter,
```
constant(8,1));
regce_o(clk,mclk_counter_plus_1,new_mclk,mclk_counter);

// create SCLK: bit 1 (divide mclk by four). This signal
// must be 180 deg out of phase with LRCLK.
internal_sclk = buf(mclk_counter.gw(1),"internal_sclk");
new ofd(this,"sclk_pin",clk,internal_sclk,sclk_i);
// sclk_i.addProperty(this,"LOC","P5");

// create LRCLK: bit 7
internal_lrclk = buf(mclk_counter.gw(7),
                     "internal_lrclk");
new ofd(this,"lrclk_pin",clk,internal_lrclk,lrclk_i);
// lrclk_i.addProperty(this,"LOC","P4");

/* The input data from the CODEC must be sampled on the
* falling edge of SCLK. This occurs when the 8-bit
* mclk_counter is XXXXX11 and the new_mclk signal
* is asserted.
*
* MCLK counter
* 00000011 - Sample Right bit 19
* 00000111 - Sample Right bit 18
* ... 
* 01001111 - Sample Right bit 0
* 01010011 - No bit
* 01010111 - No bit
* ... 
* 10000011 - Sample Left bit 19
* 10000111 - Sample Left bit 18
* ... 
* 11001111 - Sample Left bit 0
* 11010011 - No bit
* 11010111 - No bit
* ... 
* 11111111 - No bit
*
* Sample/shift a bit
* = CNT0*CNT1’*(CNT6’+CNT6*CNT4’*CNT5’)*NEW_MCLK
*/

// sample_bit: asserted when counter is in a SCLK cycle
// that represents a valid bit (i.e. the first twenty
// bits associated with bits 19 through 0 of the
// sample). The following 12 SCLK cycles contain invalid
// data. i.e. (CNT6’+CNT6*CNT4’*CNT5’)*NEW_MCLK
wire sample_bit = and(new_mclk,
                     or(not(mclk_counter.gw(6)),
                        and(mclk_counter.gw(6),
                            not(mclk_counter.gw(4)),
                            not(mclk_counter.gw(5))))),
                    "sample_bit");
// Sample the left input signal. This occurs when
// in a valid SCLK cycle AND on falling edge of
// SCLK (XXXXXX01) AND LRCLK=1 (mclk_counter.gw(7))
Wire sample_left_input = and(sample_bit,
    mclk_counter.gw(0),
    not(mclk_counter.gw(1)),
    mclk_counter.gw(7),
    "sample_left_input");

// Sample the right input signal. This occurs when
// in a valid SCLK cycle AND on falling edge of
// SCLK (XXXXXX01) AND LRCLK=0 (mclk_counter.gw(7))
Wire sample_right_input = and(sample_bit,
    mclk_counter.gw(0),
    not(mclk_counter.gw(1)),
    not(mclk_counter.gw(7)),
    "sample_right_input");

// SDOUT pin: internal_sdout is the internal wire that
// contains the serial data coming from the CODEC.
internal_sdout = wire(1,"internal_sdout");
new ifd(this,"sdout_pin",clk,sdout_i,internal_sdout);
//sdout_i.addProperty(this,"LOC","P7");

// Next left output. Shifted left with LSB coming from
// internal serial data from codec.
Wire next_left_output_reg = concat(ldataout.range(18,0),
    internal_sdout);
// left output register.
regce_o(clk,next_left_output_reg,sample_left_input,
    ldataout);

// Next right output. Shifted left with LSB coming from
// internal serial data from codec.
Wire next_right_output_reg = concat(rdataout.range(18,0),
    internal_sdout);
// right output register.
regce_o(clk,next_right_output_reg,sample_right_input,
    rdataout);

// SDIN pin: internal_sdin is the internal wire that will
// go to the CODEC.
internal_sdin = wire(1,"internal_sdin");
new ofd(this,"sdin_pin",clk,internal_sdin,sdin_i);
//sdin_i.addProperty(this,"LOC","P6");

// Shift the 20-bit left input value. This occurs during
// a valid SCLK bit (see note above) and during
// the RISING edge of SCLK (i.e. XXXXXX11)
// AND when LRCLK=1
Wire shift_left_input = and(sample_bit,
    mclk_counter.gw(0),
    mclk_counter.gw(1),
    mclk_counter.gw(7),
    "shift_left_input");

// Shift the 20-bit right input value. This occurs during
// a valid SCLK bit (see note above) and during
// the RISING edge of SCLK (i.e. XXXXXX11)
// AND when LRCLK=0
Wire shift_right_input = and(sample_bit,
   mclk_counter.gw(0),
   mclk_counter.gw(1),
   not(mclk_counter.gw(7)),
   "shift_right_input");

// Sample the 20-bit values from the user. This occurs
// when the count is 0x7f AND we have a new_mclk (this
// signal is also used to generate the "newsample" signal
// for the user)
Wire sample_new_output_data
   = and(new_mclk,
      and(mclk_counter.gw(4),mclk_counter.gw(5),
         mclk_counter.gw(6),not(mclk_counter.gw(7))),
      and(mclk_counter.range(3,0)),
      "sample_new_output_data");

// left input register. This signal is parallel loaded
// during sample_new_output_data and it is shifted
// serially when shift_left_input is asserted.
Wire left_input = wire(20,"left_input");
Wire next_left_input_reg
   = mux(concat(left_input.range(18,0),gnd()),
        ldatain,sample_new_output_data);
Wire left_load
   = or(shift_left_input,
        sample_new_output_data,"left_load");
regce_o(clk,next_left_input_reg,left_load,left_input);

// right input register. This signal is parallel loaded
// during sample_new_output_data and it is shifted
// serially when shift_right_input is asserted.
Wire right_input = wire(20,"right_input");
Wire next_right_input_reg
   = mux(concat(right_input.range(18,0),gnd()),
        rdatain,sample_new_output_data);
Wire right_load
   = or(shift_right_input,sample_new_output_data,
        "right_load");
regce_o(clk,next_right_input_reg,right_load,right_input);

// This signal driving the CODEC serial input is chosen
// from the MSB of the two different registers. lrclk
// decides which one goes
mux_o(right_input.gw(19),left_input.gw(19),
     internal_lrclk,internal_sdin);

// newsample output
buf_o(sample_new_output_data,newsample_i);
/**
 * This constructor will create the same circuit with additional testport pins. Since the sdout, mclk, lrclk, sclk, and sdin wires are I/O wires, they cannot be connected to any internal logic (for testing or probing). This constructor creates two additional ports to provide wires that can be used by the user and routed to different pins for debugging.
 *
 * @param testport A 5-bit wire that contains the sdout (0), mclk (1), lrclk (2), sclk (3), and the sdin (4).
 *
 **/

public StereoCodecIF( Node parent, Wire clk, Wire ldatain, Wire rdatain, Wire sdout,
                     Wire ldataout, Wire rdataout, Wire mclk, Wire lrclk,
                     Wire sclk, Wire sdin, Wire newsample, int clkdivide,
                     Wire testport) {
    this(parent, clk, ldatain, rdatain,
          sdout, ldataout, rdataout, mclk, lrclk, sclk, sdin,
          newsample, clkdivide);

    // dynamically add and connect the testport
    addPort( out("testport",5) );
    connect("testport",testport);

    // connect the testport bits to the appropriate wires in the circuit
    buf_o(internal_sdout,testport.gw(0));
    buf_o(internal_mclk,testport.gw(1));
    buf_o(internal_lrclk,testport.gw(2));
    buf_o(internal_sclk,testport.gw(3));
    buf_o(internal_sdin,testport.gw(4));
}

public void simulatorRefresh(int cycle, int phase) {}
public void simulatorUpdate(int cycle, int phase) {
    if (phase == 1) {
        // if next cycle is positive edge of clock
        // New sample: Print some output
        if (newsample_i.get(this)==1) {
            System.out.print("New Sample: In Left=0x"+
                             Integer.toHexString(ldataout_i.get(this)));
            System.out.print("In Right=0x"+
                             Integer.toHexString(rdataout_i.get(this)));
            System.out.print("Out Left=0x"+
                             Integer.toHexString(ldatain_i.get(this)));
            System.out.println("Out Right=0x"+
                               Integer.toHexString(rdatain_i.get(this)));
        }
    }
}
public void simulatorReset() {} 

// Assuming the maximum input frequency is 100 MHz,
// Max divide = 100MHz / (256 * 16000) = 24.41 or 25.
public static final int MAX_DIVIDE = 25;

public Wire clk_i;
public Wire ldatain_i;
public Wire rdatain_i;
public Wire sdout_i;
public Wire rdataout_i;
public Wire ldataout_i;
public Wire mclk_i;
public Wire lrclk_i;
public Wire sclk_i;
public Wire sdin_i;
public Wire newsample_i;

public Wire internal_mclk;
public Wire internal_sclk;
public Wire internal_lrclk;
public Wire internal_sdout;
public Wire internal_sdin;

}

tbAudioProc.java

package model;

import javax.swing.JFrame;

import byucc.jhdl.apps.Viewers.cvt.cvtFrame;
import byucc.jhdl.base.HWSystem;
import byucc.jhdl.base.ProgrammaticTestBench;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;
import byucc.jhdl.Logic.TechMapper;
import byucc.jhdl.Xilinx.Virtex.VirtexTechMapper;
import view.AudioProcView;

/** Program to execute and visualize the AudioProc design *
 * @author Anthony L. Slade */
public class tbAudioProc extends Logic
implements ProgrammaticTestBench {

public static void printUsage() {
}
public static void main(String[] args) {
    int cycles = -1;
    boolean netlist = false;
    boolean cvt = false;
    String fileName = null;
    boolean noGUI = false;
    for (int ai = 0; ai < args.length; ++ai) {
        String arg = args[ai];
        if ("-netlist".equals(arg)) {
            netlist = true;
        } else if ("-d".equals(arg)) {
            debug = true;
        } else if ("-cvt".equals(arg)) {
            cvt = true;
        } else if ("-nw".equals(arg)) {
            noGUI = true;
        } else if ("-i".equals(arg)) {
            try {
                waveFileIn = args[++ai];
            } catch (ArrayIndexOutOfBoundsException aioobe) {
                System.err.println("Must include input file name with -i option");
                printUsage();
                return;
            }
        } else if ("-o".equals(arg)) {
            try {
                waveFileOut = args[++ai];
            } catch (ArrayIndexOutOfBoundsException aioobe) {
                System.err.println("Must include output file name with -o option");
                printUsage();
                return;
            }
        } else if ("-f".equals(arg)) {
            try {
                fileName = args[++ai];
            } catch (ArrayIndexOutOfBoundsException aioobe) {
                System.err.println("Must include output file name with -f option");
                printUsage();
                return;
            }
        } else {
            try {
                cycles = Integer.parseInt(arg);
            } catch (NumberFormatException nfe) {
                System.err.println("Cannot parse the value " + arg + " as an integer");
                printUsage();
            }
        }
    }
}
return;
}
}
}
}
}

tbAudioProc tb = new tbAudioProc();
if ( netlist ) {
    TechMapper tm = getDefaultTechMapper();
    ((byucc.jhdl.Xilinx.TechMapper)tm).setInsertPads(false);  
    tm.netlist(tb.design, "audio.edn");
    return;
}

cvtFrame myCVT = null;
if ( cvt ) {
    myCVT = new cvtFrame(tb);
}
AudioProcView view = new AudioProcView(tb.getSystem());

if ( cycles > 0 ) {
    numCycles = cycles;
    if ( cvt ) {
        myCVT.getInterp().parseCommand("cycle"+cycles);
    } else {
        tb.execute();  
    }
}

System.out.println("Ran circuit for"+cycles+"cycles");
if ( null != fileName ) {
    view.writeToFile(fileName);
    System.out.println("Wrote data to the file"+fileName);
}

if ( !noGUI ) {
    JFrame frame = new JFrame("Audio Processing Circuit"  
+"Input & Output");
    frame.setContentPane(view);
    frame.pack();frame.setVisible(true);
}
if ( !cvt ) {
    System.out.println("Done executing. Now exiting...");
    System.exit(0);
}

private AudioProc design;
private CodecWave codec;

public tbAudioProc() {
    super(new HWSystem());
    setDefaultTechMapper(new VirtexTechMapper(true));
    clock = wire(l, "tbClock");
    clockDriver(clock, "01", "tbClock");
    setDefaultClock(clock);
    Wire sdout = wire(l, "sdout");
    Wire mclk = wire(l, "mclk");
}
Wire lrck = wire(1, "lrck");
Wire sclk = wire(1, "sclk");
Wire sdin = wire(1, "sdin");
//Wire testport = wire(5, "testport");
design = new AudioProc(this, clock, sdout, mclk, lrck, sclk, sdin); //, testport);
codec = new CodecWave(this, mclk, lrck, sclk, sdin, sdout, waveFileIn, waveFileOut);
clockVal = true;
}

private static int numCycles = 1024;
public void execute() {
  HWSystem system = getSystem();
  long count = 0;
  if (debug) {
    System.err.println(count);
  }
  while (!codec.isDone()) {
    long time0 = System.currentTimeMillis();
    system.cycle(numCycles);
    long time1 = System.currentTimeMillis();
    count += numCycles;
    long time = time1 - time0;
    if (debug) {
      System.err.println(count + "; time(ms)=" + time + "; cycles-per-second=" + (((double)numCycles) / ((double)time)) * 1000.0) + "; ms-per-cycle=" + (((double)time) / ((double)numCycles)) / 1000.0));
    }
  }
}

public void reset() {
  //clock.putB(this, clockVal = true);
}

public void clock() {
  //clock.putB(this, clockVal = !clockVal);
}

private Wire clock;
private boolean clockVal;

private static String waveFileIn = CodecWave.WAVE_IN_DEFAULT;
private static String waveFileOut = CodecWave.WAVE_OUT_DEFAULT;

private static boolean debug = false;
C.2.2 Audio Processing Application View

This section includes the following source code files:

- AudioProcView.java
- WaveViewComponent.java
- WaveViewComponentData.java
- WaveViewPanel.java

AudioProcView.java

```java
package view;
import java.awt.Color;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.Random;
import javax.swingBoxLayout;
import javax.swing.JFrame;
import javax.swing.JPanel;
import byucc.jhdl.base.Browser;
import byucc.jhdl.base.HWSystem;
import byucc.jhdl.base.SimulatorCallback;
import byucc.jhdl.base.Wire;
import byucc.jhdl.base.WireValueException;

/** Custom view of the input and output of the AudioProc design 
 * @author Anthony L. Slade */

public class AudioProcView extends JPanel implements Browser, SimulatorCallback {

    public static final int DEFAULT_NUM_SAMPLES = 10000;
    public static final int MAX_VALUE = 1+(1<<9);
    //public static final int MAX_VALUE = 1+(1<<8);
    //public static final int MIN_VALUE = -MAX_VALUE;
    public static final int MIN_VALUE = -5;

    public static final String NAME_RIGHT_IN =
```
public static final String NAME_LEFT_IN = "tbAudioProc/AudioProcDemo/ldatain";
public static final String NAME_RIGHT_OUT = "tbAudioProc/AudioProcDemo/rdataout";
public static final String NAME_LEFT_OUT = "tbAudioProc/AudioProcDemo/ldataout";
public static final String NAME_SAMPLE_DATA = "tbAudioProc/AudioProcDemo/sampleData";

public static void main(String[] args) {
    boolean test = false;
    String fileName = null;
    for (int ai = 0; ai < args.length; ++ai) {
        String arg = args[ai];
        if ("-test".equals(arg)) {
            test = true;
        } else {
            fileName = arg;
        }
    }
    if (test) {
        JFrame frame = new JFrame("");
        AudioProcView view = new AudioProcView(new HWSystem());
        //Random rand = new Random();
        //int[] samples = new int[4];
        for (int ii = 0; ii < DEFAULT_NUM_SAMPLES; ++ii) {
            int lIn =
                (int)((MAX_VALUE*Math.abs(Math.cos(.005*Math.PI*ii))));
            int lOut = (lIn * 5) / 7;
            int rIn =
                (int)((MAX_VALUE>>1)*Math.cos(.01*Math.PI*ii+2));
            int rOut = (rIn * 5) / 7;
            rIn += (MAX_VALUE>>1);
            rOut += (MAX_VALUE>>1);
            view.addSamples(rIn,lIn,rOut,lOut,false);
        } /* for ( int ib = 0; ib < 4; ++ib ) { */
        view.addSamples(samples[0],samples[1],samples[2],samples[3],false);
        /* */
        frame.setContentPane(view);
        frame.pack();frame.setVisible(true);
        return;
    } else if (null != fileName) {
        AudioProcView view
public AudioProcView(HWSystem system) {
    this(system, DEFAULT_NUM_SAMPLES);
}

public AudioProcView(HWSystem system, int numSamples) {
    this(system, numSamples,
            NAME_RIGHT_IN, NAME_LEFT_IN,
            NAME_RIGHT_OUT, NAME_LEFT_OUT,
            NAME_SAMPLE_DATA);
}

public AudioProcView(HWSystem system, int numSamples,
                      String nameRightIn, String nameLeftIn,
                      String nameRightOut, String nameLeftOut,
                      String nameSampleData) {
    rdatain = (Wire) system.findNamed(nameRightIn);
    ldatain = (Wire) system.findNamed(nameLeftIn);
    rdataout = (Wire) system.findNamed(nameRightOut);
    ldataout = (Wire) system.findNamed(nameLeftOut);
    sampleData = (Wire) system.findNamed(nameSampleData);
    system.addSimulatorCallback(this);

    System.err.println("MIN_VALUE = \$0x" + Integer.toHexString(MIN_VALUE) + ";
                      MAX_VALUE = \$0x" + Integer.toHexString(MAX_VALUE));

    inR = new WaveViewPanel(MIN_VALUE, MAX_VALUE,
                            numSamples, "Channel 2 Input");
    inL = new WaveViewPanel(MIN_VALUE, MAX_VALUE,
                            numSamples, "Channel 1 Input");
    outR = new WaveViewPanel(MIN_VALUE, MAX_VALUE,
                            numSamples, "Channel 2 Output");
    outL = new WaveViewPanel(MIN_VALUE, MAX_VALUE,
                            numSamples, "Channel 1 Output");

    inR.setWaveColor(Color.red);
    outR.setWaveColor(Color.red.darker());
    inL.setWaveColor(WaveViewComponent.FOREGROUND_DEFAULT);
    outL.setWaveColor(WaveViewComponent.FOREGROUND_DEFAULT.darker());

    inputPanel = new JPanel();
    inputPanel.setLayout(new BoxLayout(inputPanel, BoxLayout.X_AXIS));
    JPanel outputPanel = new JPanel();
    outputPanel.setLayout(new BoxLayout(outputPanel, BoxLayout.X_AXIS));

public void addSamples(int inRSample, int inLSample, int outRSample, int outLSample, boolean repaint) {
    int offset = 0;
    inR.addSample(inRSample+offset, repaint);
    inL.addSample(inLSample+offset, repaint);
    outR.addSample(outRSample+offset, repaint);
    outL.addSample(outLSample+offset, repaint);
}

public void resetSample() {
    inR.resetSamples();
    inL.resetSamples();
    outR.resetSamples();
    outL.resetSamples();
}

public void simulatorReset() {
    resetSample();
}

public void simulatorUpdate(int cycle, int step) {
    if (lastSampleDataAsserted) {
        //System.err.println("putting samples");
        try {
            addSamples(rdatain.get(this), ldatain.get(this),
                       rdataout.get(this), ldataout.get(this),
                       false);
        } catch (WireValueException wve) {
            addSamples(0,0,0,0,false);
        }
        lastSampleDataAsserted = false;
    } else {
        try {
            lastSampleDataAsserted = sampleData.get(this)==1;
        } catch (WireValueException wve ) {
            lastSampleDataAsserted = false;
        }
    }
}

public void simulatorRefresh(int cycle, int step) {
    inR.repaint();
    inL.repaint();
    outR.repaint();
    outL.repaint();
public void writeToFile(String fileName) {
    WaveViewComponentData dataInR, dataInL, dataOutR, dataOutL;
    dataInR = inR.getData();
    dataInL = inL.getData();
    dataOutR = outR.getData();
    dataOutL = outL.getData();
    try {
        File file = new File(fileName);
        FileOutputStream fos = new FileOutputStream(file);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(dataInR);
        oos.writeObject(dataInL);
        oos.writeObject(dataOutR);
        oos.writeObject(dataOutL);
        oos.close();
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}

public static AudioProcView readFromFile(HWSystem system, String fileName) {
    AudioProcView view = new AudioProcView(system);
    WaveViewComponentData dataInR = null, dataInL = null, dataOutR = null, dataOutL = null;
    try {
        File file = new File(fileName);
        FileInputStream fis = new FileInputStream(file);
        ObjectInputStream ois = new ObjectInputStream(fis);
        dataInR = (WaveViewComponentData)ois.readObject();
        dataInL = (WaveViewComponentData)ois.readObject();
        dataOutR = (WaveViewComponentData)ois.readObject();
        dataOutL = (WaveViewComponentData)ois.readObject();
    } catch (IOException ioe) {
        ioe.printStackTrace();
        return null;
    } catch (ClassNotFoundException cnfe) {
        cnfe.printStackTrace();
        return null;
    }
    view.inR.setData(dataInR);
    view.inL.setData(dataInL);
    view.outR.setData(dataOutR);
    view.outL.setData(dataOutL);
    return view;
}

/** The panels to display the audio samples */
private WaveViewPanel inR, inL, outR, outL;

/** The wires of interest */
private Wire rdatain, ldatain, rdataout, ldataout, sampleData;

/** Used to tell when to sample the data */
private boolean lastSampleDataAsserted;
}

WaveViewComponent.java

package view;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.AdjustmentEvent;
import java.awt.event.AdjustmentListener;
import javax.swing.JComponent;
import javax.swing.JScrollBar;

public class WaveViewComponent extends JComponent implements AdjustmentListener {

public static final Color BACKGROUND
= Color.gray.brighter();
public static final Color BACKGROUND_DARKER
= BACKGROUND.darker();
public static final Color BACKGROUND_BRIGHTER
= BACKGROUND.brighter();
public static final Color FOREGROUND_DEFAULT
= Color.blue;

public WaveViewComponent(int minValue, int maxValue,
int[] samplesArray, boolean connected,
JScrollBar scrollBar) {
this(minValue,maxValue,new int[numSamples],
connected, scrollBar);
}

public WaveViewComponent(int minValue, int maxValue,
boolean connected,
JScrollBar scrollBar) {
this(new WaveViewComponentData(minValue,maxValue,
samplesArray),
connected, scrollBar);
}

public WaveViewComponent(WaveViewComponentData data,
boolean connected,
JScrollBar scrollBar) {
this.data = data;
totalValue = (double)(data.maxValue-data.minValue);
maxRatio = ((double)data.maxValue)/totalValue;
this.connected = connected;
this.scrollBar = scrollBar;
scrollBar.addAdjustmentListener(this);
}
public void addSample(int sample, boolean repaint) {
    try {
        data.samples[data.sIndex++] = sample;
    } catch (ArrayIndexOutOfBoundsException aioobe) {
        data.samples[0] = sample;
        data.sIndex = 1;
    }
    if (data.sIndex == data.sStart) {
        ++data.sStart;
        if (data.sStart == data.samples.length)
            data.sStart = 0;
    }
    if (repaint)
        repaint();
}

public WaveViewComponentData getData() {
    return data;
}

public void setData(WaveViewComponentData newData) {
    if (null != newData)
        data = newData;
}

public void resetSamples() {
    data.sIndex = 0;
    data.sStart = 0;
    for (int ind = 0; ind < data.samples.length; ++ind) {
        data.samples[ind] = 0;
    }
    repaint();
}

public void setConnected(boolean connected) {
    this.connected = connected;
    repaint();
}

public void zoomIn() {
    int visAmount = scrollBar.getVisibleAmount();
    if (1 >= visAmount)
        return;
    int decAmount = visAmount/10;
    if (0 == decAmount)
        decAmount = 1;
    visAmount -= decAmount;
    scrollBar.setVisibleAmount(visAmount);
    scrollBar.setBlockIncrement(visAmount);
}

public void zoomOut() {
```java
int visAmount = scrollBar.getVisibleAmount();
int incAmount = visAmount/10;
if ( 0 == incAmount )
    incAmount = 1;
visAmount += incAmount;
if ( visAmount > data.samples.length )
    visAmount = data.samples.length;
if ( ( scrollBar.getValue() + visAmount )
    > data.samples.length ) {
    scrollBar.setValue(data.samples.length-visAmount);
}
scrollBar.setVisibleAmount(visAmount);
scrollBar.setBlockIncrement(visAmount);
}

public void paint(Graphics gr) {
    int index = data.sStart+scrollBar.getValue();
    int nSamples = scrollBar.getVisibleAmount();
    int width = getWidth();
    int height = getHeight();
    int center = (int)(maxRatio*height);
    double xIncr = (width) / (nSamples-1.0);
    gr.setColor(BACKGROUND);
    gr.fillRect(0,0,width,height);
    gr.setColor(BACKGROUND_DARKER);
    gr.drawLine(0,center,width,center);
    gr.setColor(BACKGROUND_BRIGHTER);
    gr.drawLine(0,center+1,width,center+1);
    gr.setColor(getForeground());
    int lastX = 0;
    int lastY = center -
        (int)(height*(data.samples[index]/totalValue));
    for ( int cnt = 0; cnt < nSamples; ++cnt ) {
        try {
            smpl = data.samples[index+cnt];
        } catch (ArrayIndexOutOfBoundsException aioobe) {
            index -= data.samples.length;
            smpl = data.samples[0];
        }
        int x = (int)((double)cnt)*xIncr);
        int y = center;
        if ( smpl != 0 )
            y -= height*(double)smpl/totalValue;
        if ( connected ) {
            gr.drawLine(lastX,lastY,x,y);
            lastX = x; lastY = y;
        } else {
            gr.fillRect(x,y,2,2);
        }
    }
}
```
public void adjustmentValueChanged(AdjustmentEvent event) {
    repaint();
}

/** The core data of this component. Includes minValue, *
 * maxValue, and the samples array. */
WaveViewComponentData data;
/** The total of minValue and maxValue */
private double totalValue;
/** The ration of the total of minValue and maxValue that *
 * is made up of maxValue */
private double maxRatio;
/** If true a line will connected consecutive samples, *
 * otherwise, they will be represented by dots */
private boolean connected;
/** Used to determine location of sample space to display *
 * if the view is zoomed in. */
private JScrollBar scrollBar;

WaveViewComponentData.java

package view;

import java.io.Serializable;

/** Class to contain the core data for the WaveViewComponent *
 * class. This class also makes it easier to serialize the *
 * data for the WaveViewComponent class. *
 * @author Anthony L. Slade */
public class WaveViewComponentData implements Serializable {
    public WaveViewComponentData(int minValue,
                           int maxValue,
                           int[] samples) {
        this.minValue = minValue;
        this.maxValue = maxValue;
        this.samples = samples;
        sIndex = sStart = 0;
    }
    /** The set of samples collected */
    int[] samples;
    /** The minimum value */
    int minValue;
    /** The maximum value */
    int maxValue;
    /** The current index into the samples. This is where new *
    * samples are added */
    int sIndex;
    /** This is the start index of the samples. When we start *
    * to wrap around, this index will shift up */
    int sStart;
}
package view;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.BorderFactory;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollBar;

public class WaveViewPanel extends JPanel {

    public static final boolean DEFAULT_CONNECTED = true;
    public static final Dimension DEFAULT_MIN_WAVE_VIEW_SIZE = new Dimension(100, 80);

    public static void main(String[] args) {
        JFrame frame = new JFrame("Audio Wave Viewer");
        WaveViewPanel panel1 = new WaveViewPanel(-10, 10, 20, "Test Viewer 1");
        WaveViewPanel panel2 = new WaveViewPanel(-20, 10, 15, "Test Viewer 2");
        WaveViewPanel panel3 = new WaveViewPanel(-10, 10, 1500, "Test Viewer 3");
        BoxLayout bl = new BoxLayout(frame.getContentPane(), BoxLayout.Y_AXIS);
        frame.getContentPane().setLayout(bl);
        panel2.setWaveColor(Color.orange.brighter().brighter());
        panel3.setWaveColor(Color.pink.darker());
        frame.getContentPane().add(panel1);
        frame.getContentPane().add(panel2);
        frame.getContentPane().add(panel3);
        frame.pack(); frame.setVisible(true);

        int[] values = {
            0, -5, -5, -9, -4, -1, 3, 7, 9, 10, 5, 0, -7, -15, -16, -9,
        };
        for (int ii = 0; ii < values.length; ++ii) {
            panel1.addSample(values[ii], true);
            panel2.addSample(values[ii], true);
            for (int iii = 0; iii < values.length; ++iii) {
                panel3.addSample(values[iii], true);
            }
        }
    }
}
public WaveViewPanel(int minValue, int maxValue,  
    int numSamples,  
    String name) {  
    if (null != name) {  
        setBorder(BorderFactory.createTitledBorder(name));  
    }  
    setLayout(new BorderLayout());  
    scrollBar = new JScrollBar(JScrollBar.HORIZONTAL,  
        0, numSamples,  
        0, numSamples);  
    waveView = new WaveViewComponent(minValue, maxValue,  
        numSamples,  
        DEFAULT_CONNECTED,  
        scrollBar);  
    waveView.setMinimumSize(DEFAULT_MIN_WAVE_VIEW_SIZE);  
    waveView.setPreferredSize(DEFAULT_MIN_WAVE_VIEW_SIZE);  
    JPanel main = new JPanel(new BorderLayout());  
    main.add(waveView, BorderLayout.CENTER);  
    main.add(scrollBar, BorderLayout.SOUTH);  
    add(main, BorderLayout.CENTER);  
    add(createButtonBar(), BorderLayout.SOUTH);  
}

public void addSample(int sample, boolean repaint) {  
    waveView.addSample(sample, repaint);  
}

public WaveViewComponentData getData() {  
    return waveView.getData();  
}

public void setData(WaveViewComponentData data) {  
    waveView.setData(data);  
}

public void repaint() {  
    super.repaint();  
    if (null != waveView)  
        waveView.repaint();  
}

public void resetSamples() {  
    waveView.resetSamples();  
}

public void setWaveColor(Color foreground) {  
    waveView.setForeground(foreground);  
}

private JPanel createButtonBar() {  
    JPanel retPanel = new JPanel();  
    return retPanel;  
}
retPanel.setLayout(new BoxLayout(retPanel, BoxLayout.X_AXIS));
retPanel.add(Box.createHorizontalGlue());

JButton zoomIn = new JButton("+");
JButton zoomOut = new JButton("-");
zoomIn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        waveView.zoomIn();
    }
});
zoomOut.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        waveView.zoomOut();
    }
});
retPanel.add(zoomIn);
retPanel.add(zoomOut);

JCheckBox connectedBox = new JCheckBox("Connect Sample Points", DEFAULT_CONNECTED);
connectedBox.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        JCheckBox src = (JCheckBox)event.getSource();
        waveView.setConnected(src.isSelected());
    }
});
retPanel.add(connectedBox);

return retPanel;

private WaveViewComponent waveView;
private JScrollBar scrollBar;

C.2.3 Audio Processing Application Data Stream Simulation

This section includes the following source code files:

- CodecWave.java
- SerialWave.java
- SerialWaveOut.java

CodecWave.java

package model;
import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

import data.SerialWave;
import data.SerialWaveOut;

public class CodecWave extends Logic {

    public static CellInterface[] cell_interface = {
        in("mclk",1), // master clock (\~4X sclk)
        in("lrck",1), // left/right clock
        in("sclk",1), // sample clock
        in("sdin",1), // serial data in (from FPGA)
        out("sdout",1), // serial data out (to FPGA)
        param("wavFileNameIn",STRING),
        param("wavFileNameOut",STRING),
    };

    public static final String WAVE_IN_DEFAULT = "stereo_8.wav";
    public static final String WAVE_OUT_DEFAULT = "stereo_8_out.wav";

    public CodecWave(Node parent, Wire mclk, Wire lrck, Wire sclk, Wire sdin, Wire sdout) {
        this(parent,mclk,lrck,sclk,sdin,sdout,
             WAVE_IN_DEFAULT,WAVE_OUT_DEFAULT);
    }

    public CodecWave(Node parent, Wire mclk, Wire lrck, Wire sclk, Wire sdin, Wire sdout,
                      String wavFileNameIn, String wavFileNameOut) {
        super(parent,"CodecWave");
        connect("mclk",mclk);
        this.lrck = connect("lrck",lrck);
        this.sclk = connect("sclk",sclk);
        this.sdin = connect("sdin",sdin);
        this.sdout = connect("sdout",sdout);
        bind("wavFileNameIn",wavFileNameIn);
        bind("wavFileNameOut",wavFileNameOut);

        waveFileIn = new SerialWave(wavFileNameIn);
        waveFileOut = new SerialWaveOut(wavFileNameIn,wavFileNameOut);
    }

    public boolean isDone() {
        return waveFileOut.isDone();
    }
}
public void reset() {
    lastSclk = true;
    lastLrck = false;
    sdoutValue = false;
    left = true;
    bitInd = 19;
    // would be good to reset .WAV file here too!
    sdout.putB(this,sdoutValue);
}

public void clock() {
    if ( sclk.getB(this) ) {
        lastSclk = true;
    } else {
        if ( lastSclk ) {
            updateSamples();
        }
        lastSclk = false;
    }
    sdout.putB(this,sdoutValue);
}

private void updateSamples() {
    boolean lrckVal = lrck.getB(this);
    if ( lrckVal ) {
        if ( !lastLrck ) {
            // switch to left channel
            left = true;
            bitInd = 19;
            //System.err.println("setting next real frame");
            waveFileIn.setNextRealFrame();
            waveFileOut.setNextRealFrame();
        }
    } else if ( lastLrck) {
        // switch to right channel
        left = false;
        bitInd = 19;
    }
    lastLrck = lrckVal;
}

int sdinValue = sdin.get(this);
waveFileOut.setBit(left,bitInd,sdinValue);
sdoutValue = waveFileIn.getBit(left,bitInd--)==1;
}

private Wire lrck;
private Wire sclk;
private Wire sdin;
private Wire sdout;
/** If true the current output and input are for the left
 * channel, otherwise they’re for the right channel */
private boolean left;
SerialWave.java

package data;

/**
 * This file reads in a frame(1 sample from both left and
 * right channels) from a wav file, and then is able to
 * return bits from those channels
 * 
 * @author M. Ryan Byrd & Anthony L. Slade
 * @version 1.0
 */
import javax.sound.sampled.AudioFormat;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.UnsupportedAudioFileException;
import java.io.File;
import java.io.IOException;

public class SerialWave {

    private AudioInputStream myStream;
    /**because WAV files use a "differential" PCM encoding,
     *a running, modded total is used to get the "real" value*/
    private int currRealLeft = 0;
    /**because WAV files use a "differential" PCM encoding,
     *a running, modded total is used to get the "real" value*/
    private int currRealRight = 0;
    //supports 8 or 16 bit wav files
    public static final int SAMPLE16 = 0x10000;
    public static final int SAMPLE8 = 0x100;
    private int sampleSize;
    private byte[] bArray;
    private int[] iArray;

    public SerialWave(String fileName) {

    }
File myFile = new File(fileName);

try {
    myStream = AudioSystem.getAudioInputStream(myFile);
} catch (IOException myException) {
    return;
} catch (UnsupportedAudioFileException myOtherException) {
    return;
}

AudioFormat myFormat = myStream.getFormat();
if (myFormat.getSampleSizeInBits() == 16)
    sampleSize=SAMPLE16;
else
    sampleSize=SAMPLE8;
bArray = new byte[myFormat.getFrameSize()];
iArray = new int[myFormat.getFrameSize()];

public void setNextRealFrame() {

    int numBytes;
    try {
        numBytes = myStream.read(bArray);
    } catch (IOException ioe) {
        ioe.printStackTrace();
        return; // currRealRight and currRealLeft stay the same
    }
    for (int ii = 0; ii < numBytes; ++ii) {
        iArray[ii] = ((int)bArray[ii])&0xFF;
    }

    switch (numBytes) {
        case 1: //MONO 8 Bits
            currRealLeft = (currRealLeft+iArray[0]) % sampleSize;
            currRealRight = currRealLeft;
            break;

        case 2: //MONO 16 bits or STEREO 8 Bits
            if (sampleSize==SAMPLE16) { //MONO 16 Bit
                int diff = iArray[0] | (iArray[1]<<8);
                currRealLeft = (currRealLeft+diff) % sampleSize;
                currRealRight = currRealLeft;
            } else { //STEREO 8 Bit
                currRealLeft = (currRealLeft+iArray[0]) % sampleSize;
                currRealRight = (currRealRight+iArray[1]) % sampleSize;
            }
            break;

        case 4: //STEREO 16 Bit
            int diff = iArray[0] | (iArray[1]<<8);
            int diff2 = iArray[2] | (iArray[3]<<8);
currRealLeft = (currRealLeft+diff) % sampleSize;
currRealRight = (currRealRight+diff2) % sampleSize;
break;

default:  
   // currRealRight and currRealLeft stay the same;
}
}

public int getRealValueLeft() {  return currRealLeft;  }
public int getRealValueRight() {  return currRealRight;  }

public int getBit(boolean left, int bit){
   if (left)
      return ((currRealLeft>>bit)&1);
   else
      return ((currRealRight>>bit)&1);
}

public long getFrameLength() {
   return myStream.getFrameLength();
}
public float getSampleRate() {
   return myStream.getFormat().getSampleRate();
}

}  

SerialWaveOut.java

package data;
/**
 * This file writes out a frame(1 sample from both left and
 * right channels) from to a wav file as processed from input
 * it gets.
 * @author M. Ryan Byrd & Anthony L. Slade
 * @version 1.0
 */

import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import java.io.File;
//import java.io.InputStream;
import java.io.IOException;
//import java.io.OutputStream;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;
import javax.sound.sampled.UnsupportedAudioFileException;
import javax.sound.sampled.AudioFileFormat;
import javax.sound.sampled.AudioFileFormat.Type;
import javax.sound.sampled.AudioFormat;

public class SerialWaveOut {

public class SerialWaveOut {
public static final String DEFAULT_FILE = "stereo_8";

public static final int NUM_ECHO_SAMPLES = 5120;

public static final int NUM_ECHO_SAMPLES = 0;

public static void main(String[] args) {
    SerialWave swIn = new SerialWave(DEFAULT_FILE + "\.wav");
    SerialWaveOut swOut = new SerialWaveOut(DEFAULT_FILE + "\.wav",
                                            DEFAULT_FILE + "\_out\.wav");
    for (int frame = 0; frame < swOut.getFrameLength(); ++frame ) {
        if (0 == frame % 100)
            System.err.print(frame + ",");
        swIn.setNextRealFrame();
        for (int chan = 0; chan < 2; ++chan ) {
            boolean left = (chan == 0);
            for (int bit = 0; bit < 20; ++bit ) {
                swOut.setBit(left, bit, swIn.getBit(left, bit));
            }
        }
        swOut.setNextRealFrame();
    }
}

public SerialWaveOut(String fileNameIn, String fileNameOut) {
    file = new File(fileNameOut);
    File myFileIn = new File(fileNameIn);
    AudioInputStream myStreamIn = null;
    try {
        myStreamIn = AudioSystem.getAudioInputStream(myFileIn);
        fileType = AudioSystem.getAudioFileFormat(myFileIn).getType();
    } catch (IOException myException) {
        return;
    } catch (UnsupportedAudioFileException myOtherException) {
        return;
    }
    AudioFormat myFormatIn = myStreamIn.getFormat();
    AudioFormat myFormatOut = new AudioFormat(myFormatIn.getSampleRate(),
                                              8, // 8-bit samples
                                              2, // 2 channels
                                              false, // unsigned
                                              myFormatIn.isBigEndian());

    streamOut = new PipedReaderStream();
    streamIn = new PipedWriterStream();
    try {
        streamOut.connect(streamIn);
    } catch (IOException ioe) {
ioe.printStackTrace();
}
frameLength = myStreamIn.getFrameLength()
+NUM_ECHO_SAMPLES;
myStreamOut = new AudioInputStream(streamOut,myFormatOut,
frameLength);
bArray = new byte[myFormatOut.getFrameSize()];
iArray = new int[myFormatOut.getFrameSize()];
startWriteThread();
}

public SerialWaveOut(String fileNameOut,
float sampleRate,
long frameLength) {
file = new File(fileNameOut);
fileType = AudioFileFormat.Type.WAVE;
AudioFormat myFormatOut
= new AudioFormat(sampleRate,
8,//8-bit samples
2,//2 channels
false,//unsigned
false);//isBigEndian
streamOut = new PipedInputStream();
streamIn = new PipedOutputStream();
try {
    streamOut.connect(streamIn);
} catch (IOException ioe) {
    ioe.printStackTrace();
}
this.frameLength = frameLength;
myStreamOut = new AudioInputStream(streamOut,myFormatOut,
frameLength);
bArray = new byte[myFormatOut.getFrameSize()];
iArray = new int[myFormatOut.getFrameSize()];
startWriteThread();
}

private void startWriteThread() {
//This is done in a separate thread so that deadlock does
//not result;see API documentation for the Piped*Stream
//classes.
(new Thread() {
    public void run() {
        try {
            AudioSystem.write(myStreamOut,fileType,file);
            close();
        } catch (IOException ioe) {
            
    } catch (IOException ioe) {
public long getFrameLength() {
    return frameLength;
}

public boolean isDone() {
    return done;
}

private void close() {
    try {
        streamIn.flush();
        myStreamOut.close();
        streamIn.close();
        streamOut.close();
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
    done = true;
}

/* debug stuff */
private long cumRealRight = 0;
private long cumRealLeft = 0;
private int maxRealRight = 0;
private int maxRealLeft = 0;
private long numSamples = 0;
/* */

public void setNextRealFrame(){
    /* debug stuff */
    ++numSamples;
    cumRealRight += currRealRight;
    cumRealLeft += currRealLeft;
    if ( currRealRight > maxRealRight )
        maxRealRight = currRealRight;
    if ( currRealLeft > maxRealLeft )
        maxRealLeft = currRealLeft;
    if ((numSamples%1000)==1) {
        System.err.println("Num Samples = "+numSamples);
        System.err.println("Average data value right = "
            +(cumRealRight/numSamples));
        System.err.println("Average data value left = "
            +(cumRealLeft/numSamples));
        System.err.println("Max right = "+maxRealRight
            +"; Max left = "+maxRealLeft);
    }
    /* end debug stuff */
int diffLeft = currRealLeft - lastRealLeft;
int diffRight = currRealRight - lastRealRight;
iArray[0] = diffLeft & 0xFF;
iArray[1] = diffRight & 0xFF;
for (int ind = 0; ind < 2; ++ind) {
    bArray[ind] = (byte)iArray[ind];
}
if ( !done ) {
    try {
        streamIn.write(bArray,0,2);
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}
lastRealLeft = currRealLeft;
lastRealRight = currRealRight;
//System.err.println("Output: Left="+currRealLeft
//+";Right="+currRealRight
//+";diffLeft="+diffLeft
//+";diffRight="+diffRight);
currRealRight = currRealLeft = 0;
}

public void setNextRealFrame(int chLeft, int chRight){
    currRealLeft = chLeft;
currRealRight = chRight;
    setNextRealFrame();
}

public void setBit(boolean left, int bit, int value){
    if ( 1 == value ) {
        if (left)
            currRealLeft |= (1<<bit);
        else
            currRealRight |= (1<<bit);
    }
}

private PipedOutputStream streamIn;
private PipedInputStream streamOut;
private AudioInputStream myStreamOut;
private AudioFileFormat.Type fileType;
private File file;
private long frameLength;
/**because WAV files use a "differential" PCM encoding,
a running, modded total is used to get the "real" value */
private int currRealLeft = 0;
/**because WAV files use a "differential" PCM encoding,
a running, modded total is used to get the "real" value */
private int currRealRight = 0;
private int lastRealLeft = 0;
private int lastRealRight = 0;
private byte[] bArray;
private int[] iArray;
private boolean done;
}

C.3 Example Application: Edit Distance

This section contains the code for the example application in Section 5.2. The
code for the application model (Section C.2.1) view (Section C.2.2) and controller (Sec-
tion C.2.1) is included. In each subsection, the classes are listed in alphabetical order.

C.3.1 Edit Distance Application Model

This section includes the following source code files:

- EDAlphabet.java
- EditCellInner.java
- EditCellInput.java
- EditCellOutput.java
- EditDistance.java
- NextDForNotNullIn.java
- NextDForNullIn.java

EDAlphabet.java

package model;

import byucc.jhdl.base.Cell;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

/** Class that helps support the edit distance circuits. */
* @author Anthony L. Slade */
public class EDAlphabet {
  /** constant for the Thymine base */
  public static final int BASE_T = 0x8; // 0b1000
  /** constant for the Guanine base */
  public static final int BASE_G = 0x4; // 0b0100
}
/** constant for the Cytosine base */
public static final int BASE_C = 0x2; // 0b0010
/** constant for the Adenine base */
public static final int BASE_A = 0x1; // 0b0001
/** constant for a wildcard base */
public static final int BASE_WILDCARD = 0x0;

/** @return the character for the given base value */
public static char getBaseChar(int base) {
    switch (base) {
    case BASE_T:
        return 'T';
    case BASE_G:
        return 'G';
    case BASE_C:
        return 'C';
    case BASE_A:
        return 'A';
    case BASE_WILDCARD:
        default:
            return 'W';
    }
}

/** @return the integer value representation (BASE_T, BASE_G, BASE_C, BASE_A, or BASE_WILDCARD) of the given character */
public static int getBaseValue(char base) {
    switch (base) {
    case 't':
    case 'T':
        return BASE_T;
    case 'g':
    case 'G':
        return BASE_G;
    case 'c':
    case 'C':
        return BASE_C;
    case 'a':
    case 'A':
        return BASE_A;
    default:
        return BASE_WILDCARD;
    }
}

/** @return the integer value representation (BASE_T, BASE_G, BASE_C, BASE_A, or BASE_WILDCARD) of the first character of the given string */
public static int getBaseValue(String base) {
    return getBaseValue(base.charAt(0));
}

/** @return the integer value representation (BASE_T, BASE_G, BASE_C, BASE_A, or BASE_WILDCARD) of the first character of the given string */
public static int getBaseValue(String base) {
    return getBaseValue(base.charAt(0));
}
public static int getBaseValue(String base, int index) {
    return getBaseValue(base.charAt(index));
}

public static Wire charEqualsConstant(Cell parent, Wire input, int base) {
    return charEqualsConstant(parent, input,
        Logic.wire(parent,1, "charInEqConstChar"), base);
}

public static Wire charEqualsConstant(Cell parent, Wire input, Wire output, int base) {
    new byucc.jhdl.Xilinx.Virtex.lut4(parent, "compare"
        +getBaseChar(base), input,
        output, getBaseCompareString(base));
    return output;
}

public static String getBaseCompareString(int base) {
    switch (base) {
    case BASE_T:
        return "0100"; // "0b0000000100000000";
    case BASE_G:
        return "0010"; // "0b0000000000010000";
    case BASE_C:
        return "0004"; // "0b0000000000000100";
    case BASE_A:
        return "0002"; // "0b0000000000000010";
    case BASE_WILDCARD:
        default:
            return "FFFF"; // "0b1111111111111111";
    }
}

EditCellInner.java

package model;
import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;
/** Inner PE of the edit distance circuit. This circuit is partially based on Justin Tripp’s VHDL version of the edit distance circuit for FPGAs. * @author Anthony. L. Slade */

public class EditCellInner extends Logic {

    public static final String PORT_CHAR_IN = "charIn";
    public static final String PORT_A_IN = "aIn";
    public static final String PORT_NULL_IN = "nullIn";
    public static final String PORT_A_OUT = "aOut";
    public static final String PORT_CHAR_OUT = "charOut";
    public static final String PORT_NULL_OUT = "nullOut";
    public static final String PARAM_INIT_A = "initA";
    public static final String PARAM_INIT_C = "initC";
    public static final String PARAM_CONST_CHAR = "constChar";

    public static CellInterface[] cell_interface = {
        in(PORT_CHAR_IN,4),
        in(PORT_A_IN,2),
        in(PORT_NULL_IN,1),
        out(PORT_A_OUT,2),
        out(PORT_CHAR_OUT,4),
        out(PORT_NULL_OUT,1),
        param(PARAM_INIT_A,INTEGER),
        param(PARAM_INIT_C,INTEGER),
        param(PARAM_CONST_CHAR,INTEGER),
    };

    public EditCellInner(Node parent,
        Wire charIn,
        Wire aIn,
        Wire nullIn,
        Wire aOut,
        Wire charOut,
        Wire nullOut,
        int initA,
        int initC,
        int constChar) {
        this(parent,charIn,aIn,nullIn,aOut,charOut,nullOut,
            initA,initC,constChar,"EditDistancePE");
    }

    public EditCellInner(Node parent,
        Wire charIn,
        Wire aIn,
        Wire nullIn,
        Wire aOut,
        Wire charOut,
        Wire nullOut,
        int initA,
        int initC,
        int constChar,
        String name) {

super(parent, name);
bind(PARAM_INIT_A, initA);
bind(PARAM_INIT_C, initC);
bind(PARAM_CONST_CHAR, constChar);
System.err.println(name +":" + EDAlphabet.getBaseChar(constChar));
connect(PORT_CHAR_IN, charIn);
connect(PORT_A_IN, aIn);
connect(PORT_A_OUT, aOut);
connect(PORT_CHAR_OUT, charOut);
connect(PORT_NULL_OUT, nullOut);

if (false) {
    System.err.println("New_EditCellInner: ");
    System.err.println("\tinItA=" + initA);
    System.err.println("\tinItC=" + initC);
    System.err.println("\tconstChar=" + constChar);
}

reg_o(nor(charIn, "nullChar"), nullOut, "nullCharOut");

reg_o(charIn, charOut, "charReg");

ccharInEqConstChar = EDAlphabet.charEqualsConstant(this, charOut, constChar);
Wire a = reg(aIn, "a");
Wire d = aOut;
Wire nextD = wire(2, "nextD");

/*
   if ( nullIn ) { // 3 bits
       if ( charactersEqual ) {
           d = initA;
       } else {
           if ( initC == aIn )
               d = initA + 2;
           else
               d = initA;
       }
   } else { // 7 bits
       if ( charactersEqual ) {
           d = a;
       } else {
           if ( (a+1) == aIn && (a+1) == d )
               d = a + 2;
           else
               d = a;
       }
   }
*/

Wire wireForNotNullIn = wire(2, "wireForNotNullIn");
new NextDForNotNullIn(this, charInEqConstChar, a, aIn, d,

152
wireForNotNullIn); 
Wire initWireForNullIn = wire(2,"initWireForNullIn"); 
new NextDForNullIn(this,charInEqConstChar,aIn, 
initWireForNullIn,initA,initC); 
mux_o(wireForNotNullIn, 
initWireForNullIn, 
nullIn, 
nextD, 
"selectNextD"); 
reg_o(nextD,d,"regD"); 
}

public Wire getCharactersEqualWire() { 
return charInEqConstChar; 
}

protected Wire charInEqConstChar;

/** Front end to the edit distance circuit. This design is 
* based on Justin Tripp’s implementation of the edit 
* distance algorithm in VHDL. 
* @author Anthony L. Slade */
public class EditCellInput extends Logic {

public static final String PORT_CHAR_IN = "charIn";
public static final String PORT_NULL_IN = "nullIn";
public static final String PORT_A_OUT = "aOut";
public static final String PORT_CHAR_OUT = "charOut";
public static final String PARAM_CONST_CHAR = "constChar";

public static CellInterface[] cell_interface = {
in(PORT_CHAR_IN,4),
in(PORT_NULL_IN,1),
out(PORT_A_OUT,2),
out(PORT_CHAR_OUT,4),
param(PARAM_CONST_CHAR,INTEGER),
};

public EditCellInput(Node parent, 
Wire charIn, 
Wire nullIn, 
Wire aOut, 
Wire charOut, 
int constChar) {
```java
this(parent,charIn,nullIn,aOut,charOut,constChar,
    "EditDistanceInput");
}

public EditCellInput(Node parent,
    Wire charIn,
    Wire nullIn,
    Wire aOut,
    Wire charOut,
    int constChar,
    String name) {
    super(parent,name);
    bind(PARAM_CONST_CHAR,constChar);
    System.err.println(name":"+
        EDAlphabet.getBaseChar(constChar));
    connect(PORT_CHAR_IN,charIn);
    connect(PORT_NULL_IN,nullIn);
    connect(PORT_A_OUT,aOut);
    connect(PORT_CHAR_OUT,charOut);

    Wire d = aOut;
    prevCountWire = wire(2,"a");
    Wire a = prevCountWire;
    Wire b = wire(2,"b");

    reg_o(charIn,charOut,"charPass");
    regr_o(b.gw(1),nullIn,a.gw(1),"regA1");
    regs_o(b.gw(0),nullIn,a.gw(0),"regA0");
    Wire bPlusOne = add(b,constant(2,1),"bPlusOne");
    regs_o(bPlusOne.gw(1),nullIn,b.gw(1),"regB1");
    regr_o(bPlusOne.gw(0),nullIn,b.gw(0),"regB0");
    //regr_o( Wire d, Wire r,Wire q,String name)
    //regs_o( Wire d, Wire s,Wire q,String name)
    //regse_o(Wire d,Wire ce,Wire s,Wire q,String name)
    //regpe_o(Wire d,Wire ce, Wire q,String name)
    Wire bEqD = wire(1,"bEqD");
    new byucc.jhdl.Xilinx.Virtex.lut4(this,
        "bEqD",
        concat(b,d,"concatBD"),
        bEqD,
        /* 0b 1000 0100 0010 0001 */ "8421");

    Wire nextD = concat(wire(1,"nextD1"),a.gw(0),"nextD");
    charInEqConstChar
        = EDAlphabet.charEqualsConstant(this, charOut,
            constChar);
    xor_o(a.gw(1),and(bEqD,not(charInEqConstChar,
        "notCharInEqConstChar"),
        "addTwo"),
        nextD.gw(1),"nextD1");
    new byucc.jhdl.Xilinx.Virtex.fdrs(this,"regD1",
        nextD.gw(1),
```
public Wire getCharactersEqualWire() {
    return charInEqConstChar;
}
public Wire getPreviousCountWire() {
    return prevCountWire;
}
protected Wire charInEqConstChar;
protected Wire prevCountWire;

EditCellOutput.java

package model;
import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;
import byucc.jhdl.Xilinx.Virtex.lut4;

/** Back end to the edit distance circuit. This design is
 * based on Justin Tripp's implementation of the edit
 * distance algorithm in VHDL.
 * @author Anthony L. Slade */
public class EditCellOutput extends Logic {

    public static final String PORT_CHAR_IN = "charIn";
    public static final String PORT_A_IN = "aIn";
    public static final String PORT_NULL_OUT = "nullOut";
    public static final String PORT_DISTANCE = "distance";
    public static final String PARAM_COUNTSIZE = "COUNTSIZE";
    public static final String PARAM_LENGTH = "length";

    public static CellInterface[] cell_interface = {
        in(PORT_CHAR_IN,4),
        in(PORT_A_IN,2),
        out(PORT_NULL_OUT,1),
        out(PORT_DISTANCE,PARAM_COUNTSIZE),
        param(PARAM_COUNTSIZE,INTEGER),
        param(PARAM_LENGTH,INTEGER),// length of target string
    };

    public EditCellOutput(Node parent,
        Wire charIn,
        Wire aIn,
        Wire nullOut,
        Wire nullIn,
        d.gw(1));
    regr_o(nextD.gw(0),nullIn,d.gw(0),"regD0");
    //public fdrs(Node parent,Wire d,Wire r,Wire s,Wire q)
}
public EditCellOutput(Node parent, Wire charIn, Wire aIn, Wire nullOut, Wire distance, int length, String name) {
    super(parent, name);
    int distWidth = distance.getWidth();
    bind(PARAM_COUNTSIZE, distWidth);
    bind(PARAM_LENGTH, length);
    connect(PORT_CHAR_IN, charIn);
    connect(PORT_A_IN, aIn);
    connect(PORT_NULL_OUT, nullOut);
    connect(PORT_DISTANCE, distance);

    Wire nullChar = wire(1, "nullChar");
    reg_o(nor_o(charIn, nullChar, "nullChar"), nullOut, "nullChar");

    Wire nextCount = wire(distWidth, "nextCount");

    // Here's the scheme for nextCount:
    // ----------------------------------------
    // | aIn | currDistance[1:0] | nextCount |
    // ----------------------------------------
    // | 0  | 0 | distance+0 |
    // 0  | 1 | distance-1 |
    // 0  | 2 | distance-2 |
    // 0  | 3 | distance+1 | *
    // ----------------------------------------
    // | 1  | 0 | distance+1 |
    // 1  | 1 | distance+0 |
    // 1  | 2 | distance-1 |
    // 1  | 3 | distance-2 |
    // ----------------------------------------
    // | 2  | 0 | distance+2 |
    // 2  | 1 | distance+1 |
    // 2  | 2 | distance+0 |
    // 2  | 3 | distance-1 |
    // ----------------------------------------
    // | 3  | 0 | distance-1 | *
    // 3  | 1 | distance+2 |
    // 3  | 2 | distance+1 |
    // 3  | 3 | distance+0 |
    // ----------------------------------------

    // Set up the adder/subtractor
    // This adder/subtractor should be optimized to be able
// to add or subtract either 1 or 2. This table shows  
// how it works:
// | subtract | cin | one | result |
// ---------------------------------
// | 0 | 0 | 0 | +0 |
// | 0 | 0 | 1 | +1 |
// | 0 | 1 | 0 | +1 |
// | 0 | 1 | 1 | +2 |
// | 1 | 0 | 0 | -2 |
// | 1 | 0 | 1 | -1 |
// | 1 | 1 | 0 | -1 |
// | 1 | 1 | 1 | -0 |

Wire subtract = wire(1, "subtract");
Wire cin = wire(1, "cin");
Wire negate = subtract;
Wire oneNeg = negate;
for ( int negi = 2; negi < distWidth; ++negi ) {
    oneNeg = concat(negate, oneNeg);
}

Wire one = wire(1, "one");
// negative or positive one (can also be used for
// negative 2)
Wire oneNegPos = concat(oneNeg, one, "oneNegPos");
add_o(distance, oneNegPos, cin, nextCount, "addSub");

// derive the control signals for the adder/subtractor  
// from ain and the current count state.
Wire catAInLowCount = concat(aIn, distance.range(1,0),  
    "catAInLowCount");

new lut4(this,  
    "subtract",  
    catAInLowCount,  
    subtract,  
    "9CE7"); // 0b 1001 1100 1110 0111

new lut4(this,  
    "cin",  
    catAInLowCount,  
    cin,  
    "A521"); // 0b 1010 0101 0010 0001

new lut4(this,  
    "one",  
    catAInLowCount,  
    one,  
    "FF7B"); // 0b 1111 1111 0111 1011

Wire resetOrNextCount = wire(distWidth, "muxOutCount");
Wire constantInitLength = constant(distWidth, length,  
    "initLength");
mux_o(nextCount, constantInitLength, nullOut,  
    resetOrNextCount, "selCount");
reg_o(resetOrNextCount, distance, "countReg");
package model;

import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;

import byucc.jhdl.Logic.Logic;

public class EditDistance extends Logic {
    public static final String PORT_CHAR_IN = "charIn";
    public static final String PORT_DISTANCE = "distance";
    public static final String PARAM_TARGET = "target";
    public static final String PARAM_WIDTH = "WIDTH";

    public static CellInterface[] cell_interface = {
        in(PORT_CHAR_IN, 4),
        out(PORT_DISTANCE, PARAM_WIDTH),
        param(PARAM_WIDTH, INTEGER),
        param(PARAM_TARGET, STRING),
    };

    public EditDistance( Node parent,
        Wire charIn,
        Wire distance,
        String targetStr ) {
        super(parent);
        int width = distance.getWidth();
        bind(PARAM_TARGET, targetStr);
        bind(PARAM_WIDTH, width);
        connect(PORT_CHAR_IN, charIn);
        connect(PORT_DISTANCE, distance);
        int numCells = targetStr.length();
    
        Wire nullIn = wire(numCells, "nullIn");
        Wire[] aOut = new Wire[numCells];
        Wire[] charOut = new Wire[numCells];
        int[] initA = new int[numCells];
        int[] initC = new int[numCells];
        int[] constChar = new int[numCells];
        initConstantArrays(initA, initC, constChar, targetStr, numCells);
        aOut[0] = wire(2, "aOut0");
        charOut[0] = wire(4, "charOut0");

        new EditCellInput(this,
            charIn,
            nullIn.gw(0),
            aOut[0],
            charOut[0],
            EDAlphabet.getBaseValue(targetStr, 0));
    }
}
for ( int inner = 1; inner < numCells; ++inner ) {
    aOut[inner] = wire(2,"aOut"+inner);
    charOut[inner] = wire(4,"charOut"+inner);
    new EditCellInner(this,
        charOut[inner-1],
        aOut[inner-1],
        nullIn.gw(inner),
        aOut[inner],
        charOut[inner],
        nullIn.gw(inner-1),
        initA[inner%numCells],
        initC[inner%numCells],
        constChar[inner],
        "EditCellInner"+(inner-1));
}
new EditCellOutput(this,
    charOut[charOut.length-1],
    aOut[aOut.length-1],
    nullIn.gw(nullIn.getWidth()-1),
    distance,
    numCells);//length

private void initConstantArrays(int[] initA, int[] initC,
    int[] constChar,
    String target,
    int numCells) {
    for ( int ind = 0; ind < numCells; ++ind ) {
        initA[ind] = ind&0x3;
        initC[ind] = ((ind+1)%4)&0x3;
        constChar[ind] = EDAlphabet.getBaseValue(target,ind);
    }
}

NextDForNotNullIn.java

package model;

import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

/** Design to build hierarchy for the EditCellInner design
* @author Anthony L. Slade */
public class NextDForNotNullIn extends Logic {
    public static CellInterface[] cell_interface = {
        in("charInEqConstChar",1),
        in("a",2),
        in("aIn",2),
        in("d",2),
        out("nextDForNotNullInWire",2),
public NextDForNotNullIn(Node parent, Wire charInEqConstChar, Wire a, Wire aIn, Wire d, Wire output) {

super(parent,"NextDForNotNullIn");

/*
if (!nullIn) {
  if ( charactersEqual ) {
    d = a;
  } else {
    if ( (a+1)==aIn && (a+1)==d )
      d = a+2;
    else
      d = a;
  }
}
*/

connect("charInEqConstChar",charInEqConstChar);
connect("a",a);
connect("aIn",aIn);
connect("d",d);
connect("nextDForNotNullInWire",output);

// a.gw(0) will just pass through no matter what
buf_o(a.gw(0),output.gw(0));
xor_o(a.gw(1),
    and(not(charInEqConstChar,"charsNotEqual"),
    twoBitsPlusEqual(a,aIn,"aPlusEqAIn"),
    twoBitsPlusEqual(a,d,"aPlusEqD"),
    "decideAdd"),
    output.gw(1));
}

/** @return output of a lut that compares the two given*
* wires. If the wires are equal (after adding 1 to the*
* first wire), the output is asserted (high). */
private Wire twoBitsPlusEqual(Wire wireA, Wire wireB, String name) {
  Wire output = wire(1,name);
  String lutInit = "1842"; // 0b 0001 1000 0100 0010
  new byucc.jhdl.Xilinx.Virtex.lut4(this,
      name,
      concat(wireA,wireB,
          "concat"+name),
      output,
      lutInit);

  return output;
}
package model;

import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

/** Design to build hierarchy for the EditCellInner design
 * @author Anthony L. Slade */
public class NextDForNullIn extends Logic {
    public static CellInterface[] cell_interface = {
        in("charInEqConstChar",1),
        in("aIn",2),
        out("nextDForNullInWire",2),
    }
    public NextDForNullIn(Node parent, Wire charInEqConstChar, Wire aIn, Wire output, int initA, int initC) {
        super(parent,"NextDForNullIn");
        if ( nullIn ) {
            if ( charactersEqual ) {
                d = initA;
            } else {
                if ( initC==aIn )
                    d = initA+2;
                else
                    d = initA;
            }
        }
        connect("charInEqConstChar",charInEqConstChar);
        connect("aIn",aIn);
        connect("nextDForNullInWire",output);
        int init0=0, init1=0;
        int initA0 = initA & 0x1;
        int initA1 = (initA >> 1) & 0x1;
        for ( int initInd = 0; initInd < 8; ++initInd ) {
            init0 |= (initA0 << initInd);
            init1 |= (initA1 << initInd);
        }
        // The output will always just be initA unless
        // aIn==initC; in that case, the output will be initA+2,
        // so we just add 1 to bit[1] of the output for that
        // initC value; that will just be this xor:
        init1 ^= (1 << (initC&0x3));
    }
}
String initStr0 = Integer.toHexString(init0);
String initStr1 = Integer.toHexString(init1);

Wire lutInput = concat(charInEqConstChar,aIn,"chEqAInA");

new byucc.jhdl.Xilinx.Virtex.lut3(this,
    "initWireForNullIn0",
    lutInput,
    output.gw(0),
    initStr0);
new byucc.jhdl.Xilinx.Virtex.lut3(this,
    "initWireForNullIn1",
    lutInput,
    output.gw(1),
    initStr1);
}

C.3.2 Edit Distance Application View

This section includes the following source code files:

- ComponentArrow.java
- ComponentInnerPE.java
- ComponentInputPE.java
- ComponentOutput.java
- ComponentPE.java
- EDFrame.java
- EDPanel.java

ComponentArrow.java

package view;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import javax.swing.JComponent;

public class ComponentArrow extends JComponent {
    public static final Color DEFAULT_COLOR = Color.red;
    
    public ComponentArrow() {
        super();
    }
    
    public void paint(Graphics g) {
        g.setColor(DEFAULT_COLOR);
        g.fillRect(0, 0, getWidth(), getHeight());
    }
}

EDFrame.java

package view;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import javax.swing.JComponent;

public class EDFrame extends JComponent {
    public static final Color DEFAULT_COLOR = Color.red;
    
    public EDFrame() {
        super();
    }
    
    public void paint(Graphics g) {
        g.setColor(DEFAULT_COLOR);
        g.fillRect(0, 0, getWidth(), getHeight());
    }
}
public static final int PREFERRED_HEIGHT = 20;
public static final int PREFERRED_WIDTH = 40;

public ComponentArrow() {
    this(DEFAULT_COLOR);
}

public ComponentArrow(Color color) {
    arrowColor = color;
    setPreferredSize(new Dimension(PREFERRED_WIDTH, PREFERRED_HEIGHT));
}

public void paint(Graphics gr) {
    int height = getHeight();
    int midHeight = height/2;
    int iWidth = getWidth();
    double dWidth = (double)iWidth;
    double dSeventy = .7*dWidth;
    int iSeventy = (int)dSeventy;
    gr.setColor(arrowColor);
    int[] xpoints = new int[9];
    int[] ypoints = new int[9];
    xpoints[0] = 0; ypoints[0] = midHeight-2;
    xpoints[1] = iSeventy; ypoints[1] = midHeight-2;
    xpoints[7] = iSeventy; ypoints[7] = midHeight+2;
    xpoints[8] = 0; ypoints[8] = midHeight+2;
    gr.fillPolygon(xpoints,ypoints,9);
}

private Color arrowColor;

ComponentInnerPE.java

package view;

import javax.swing.JPanel;
import byucc.jhdl.base.Browser;
import byucc.jhdl.base.Cell;
import byucc.jhdl.base.SimulatorCallback;
import byucc.jhdl.base.Wire;
import model.EditCellInner;

/** Graphical representation of the state of an EditCellInner
 * @author Anthony L. Slade */
public class ComponentInnerPE extends ComponentPE {

    public ComponentInnerPE(EditCellInner cell) {

}
Wire compCharWire;
Wire compEqualWire;
Wire prevCountWire;
Wire countWire;

compCharWire = cell.getAttachedWire(cell.PORT_CHAR_OUT);
compEqualWire = cell.getCharactersEqualWire();
prevCountWire = cell.getPreviousCountWire();
countWire = cell.getAttachedWire(cell.PORT_A_OUT);

buildPanel(cell,
    Integer.parseInt(cell.getArgument(cell.PARAM_CONST_CHAR).toString()),
    compCharWire, compEqualWire,
    prevCountWire, countWire);

ComponentInputPE.java

package view;

import javax.swing.JPanel;
import byucc.jhdl.base.Browser;
import byucc.jhdl.base.Cell;
import byucc.jhdl.base.SimulatorCallback;
import byucc.jhdl.base.Wire;
import model.EditCellInput;

/** Graphical representation of the state of an EditCellInner *
   * @author Anthony L. Slade */
public class ComponentInputPE extends ComponentPE {

    public ComponentInputPE(EditCellInput cell) {
        Wire compCharWire;
        Wire compEqualWire;
        Wire prevCountWire;
        Wire countWire;

        compCharWire = cell.getAttachedWire(cell.PORT_CHAR_OUT);
        compEqualWire = cell.getCharactersEqualWire();
        prevCountWire = cell.getPreviousCountWire();
        countWire = cell.getAttachedWire(cell.PORT_A_OUT);

        buildPanel(cell,
            Integer.parseInt(cell.getArgument(cell.PARAM_CONST_CHAR).toString()),
            compCharWire, compEqualWire,
            prevCountWire, countWire);
    }
}
package view;

import java.awt.Container;
import java.awt.GridLayout;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import byucc.jhdl.base.Browser;
import byucc.jhdl.base.SimulatorCallback;
import byucc.jhdl.base.Wire;
import byucc.jhdl.base.WireValueException;
import model.EditCellOutput;

/** Graphical representation of the state of an EditCellPE
 * @author Anthony L. Slade */
public class ComponentOutput extends JPanel implements Browser, SimulatorCallback {

    public static final String LABEL_PREFIX_GRAY = "<html><font color="gray"><font size="+1">";
    public static final String LABEL_PREFIX_BLUE = "<html><font color="blue"><font size="+1">";
    public static final String LABEL_SUFFIX = "</font></font></html>";

    public ComponentOutput(EditCellOutput cell) {
        setLayout(new GridLayout(4, 2)); // 4 rows, 2 columns
        // row 0
        add(new Container());
        JLabel editTextLabel = new JLabel("<html><font color="blue"><font size="+0">Edit</font></font></html>");
        editTextLabel.setVerticalAlignment(JLabel.BOTTOM);
        add(editTextLabel);
        // row 1
        add(new Container());
        JLabel distanceTextLabel = new JLabel(LABEL_PREFIX_GRAY + "0" + LABEL_SUFFIX, JLabel.CENTER);
        add(distanceLabel);
        // row 2
        add(new Container());
        JLabel distanceTextLabel = new JLabel("<html><font color="blue"><font size="+0">Distance:</font></font></html>");
        distanceTextLabel.setVerticalAlignment(JLabel.TOP);
        add(distanceTextLabel);
        // row 3
        add(new Container());
        distanceLabel = new JLabel(LABEL_PREFIX_GRAY + "0" + LABEL_SUFFIX, JLabel.CENTER);
        add(distanceLabel);
    }
}
prevCountTextField = new JTextField("0",1);
add(prevCountTextField);
prevCountTextField.setEditable(false);
prevCountTextField.setHorizontalAlignment(JTextField.CENTER);
add(new Container());

55
cell.getSystem().addSimulatorCallback(this);
distanceWire = cell.getAttachedWire(cell.PORT_DISTANCE);
prevCountWire = cell.getAttachedWire(cell.PORT_A_IN);
nullOutWire = cell.getAttachedWire(cell.PORT_NULL_OUT);
lastNullOut = 1;
}

public void simulatorUpdate(int cycle, int step) {
    return; // do nothing!
}

public void simulatorRefresh(int cycle, int step) {
    int distanceInt = 0;
    int prevCountInt = 0;
    int nullOutInt = 0;
    String labelPrefix = LABEL_PREFIX_GRAY;

    try {
        distanceInt = distanceWire.get(this);
        prevCountInt = prevCountWire.get(this);
        nullOutInt = nullOutWire.get(this);

        if (lastNullOut == 0 && nullOutInt == 1)
            labelPrefix = LABEL_PREFIX_BLUE;

    } catch (WireValueException wve) {} 

    distanceLabel.setText(labelPrefix+distanceInt
    +LABEL_SUFFIX);
    prevCountTextField.setText(""+prevCountInt);
}

public void simulatorReset() {
simulatorRefresh(0,0);
}

private Wire distanceWire;
private Wire prevCountWire;
private Wire nullOutWire;

private int lastNullOut;

private JLabel distanceLabel;
private JTextField prevCountTextField;

ComponentPE.java

package view;

import java.awt.Container;
import java.awt.GridLayout;
import javax.swing.Box;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import byucc.jhdl.base.Browser;
import byucc.jhdl.base.Cell;
import byucc.jhdl.base.SimulatorCallback;
import byucc.jhdl.base.Wire;
import byucc.jhdl.base.WireValueException;
import model.EDAlphabet;
/** Graphical representation of the state of an EditCellPE
 * @author Anthony L. Slade */
public abstract class ComponentPE extends JPanel implements Browser, SimulatorCallback {

public static final String LABEL_EQUAL = "<html><center><font color="green"><font size="+1"">=</font></font></center></html>";
public static final String LABEL_NOT_EQUAL = "<html><center><font color="red"><font size="+1">!=</font></font></center><html>";

protected void buildPanel(Cell peCell, int thisChar, Wire compCharWire, Wire compEqualWire, Wire prevCountWire, Wire countWire) {
    peCell.getSystem().addSimulatorCallback(this);
    this.compCharWire = compCharWire;
    this.compEqualWire = compEqualWire;
    this.prevCountWire = prevCountWire;
    this.countWire = countWire;
    buildPanel(thisChar);
}

public void simulatorUpdate(int cycle, int step) {
    return; // do nothing!
}

public void simulatorRefresh(int cycle, int step) {
    int compCharInt=0;
    int compEqualInt=0;
    int prevCountInt=0;
    int countInt=0;
    try {
        compCharInt = compCharWire.get(this);
        compEqualInt = compEqualWire.get(this);
        prevCountInt = prevCountWire.get(this);
        countInt = countWire.get(this);
    } catch (WireValueException wve) {
    }
    compCharTextField.setText(""
}
package view;

import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.event.ActionEvent;

public class EDFrame extends JFrame {

    private Wire compCharWire;
    private Wire compEqualWire;
    private Wire prevCountWire;
    private Wire countWire;

    private JLabel equalityLabel;
    private JTextField compCharTextField;
    private JTextField prevCountTextField;

    public EDFrame() {
        super("ED Frame");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new GridLayout(4, 2));
        // row 0
        JLabel charLabel = new JLabel("" + EDAlphabet.getBaseChar(thisChar), JLabel.CENTER);
        add(charLabel);
        // row 1
        JLabel equalityLabel = new JLabel(LABEL_NOT_EQUAL, JLabel.CENTER);
        add(equalityLabel);
        // row 2
        JTextField compCharTextField = new JTextField("_", 1);
        compCharTextField.setEditable(false);
        add(compCharTextField);
        // row 3
        JTextField prevCountTextField = new JTextField("0", 1);
        add(prevCountTextField);
    }

    public void simulatorReset() {
        simulatorRefresh(0, 0);
    }

    private void buildPanel(int thisChar) {
        setLayout(new GridLayout(4, 2)); // 4 rows, 2 columns
        // row 0
        JLabel charLabel = new JLabel("" + EDAlphabet.getBaseChar(thisChar), JLabel.CENTER);
        add(charLabel);
        // row 1
        JLabel equalityLabel = new JLabel(LABEL_NOT_EQUAL, JLabel.CENTER);
        add(equalityLabel);
        // row 2
        JTextField compCharTextField = new JTextField("_", 1);
        compCharTextField.setEditable(false);
        add(compCharTextField);
        // row 3
        JTextField prevCountTextField = new JTextField("0", 1);
        add(prevCountTextField);
    }

    private EDAlphabet alphabet;
    private Wire inputWire;
    private Wire outputWire;

    public void simulatorReset() {
        simulatorRefresh(0, 0);
    }
}

EDFrame.java
import java.awt.event.KeyEvent;
import javax.swing.Box;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.KeyStroke;
import byucc.jhdl.apps.Broker.Broker;
import byucc.jhdl.apps.Broker.GUIListener;
import byucc.jhdl.apps.Broker.GUIListenerFrame;
import byucc.jhdl.apps.Viewers.ViewerFrame;
import byucc.jhdl.apps.Viewers.SimControl.SimControlPanel;
import byucc.jhdl.apps.util.MenuFactory;
import byucc.jhdl.base.HWSystem;
import byucc.jhdl.util.cli.CLInterpreter;

/** Frame that contains an EDPanel, a SimControlPanel and a
 * basic menu to give a custom visualization of the edit
 * distance circuit.
 * @author Anthony L. Slade*/
public class EDFrame extends ViewerFrame {

    public static final String ACTION_ABOUT_JHDL = "About JHDL";

    public EDFrame(HWSystem system, CLInterpreter interp) {
        this.interp = interp;
        contentPane = new JPanel(new BorderLayout());
        // edpanel = new EDPanel(design,sourceString,interp);
        edpanel = new EDPanel(system,interp);
        contentPane.add(edpanel,BorderLayout.NORTH);
        SimControlPanel scPanel =
            new SimControlPanel(true,interp);
        scPanel.addSimControlActionListener(
            new GUIListener(interp));
        contentPane.add(scPanel,BorderLayout.SOUTH);

        buildAndShowFrame();
    }

    protected Container buildContentPane() {
        return contentPane;
    }

    protected JMenuBar buildMenuBar() {
        JMenuBar menuBar = new JMenuBar();

        // Build the menus
        fileMenu = buildFileMenu();
        jhdlMenu = buildJHDLMenu();
        menuBar.add(fileMenu);
        menuBar.add(Box.createHorizontalGlue());
        menuBar.add(jhdlMenu);
        return menuBar;
    }

    // Other methods...
return menuBar;
}

public JMenu buildFileMenu() {
JMenu menu = new JMenu("File");
menu.setMnemonic(KeyEvent.VK_F);

menu.add(BuilderFactory.newMenuItem(ACTION_CLOSE,
        KeyEvent.VK_C,
        KeyStroke.getKeyStroke(KeyEvent.VK_W,
        KeyEvent.CTRL_MASK),
        this));
menu.addSeparator();
menu.add(BuilderFactory.newMenuItem(ACTION_EXIT,
        KeyEvent.VK_X,
        KeyStroke.getKeyStroke(KeyEvent.VK_X,
        KeyEvent.CTRL_MASK),
        this));
return menu;
}

public JMenu buildJHDLMenu() {
JMenu menu = new JMenu("JHDL");
menu.setMnemonic(KeyEvent.VK_J);

menu.add(BuilderFactory.newMenuItem(ACTION_ABOUT_JHDL,
        KeyEvent.VK_A,
        KeyStroke.getKeyStroke(KeyEvent.VK_A,
        KeyEvent.CTRL_MASK),
        this));
return menu;
}

public void actionPerformed(ActionEvent e) {
    String s = e.getActionCommand();
    if (s.equals(ACTION_EXIT)) {
        if (!showConfirmDialogOnExit ||
            JOptionPane.showConfirmDialog(
                this,
                "Do you really want to quit?",
                "Exit Confirmation",
                JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION)
            System.exit(0);
    } else if (s.equals(ACTION_CLOSE)) {
        if (!showConfirmDialogOnClose ||
            JOptionPane.showConfirmDialog(
                this,
                "Do you really want to close this window?",
                "Close Confirmation",
                JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION)
            dispose();
    }
}
else if (s.equals(ACTION_ABOUT_JHDL)) {
    interp.parseCommand("about");
}

// public Broker getBroker() { return edpanel.broker; } 

private JMenuItem fileMenu, jhdlMenu;
private Container contentPane;
private CLInterpreter interp;
private EDPanel edpanel;

EDPanel.java

package view;

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Enumeration;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import byucc.jhdl.base.HWSystem;
import byucc.jhdl.util.cli.CLInterpreter;
import model.EditCellInner;
import model.EditCellInput;
import model.EditCellOutput;
import model.EditDistance;
import controller.EDCLICommand;

public class EDPanel extends JPanel {

    public static final String DEFAULT_SOURCE = "tagct";
    public static final String NAME_CIRCUIT = "tbEditDistance/EditDistance";

    public EDPanel(HWSystem system,
        CLInterpreter interpreter) {
        EditDistance cell
            = (EditDistance)system.findNamed(NAME_CIRCUIT);
        String targetString
            = cell.getArgument(cell.PARAM_TARGET).toString();
        int numChars = targetString.length();
        this.interp = interpreter;
        // broker = new Broker(cell.getSystem(),interpreter);
ComponentPE[] peViews = new ComponentPE[numChars];
ComponentOutput outputView = null;
Enumeration children = cell.getChildrenEnumeration();
while(children.hasMoreElements()) {
    Object child = children.nextElement();
    if ( child instanceof EditCellInput ) {
        peViews[0] = new ComponentInputPE((EditCellInput)child);
    } else if ( child instanceof EditCellInner ) {
        int index = getIndexOfChild((EditCellInner)child);
        peViews[index] = new ComponentInnerPE((EditCellInner)child);
    } else if ( child instanceof EditCellOutput ) {
        outputView = new ComponentOutput((EditCellOutput)child);
    } // else if ( some other child ) {}
}
setLayout(new BorderLayout());

JLabel targetLabel = new JLabel("Target:\u2013"+targetString);
sourceTextField = new JTextField(DEFAULT_SOURCE,20);
sourceTextField.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent event) {
        interp.parseCommand(EDCLICommand.COMMAND_SRC+"\n"+sourceTextField.getText());
    }
});

JPanel sourcePanel = new JPanel(new BorderLayout());
sourcePanel.add(new JLabel("_Source:"), BorderLayout.WEST);
sourcePanel.add(sourceTextField,BorderLayout.CENTER);
JPanel targetSourcePanel = new JPanel(new BorderLayout());
targetSourcePanel.add(targetLabel,BorderLayout.WEST);
targetSourcePanel.add(sourcePanel,BorderLayout.CENTER);
add(targetSourcePanel,BorderLayout.NORTH);

Box charBox = new Box(BoxLayout.X_AXIS);
for ( int ind = 0; ind < numChars; ++ind ) {
    charBox.add(peViews[ind]);
}
charBox.add(outputView);
add(charBox,BorderLayout.CENTER);

private int getIndexOfChild(EditCellInner child) {
    String name = child.getUserName();
    int indName = name.lastIndexOf("EditCellInner");
    String indexString = name.substring(indName+13);
    return Integer.parseInt(indexString) + 1;
}

private JTextField sourceTextField;
CLInterpreter interp;
C.3.3 Edit Distance Application Controller

This section includes the following source code files:

- **EDBroker.java**
- **EDCLICommand.java**
- **tbEditDistance.java**

**EDBroker.java**

```java
package controller;

import byucc.jhdl.apps.Broker.Broker;
import byucc.jhdl.base.HWSysstem;
import byucc.jhdl.util.cli.CLInterpreter;
import view.EDFrame;

public class EDBroker extends Broker {
    public EDBroker(HWSystem system, CLInterpreter interp) {
        super(system, interp);
        new EDFrame(system, interp);
    }

    protected void registerCLICommands(HWSystem system,
            CLInterpreter interp) {
        super.registerCLICommands(system, interp);
        EDCLICommand edc = new EDCLICommand(interp);
    }
}
```

**EDCLICommand.java**

```java
package controller;

import byucc.jhdl.apps.Stimulator.Stimulator;
import byucc.jhdl.util.cli.CLICommand;
import byucc.jhdl.util.cli.CLInterpreter;
import byucc.jhdl.util.cli.CLIInvalidUsageException;
import model.EDAlphabet;

/** Command object to put a string to the charIn wire
 * @author Anthony L. Slade */
```
public class EDCLICommand implements CLICommand {
    public static final String COMMAND_SRC = "src";
    static final String COMMAND_PUT = Stimulator.COMMAND_PUT + " charIn ";
    static final String CLITYPE = "EditDistance";
    /* Constructor */
    EDCLICommand( CLIInterpreter interp ) {
        Stimulator.registerCLICommands(interp);
        interp.registerCommand( COMMAND_SRC, this );
    }
    public Object execute( CLIInterpreter parent, String args[] )
        throws CLIInvalidUsageException {
            String cmdName = args[0];
            // PUT Command
            if ( cmdName.equals(COMMAND_SRC) ) {
                if ( args.length != 2 )
                    throw new CLIInvalidUsageException( "May only specify a single parameter: the source string" );
                String sourceString = args[1];
                String scheduleString = getSourceSchedule(sourceString);
                //parent.println("Putting the schedule " + scheduleString + " to the source string input");
                parent.parseCommandNoHistory(COMMAND.PUT+scheduleString);
            }
            return null;
        } // end method execute
    public String getHelpText(String cmdName) {
        // PUT Command
        if (cmdName.equals(COMMAND_SRC))
            return "Set the sequence of input characters to the given string.";
        return null;
    }
    public String getHelpType(String cmdName) {
        return CLITYPE;
    }
    public String getUsageText(String cmdName) {
        // PUT Command
        if (cmdName.equals(COMMAND_SRC))
            return "<source_string>";
        return null;
    }
    private static String getSourceSchedule(String source) {
        StringBuffer buff = new StringBuffer();
        buff.append('0');
        buff.append(' ');
        buff.append('0');
        buff.append(' ');
        for (int ind = 0; ind < source.length(); ++ind) {
            buff.append(EDAlphabet.getBaseValue(source, ind));
            buff.append(' ');
        }
        buff.append('0');
    }
}
C.4 Example Application: Guessing Game

This section contains the code for the example application in Section 5.3. The code for the application model (Section C.2.1) view (Section C.2.2) and controller (Section C.2.1) is included. In each subsection, the classes are listed in alphabetical order.

C.4.1 Guessing Game Application Model

This section includes the following source code files:
package model;

import byucc.jhdl.base.Browser;
import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

public class AttemptBlock extends Logic
implements PGCommon/*, UserDefinedSchematic*/,
Browser {
    public static CellInterface[] cell_interface =
    {
        in ("valueIn0",3),
        in ("valueIn1",3),
        in ("valueIn2",3),
        in ("valueIn3",3),
        in ("setValue",1),
        in ("guessIn0",3),
        in ("guessIn1",3),
        in ("guessIn2",3),
        in ("guessIn3",3),
        in ("setGuess",1),
        out("valueOut0",3),
        out("valueOut1",3),
        out("valueOut2",3),
        out("valueOut3",3),
        out("guessOut0",3),
        out("guessOut1",3),
        out("guessOut2",3),
        out("guessOut3",3),
        out("numRightColors",3),
        out("numRightColorsRightPlaces",3),
        out("winIndicator",1),
    };

    public AttemptBlock(Node parent,
        Wire valueIn0,
        Wire valueIn1,
        Wire valueIn2,
        Wire valueIn3,
        Wire setValue,
        Wire guessIn0,
        Wire guessIn1,
        Wire guessIn2,
        Wire guessIn3,
        Wire setGuess,
        Wire valueOut0,
        Wire valueOut1,
        Wire valueOut2,
        Wire valueOut3,
        Wire guessOut0,
        Wire guessOut1,
        Wire guessOut2,
        Wire guessOut3,
        Wire numRightColors,
        Wire numRightColorsRightPlaces,
        Wire winIndicator
    ) {
    }
super(parent);
connect("valueIn0", valueIn0);
connect("valueIn1", valueIn1);
connect("valueIn2", valueIn2);
connect("valueIn3", valueIn3);
connect("setValue", setValue);
connect("guessIn0", guessIn0);
connect("guessIn1", guessIn1);
connect("guessIn2", guessIn2);
connect("guessIn3", guessIn3);
connect("setGuess", setGuess);

call super(class);
connect("valueOut0", valueOut0);
connect("valueOut1", valueOut1);
connect("valueOut2", valueOut2);
connect("valueOut3", valueOut3);
connect("guessOut0", guessOut0);
connect("guessOut1", guessOut1);
connect("guessOut2", guessOut2);
connect("guessOut3", guessOut3);
connect("numRightColors", numRightColors);
connect("numRightColorsRightPlaces", 
    numRightColorsRightPlaces);
connect("winIndicator", winIndicator);

Wire setValueInternalCtl = wire(1,"setValueInternalCtl");
Wire setGuessInternalCtl = wire(1,"setGuessInternalCtl");
Wire setValueInternal = or(setValue,setValueInternalCtl, 
    "setValueInternal");
Wire setGuessInternal = or(setGuess,setGuessInternalCtl, 
    "setGuessInternal");
new SingleAssertState(this,setValue,setValueInternalCtl, 
    "setValueStateMachine");
new SingleAssertState(this,setGuess,setGuessInternalCtl, 
    "setGuessStateMachine");

regce_o(valueIn0,setValueInternal,valueOut0,"valueReg0");
regce_o(valueIn1,setValueInternal,valueOut1,"valueReg1");
regce_o(valueIn2,setValueInternal,valueOut2,"valueReg2");
regce_o(valueIn3,setValueInternal,valueOut3,"valueReg3");

Wire guessReg0 = regce_o(guessIn0, setGuessInternal, 
    guessOut0, "guessReg0");
Wire guessReg1 = regce_o(guessIn1, setGuessInternal, 
    guessOut1, "guessReg1");
Wire guessReg2 = regce_o(guessIn2, setGuessInternal, 
    guessOut2, "guessReg2");
Wire guessReg3 = regce_o(guessIn3, setGuessInternal, 
    guessOut3, "guessReg3");

// If a new value is set, then there is not a new guess
// set too:
Wire guessIsSet = regr(setGuessInternal,setGuess,
    setValueInternal,"guessIsSet");

Wire numRightColorsInternal
    = wire( 3, "numRightColorsInternal" );
Wire numRightColorsRightPlacesInternal
    = wire( 3, "numRightColorsRightPlacesInternal" );
Wire winIndicatorInternal
    = wire( 1, "winIndicatorInternal" );
new AttemptBlockRight(this, valueOut0, valueOut1,
    valueOut2, valueOut3,
    guessReg0, guessReg1,
    guessReg2, guessReg3,
    numRightColorsInternal,
    numRightColorsRightPlacesInternal,
    winIndicatorInternal);

and_o(winIndicatorInternal,guessIsSet,winIndicator);

// Generate Number of Colors in the Right Place
Wire zero = constant(3,0,"zero3bit");
mux_o(zero,numRightColorsRightPlacesInternal,guessIsSet,
    numRightColorsRightPlaces);

// Generate Number of Colors Correct
mux_o(zero,numRightColorsInternal,
    guessIsSet,numRightColors);
}

public void initUserDefinedNode(UserDefinedNode udn) {
    udn.setPortSeperation(0);
    udn.setSize(100,70);
    //udn.addInPort(0,20,"c"); //add an implicit clock port.
    //udn.addOutPort(40,30,"value");
}

drawn.Obj &&

public void paint(UserDefinedNode udn) {
    /*
    int color = 0;
    try {
        color = value.get((Browser)this);
    } catch ( byucc.jhdl.base.WireValueException wve ) {}"}  
    udn.drawRect(0,0,40,40);
    udn.setColor(COLOR_SET[color]);
    //udn.fillRect(1,1,28,38);
    udn.drawString("COLOR",1,10);
    //Add creative stuff--
    */
}

AttemptBlockRight.java
package model;

import byucc.jhdl.base.Browser;
import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

public class AttemptBlockRight extends Logic
  implements PGCommon/*, UserDefinedSchematic*/, Browser {

  public static CellInterface[] cell_interface = {
    in ("value0",3),
    in ("value1",3),
    in ("value2",3),
    in ("value3",3),
    in ("guess0",3),
    in ("guess1",3),
    in ("guess2",3),
    in ("guess3",3),
    out("numRightColors",3),
    out("numRightColorsRightPlaces",3),
    out("winIndicator",1),
  };

  public AttemptBlockRight( Node parent,
    Wire value0,
    Wire value1,
    Wire value2,
    Wire value3,
    Wire guess0,
    Wire guess1,
    Wire guess2,
    Wire guess3,
    Wire numRightColors,
    Wire numRightColorsRightPlaces,
    Wire winIndicator
  ) {
    super(parent);
    connect("value0",value0);
    connect("value1",value1);
    connect("value2",value2);
    connect("value3",value3);
    connect("guess0",guess0);
    connect("guess1",guess1);
    connect("guess2",guess2);
    connect("guess3",guess3);
    connect("numRightColors",numRightColors);
    connect("numRightColorsRightPlaces",}
numRightColorsRightPlaces);
connect("winIndicator",winIndicator);

Wire g0v0Match = nor(xor(value0,guess0),"g0v0Match");
Wire g0v1Match = nor(xor(value1,guess0),"g0v1Match");
Wire g0v2Match = nor(xor(value2,guess0),"g0v2Match");
Wire g0v3Match = nor(xor(value3,guess0),"g0v3Match");

Wire g1v0Match = nor(xor(value0,guess1),"g1v0Match");
Wire g1v1Match = nor(xor(value1,guess1),"g1v1Match");
Wire g1v2Match = nor(xor(value2,guess1),"g1v2Match");
Wire g1v3Match = nor(xor(value3,guess1),"g1v3Match");

Wire g2v0Match = nor(xor(value0,guess2),"g2v0Match");
Wire g2v1Match = nor(xor(value1,guess2),"g2v1Match");
Wire g2v2Match = nor(xor(value2,guess2),"g2v2Match");
Wire g2v3Match = nor(xor(value3,guess2),"g2v3Match");

Wire g3v0Match = nor(xor(value0,guess3),"g3v0Match");
Wire g3v1Match = nor(xor(value1,guess3),"g3v1Match");
Wire g3v2Match = nor(xor(value2,guess3),"g3v2Match");
Wire g3v3Match = nor(xor(value3,guess3),"g3v3Match");

and_o(g0v0Match,
g1v1Match,
g2v2Match,
g3v3Match,
winIndicator);

// Generate Number of Colors in the Right Place
new AttemptBlockRightPlaces(this,
g0v0Match,
g0v1Match,
g0v2Match,
g0v3Match,
g1v0Match,
g1v1Match,
g1v2Match,
g1v3Match,
g2v0Match,
g2v1Match,
g2v2Match,
g2v3Match,
g3v0Match,
g3v1Match,
g3v2Match,
g3v3Match,
numRightColorsRightPlaces);

// Generate Number of Colors Correct
new AttemptBlockRightColors(this,
public void initUserDefinedNode(UserDefinedNode udn) {
    udn.setPortSeparation(0);
    udn.setSize(100, 70);
    //udn.addInPort(0,20,"c"); //add an implicit clock port.
    //udn.addOutPort(40,30,"value");
}

public void paint(UserDefinedNode udn) {
    /*
    int color = 0;
    try {
        color = value.get((Browser)this);
    } catch ( byucc.jhdl.base.WireValueException wve ) {} 
    udn.drawRect(0,0,40,40);
    udn.setColor(COLOR_SET[color]);
    //udn.fillRect(1,1,28,38);
    udn.drawString("COLOR",1,10);
    */
}

AttemptBlockRightColors.java

package model;

import byucc.jhdl.base.Browser;
import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

public class AttemptBlockRightColors extends Logic
    implements PGCommon/*, UserDefinedSchematic*/, Browser {
    public static CellInterface[] cell_interface = {
        in ("g0v0Match", 1),
        in ("g0v1Match", 1),
        in ("g0v2Match", 1),
        in ("g0v3Match", 1),
        in ("g1v0Match", 1),
        in ("g1v1Match", 1),
        in ("g1v2Match", 1),
        in ("g1v3Match", 1),
        in ("g2v0Match", 1),
        in ("g2v1Match", 1),
        in ("g2v2Match", 1),
        in ("g2v3Match", 1),
        in ("g3v0Match", 1),
        in ("g3v1Match", 1),
        in ("g3v2Match", 1),
        in ("g3v3Match", 1),
        out("numRightColors",3),
    };

    public AttemptBlockRightColors(Node parent,
        Wire g0v0Match,
        Wire g0v1Match,
        Wire g0v2Match,
        Wire g0v3Match,
        Wire g1v0Match,
        Wire g1v1Match,
        Wire g1v2Match,
        Wire g1v3Match,
        Wire g2v0Match,
        Wire g2v1Match,
        Wire g2v2Match,
        Wire g2v3Match,
        Wire g3v0Match,
        Wire g3v1Match,
        Wire g3v2Match,
        Wire g3v3Match,
        Wire numRightColors
    ) {
        super(parent);
    }
}
connect( "g0v0Match", g0v0Match );
connect( "g0v1Match", g0v1Match );
connect( "g0v2Match", g0v2Match );
connect( "g0v3Match", g0v3Match );

connect( "glv0Match", glv0Match );
connect( "glv1Match", glv1Match );
connect( "glv2Match", glv2Match );
connect( "glv3Match", glv3Match );

connect( "g2v0Match", g2v0Match );
connect( "g2v1Match", g2v1Match );
connect( "g2v2Match", g2v2Match );
connect( "g2v3Match", g2v3Match );

connect( "g3v0Match", g3v0Match );
connect( "g3v1Match", g3v1Match );
connect( "g3v2Match", g3v2Match );
connect( "g3v3Match", g3v3Match );

connect("numRightColors",numRightColors);

// Build up some inverted signals:
Wire g0v0NotMatch = not(g0v0Match,"g0v0NotMatch");
Wire glv1NotMatch = not(glv1Match,"glv1NotMatch");
Wire g2v2NotMatch = not(g2v2Match,"g2v2NotMatch");
Wire g3v3NotMatch = not(g3v3Match,"g3v3NotMatch");

// Generate Number of Colors Correct
Wire g0v0ColorMatch = g0v0Match;
Wire g0v0NotColorMatch = not(g0v0ColorMatch,
    "g0v0NotColorMatch");
Wire g0v1ColorMatch = and(g0v1Match,
    glv1NotMatch,
    g0v0NotColorMatch,
    "g0v1ColorMatch");
Wire g0v1NotColorMatch = not(g0v1ColorMatch,
    "g0v1NotColorMatch");
Wire g0v2ColorMatch = and(g0v2Match,
    g2v2NotMatch,
    g0v0NotColorMatch,
    g0v1NotColorMatch,
    "g0v2ColorMatch");
Wire g0v2NotColorMatch = not(g0v2ColorMatch,
    "g0v2NotColorMatch");
Wire g0v3ColorMatch = and(and(g0v3Match,g3v3NotMatch),
    g0v0NotColorMatch,
    g0v1NotColorMatch,
    g0v2NotColorMatch,
    "g0v3ColorMatch");
Wire g0v3NotColorMatch = not(g0v3ColorMatch,
    "g0v3NotColorMatch");
Wire g0ColorMatch = or(g0v0ColorMatch,g0v1ColorMatch,
    g0v2ColorMatch,g0v3ColorMatch,
    "g0ColorMatch");
"g0ColorMatch";

Wire g1v1ColorMatch = g1v1Match;
Wire g1v1NotColorMatch = not(g1v1ColorMatch, "g1v1NotColorMatch");

Wire g1v2ColorMatch = and(g1v2Match, g2v2NotMatch, g0v2NotColorMatch, g1v1NotColorMatch, "g1v2ColorMatch");
Wire g1v2NotColorMatch = not(g1v2ColorMatch, "g1v2NotColorMatch");

Wire g1v3ColorMatch = and(and(g1v3Match, g3v3NotMatch), g0v3NotColorMatch, g1v3NotColorMatch, "g1v3ColorMatch");
Wire g1v3NotColorMatch = not(g1v3ColorMatch, "g1v3NotColorMatch");

Wire g1v0ColorMatch = and(and(g1v0Match, g0v0NotMatch), g1v0NotColorMatch, g1v2NotColorMatch, g1v3NotColorMatch, "g1v0ColorMatch");
Wire g1v0NotColorMatch = not(g1v0ColorMatch, "g1v0NotColorMatch");

Wire g2v2ColorMatch = g2v2Match;
Wire g2v2NotColorMatch = not(g2v2ColorMatch, "g2v2NotColorMatch");

Wire g2v3ColorMatch = and(and(g2v3Match, g3v3NotMatch), g0v3NotColorMatch, g1v3NotColorMatch, g2v2NotColorMatch, "g2v3ColorMatch");
Wire g2v3NotColorMatch = not(g2v3ColorMatch, "g2v3NotColorMatch");

Wire g2v0ColorMatch = and(and(g2v0Match, g0v0NotMatch), g1v0NotColorMatch, g2v2NotColorMatch, g2v3NotColorMatch, "g2v0ColorMatch");
Wire g2v0NotColorMatch = not(g2v0ColorMatch, "g2v0NotColorMatch");

Wire g2v1ColorMatch = and(g2v1Match, g1v1NotMatch, and(g0v1NotColorMatch, g2v2NotColorMatch, 

Wire g1ColorMatch = or(g1v0ColorMatch, g1v1ColorMatch, g1v2ColorMatch, g1v3ColorMatch, "g1ColorMatch");
new Sum4Bits(this, 
  g0ColorMatch, 
g1ColorMatch, 
g2ColorMatch, 
g3ColorMatch, 
numRightColors);
}

public void initUserDefinedNode(UserDefinedNode udn) { 
  udn.setPortSeperation(0); 
  udn.setSize(100, 70); 
  //udn.addInPort(0,20,"c"); //add an implicit clock port. 
  //udn.addOutPort(40,30,"value");
public void paint(UserDefinedNode udn) {
    /*
     * int color = 0;
     * try {
     *     color = value.get((Browser)this);
     * } catch (byucc.jhdl.base.WireValueException wve) {} 
     * udn.drawRect(0,0,40,40);
     * udn.setColor(COLOR_SET[color]);
     * //udn.fillRect(1,1,28,38);
     * udn.drawString("COLOR",1,10);
     * //Add creative stuff--
     */
    }
}

AttemptBlockRightPlaces.java

package model;

import byucc.jhdl.base.Browser;
import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

public class AttemptBlockRightPlaces extends Logic
    implements PGCommon/*, UserDefinedSchematic*/, Browser {
    public static CellInterface[] cell_interface =
    {
        in ("g0v0Match",1),
        in ("g0v1Match",1),
        in ("g0v2Match",1),
        in ("g0v3Match",1),
        in ("g1v0Match",1),
        in ("g1v1Match",1),
        in ("g1v2Match",1),
        in ("g1v3Match",1),
        in ("g2v0Match",1),
        in ("g2v1Match",1),
        in ("g2v2Match",1),
        in ("g2v3Match",1),
        in ("g3v0Match",1),
        in ("g3v1Match",1),
        in ("g3v2Match",1),
        in ("g3v3Match",1),
    }
public AttemptBlockRightPlaces(Node parent,
        Wire g0v0Match,
        Wire g0v1Match,
        Wire g0v2Match,
        Wire g0v3Match,
        Wire g1v0Match,
        Wire g1v1Match,
        Wire g1v2Match,
        Wire g1v3Match,
        Wire g2v0Match,
        Wire g2v1Match,
        Wire g2v2Match,
        Wire g2v3Match,
        Wire g3v0Match,
        Wire g3v1Match,
        Wire g3v2Match,
        Wire g3v3Match,
        Wire numRightColorsRightPlaces
    ) {
        super(parent);
        connect( "g0v0Match", g0v0Match );
        connect( "g0v1Match", g0v1Match );
        connect( "g0v2Match", g0v2Match );
        connect( "g0v3Match", g0v3Match );
        connect( "g1v0Match", g1v0Match );
        connect( "g1v1Match", g1v1Match );
        connect( "g1v2Match", g1v2Match );
        connect( "g1v3Match", g1v3Match );
        connect( "g2v0Match", g2v0Match );
        connect( "g2v1Match", g2v1Match );
        connect( "g2v2Match", g2v2Match );
        connect( "g2v3Match", g2v3Match );
        connect( "g3v0Match", g3v0Match );
        connect( "g3v1Match", g3v1Match );
        connect( "g3v2Match", g3v2Match );
        connect( "g3v3Match", g3v3Match );
        connect( "numRightColorsRightPlaces", numRightColorsRightPlaces );

        // Generate Number of Colors in the Right Place
        new Sum4Bits(this,
                g0v0Match,
                g1v1Match,
                g2v2Match,
public void initUserDefinedNode(UserDefinedNode udn) {
    udn.setPortSeperation(0);
    udn.setSize(100,70);
    //udn.addInPort(0,20,"c"); //add an implicit clock port.
    //udn.addOutPort(40,30,"value");
}

public void paint(UserDefinedNode udn) {
    /*
    int color = 0;
    try {
        color = value.get((Browser)this);
    } catch ( byucc.jhdl.base.WireValueException wve ) {} 
    udn.drawRect(0,0,40,40);
    udn.setColor(COLOR_SET[color]);
    //udn.fillRect(1,1,28,38);
    udn.drawString("COLOR",1,10);
    //Add creative stuff--
    */
}

package model;
import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

public class Multiplex extends Logic {
    public static CellInterface[] cell_interface = {
        in ("address",4),
        in ("value0","WIDTH"),
        in ("value1","WIDTH"),
        in ("value2","WIDTH"),
        in ("value3","WIDTH"),
        in ("value4","WIDTH"),
        in ("value5","WIDTH"),
        in ("value6","WIDTH"),
        in ("value7","WIDTH"),
        in ("value8","WIDTH"),
        in ("value9","WIDTH"),
        in ("value10","WIDTH"),
    }
public Multiplex(Node parent, Wire address,
               Wire value0,
               Wire value1,
               Wire value2,
               Wire value3,
               Wire value4,
               Wire value5,
               Wire value6,
               Wire value7,
               Wire value8,
               Wire value9,
               Wire value10,

               Wire output
         ) {

         super(parent);

         bind("WIDTH",value0.getWidth());
         connect("address",address);

         connect("value0",value0);
         connect("value1",value1);
         connect("value2",value2);
         connect("value3",value3);
         connect("value4",value4);
         connect("value5",value5);
         connect("value6",value6);
         connect("value7",value7);
         connect("value8",value8);
         connect("value9",value9);
         connect("value10",value10);

         connect("output",output);

         Wire lower8
             = mux(value0,
                   value1,
                   value2,
                   value3,
                   value4,
                   value5,
                   value6,
                   value7,
                   address.range(2,0));

         Wire upper3
             = mux(value8,
                   value9,
package model;

import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

public class PG extends Logic implements PGCommon {
    public static CellInterface[] cell_interface = {
        in ("newGame",1),
        in ("pegInput0",3),
        in ("pegInput1",3),
        in ("pegInput2",3),
        in ("pegInput3",3),
        in ("registerGuess",1),
        in ("statusAddress",4),
        out("guess0",3),
        out("guess1",3),
        out("guess2",3),
        out("guess3",3),
        out("numRightColors",3),
        out("numRightColorsRightPlaces",3),
        out("gameOver",1),
        out("youWin",1),
    };

    public PG(Node parent,
        Wire newGame,
        Wire pegInput0,
        Wire pegInput1,
        Wire pegInput2,
        Wire pegInput3,
        Wire registerGuess,
Wire statusAddress,

Wire guess0,
Wire guess1,
Wire guess2,
Wire guess3,
Wire numRightColors,
Wire numRightColorsRightPlaces,

Wire gameOver,
Wire youWin
}

super(parent);
connect("newGame",newGame);

connect("pegInput0",pegInput0);
connect("pegInput1",pegInput1);
connect("pegInput2",pegInput2);
connect("pegInput3",pegInput3);
connect("registerGuess",registerGuess);

connect("statusAddress",statusAddress);
connect("guess0",guess0);
connect("guess1",guess1);
connect("guess2",guess2);
connect("guess3",guess3);
connect("numRightColors",numRightColors);
connect("numRightColorsRightPlaces",numRightColorsRightPlaces);
connect("gameOver",gameOver);
connect("youWin",youWin);

Wire newGameInternalCtl = wire(1,"newGameInternalCtl");
Wire registerGuessInternalCtl = wire(1,"registerGuessInternalCtl");
Wire newGameInternal = or(newGame,newGameInternalCtl,"newGameInternal");
Wire registerGuessInternal = and(registerGuess,registerGuessInternalCtl,"registerGuessInternal");
new SingleAssertState(this,newGame,newGameInternalCtl,"newGameStateMachine");
new SingleAssertState(this,registerGuess,registerGuessInternalCtl,"registerGuessStateMachine");

Wire currentAttempt = wire(4,"currentAttempt");
regre_o(add(currentAttempt,
constant(4,1),
"currentAttemptPlusOne"),
registerGuessInternal,
newGameInternal,
currentAttempt);
and_o(currentAttempt.gw(3),
    or(currentAttempt.gw(2),
        currentAttempt.gw(1),"orCurrAtt"),
    gameOver);

Wire randValue0 = wire( 3, "randValue0" );
Wire randValue1 = wire( 3, "randValue1" );
Wire randValue2 = wire( 3, "randValue2" );
Wire randValue3 = wire( 3, "randValue3" );
new RandomPuzzle(this,randValue0,randValue1,
    randValue2,randValue3);

Wire[] setGuessArray = new Wire[10];
Wire[] guessOut0 = new Wire[11];
Wire[] guessOut1 = new Wire[11];
Wire[] guessOut2 = new Wire[11];
Wire[] guessOut3 = new Wire[11];
Wire[] numRightColorsArray = new Wire[11];
Wire[] numRightColorsRightPlacesArray = new Wire[11];

Wire winIndicators = wire( 10, "winIndicators" );

for ( int attempt = 0; attempt < 10; ++attempt ) {
    setGuessArray[attempt] = wire(1,"setGuess"+attempt);
    guessOut0[attempt]=wire(3,"guessOut0Attempt"+attempt);
    guessOut1[attempt]=wire(3,"guessOut1Attempt"+attempt);
    guessOut2[attempt]=wire(3,"guessOut2Attempt"+attempt);
    guessOut3[attempt]=wire(3,"guessOut3Attempt"+attempt);
    numRightColorsArray[attempt] = wire(3,"numRightColors"+attempt);
    numRightColorsRightPlacesArray[attempt]
        = wire(3,"numRightColorsRightPlaces"+attempt);
    new AttemptBlock(this,
        randValue0,randValue1,
        randValue2,randValue3,
        newGameInternal, /*setValue*/
        pegInput0,pegInput1,
        pegInput2,pegInput3,
        setGuessArray[attempt],
        wire(3,"valueOut0_"+attempt),
        wire(3,"valueOut1_"+attempt),
        wire(3,"valueOut2_"+attempt),
        wire(3,"valueOut3_"+attempt),
        guessOut0[attempt],
        guessOut1[attempt],
        guessOut2[attempt],
        guessOut3[attempt],
        numRightColorsArray[attempt],
        numRightColorsRightPlacesArray[attempt],
        winIndicators.gw(attempt) );
}

// This Block will only output the true solution:
guessOut0[10] = wire(3,"solution0");
guessOut1[10] = wire(3,"solution1");
guessOut2[10] = wire(3,"solution2");
guessOut3[10] = wire(3,"solution3");
numRightColorsArray[10] = wire(3,"numRightColorsSolution");
numRightColorsRightPlacesArray[10] = wire(3,"numRightColorsRightPlacesSolution");

new SolutionBlock(this,
    randValue0,randValue1,
    randValue2,randValue3,
    newGameInternal,/*setValue*/
    guessOut0[10],
    guessOut1[10],
    guessOut2[10],
    guessOut3[10],
    numRightColorsArray[10],
    numRightColorsRightPlacesArray[10])
);

Wire redirectSel = wire(10,"redirectSel");
new RedirectSelect(this,currentAttempt,redirectSel);
new Redirect(this,redirectSel,registerGuessInternal,
    setGuessArray[0],
    setGuessArray[1],
    setGuessArray[2],
    setGuessArray[3],
    setGuessArray[4],
    setGuessArray[5],
    setGuessArray[6],
    setGuessArray[7],
    setGuessArray[8],
    setGuessArray[9]);

new Multiplex(this,
    statusAddress,
    guessOut0[0],
    guessOut0[1],
    guessOut0[2],
    guessOut0[3],
    guessOut0[4],
    guessOut0[5],
    guessOut0[6],
    guessOut0[7],
    guessOut0[8],
    guessOut0[9],
    guessOut0[10],
    guess0);
new Multiplex(this,
    statusAddress,
    guessOut1[0],
    guessOut1[1],
    guessOut1[2],
    guessOut1[3],
    guessOut1[4],
    guessOut1[5],
    guessOut1[6],
    guessOut1[7],
    guessOut1[8],
    guessOut1[9],
    guessOut1[10],
    guess1);
new Multiplex(this, statusAddress, guessOut1[0],
guessOut1[1],
guessOut1[2],
guessOut1[3],
guessOut1[4],
guessOut1[5],
guessOut1[6],
guessOut1[7],
guessOut1[8],
guessOut1[9],
guessOut1[10],
guess1);
new Multiplex(this, statusAddress, guessOut2[0],
guessOut2[1],
guessOut2[2],
guessOut2[3],
guessOut2[4],
guessOut2[5],
guessOut2[6],
guessOut2[7],
guessOut2[8],
guessOut2[9],
guessOut2[10],
guess2);
new Multiplex(this, statusAddress, guessOut3[0],
guessOut3[1],
guessOut3[2],
guessOut3[3],
guessOut3[4],
guessOut3[5],
guessOut3[6],
guessOut3[7],
guessOut3[8],
guessOut3[9],
guessOut3[10],
guess3);
new Multiplex(this, statusAddress, numRightColorsArray[0],
numRightColorsArray[1],
numRightColorsArray[2],
numRightColorsArray[3],
numRightColorsArray[4],
numRightColorsArray[5],
numRightColorsArray[6],
numRightColorsArray[7],
numRightColorsArray[8],
numRightColorsArray[9],
numRightColorsArray[10],
numRightColors);
new Multiplex(this, statusAddress, numRightColorsRightPlacesArray[0],
numRightColorsRightPlacesArray[1],
numRightColorsRightPlacesArray[2],
numRightColorsRightPlacesArray[3],
numRightColorsRightPlacesArray[4],
numRightColorsRightPlacesArray[5],
numRightColorsRightPlacesArray[6],
numRightColorsRightPlacesArray[7],
numRightColorsRightPlacesArray[8],
numRightColorsRightPlacesArray[9],
numRightColorsRightPlacesArray[10],
numRightColorsRightPlaces);
PGCommon.java

package model;

import java.awt.Color;

public interface PGCommon {
    public static final Color[] COLOR_SET = {
        Color.blue,
        Color.green.darker().darker(),
        Color.yellow,
        Color.magenta.darker().darker(),
        Color.pink,
    };

    static final int DEFAULT_DATA_SET_SIZE = 5;
    static final int DEFAULT_NUM_PEGS = 4;
    static final int DEFAULT_NUMTRIES = 10;

    static int dataSetSize = DEFAULT_DATA_SET_SIZE;
    static int numPegs = DEFAULT_NUM_PEGS;
    static int numTries = DEFAULT_NUMTRIES;
}

RandomPuzzle.java

package model;

import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

public class RandomPuzzle extends Logic {
    public static CellInterface[] cell_interface = {
        out("value0",3),
        out("value1",3),
    }
public RandomPuzzle( Node parent, Wire value0, Wire value1, Wire value2, Wire value3 ) {
    super(parent);
    connect("value0",value0);
    connect("value1",value1);
    connect("value2",value2);
    connect("value3",value3);

    new RandomPuzzleElement(this, value0, 0xE);
    new RandomPuzzleElement(this, value1, 0x8);
    new RandomPuzzleElement(this, value2, 0xD);
    new RandomPuzzleElement(this, value3, 0xB);
}

RandomPuzzleElement.java

package model;

import byucc.jhdl.base.Browser;
import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

public class RandomPuzzleElement extends Logic
    implements PGCommon, UserDefinedSchematic, Browser {
    public static CellInterface[] cell_interface = {
        out("value",3),
    };
    private static final int width = 10;
    private Wire value;
    public RandomPuzzleElement( Node parent, Wire value,
int configInt
    ) {
    super(parent);
    this.value = connect("value",value);
    Wire flopOuts = wire( width, "flopOut" );
    Wire feedback = flopOuts.gw(width-1);
    Wire flopIn = feedback;
    for ( int wi = 0; wi < (width-1); ++wi ) {
        Wire flopOut, xorOut, andOut;
        flopOut = flopOuts.gw(wi);
        reg_o( flopIn, flopOut, "reg"+wi );
        if ( ((configInt>>wi)&1) == 1 ) {
            flopIn = wire( 1, "xor"+wi );
            xor_o( flopOut, feedback, flopIn );
        } else {
            flopIn = flopOut;
        }
    }
    regp_o( flopIn, feedback, "reg"+(width-1) );

    Wire subOut
        = sub( flopOuts, constant( width, 1, "const1" ) );
    Wire addInA = concat( gnd(),subOut.gw(1),subOut.gw(0) );
    add_o( addInA,concat( gnd(),gnd(),subOut.gw(2) ),value );
}

public void initUserDefinedNode(UserDefinedNode udn) {
    udn.setPortSeperation(0);
    udn.setSize(40,40);
    udn.addInPort(0,20,"c"); //add an implicit clock port.
    udn.addOutPort(40,30,"value");
}

public void paint(UserDefinedNode udn) {
    int color = 0;
    try {
        color = value.get((Browser)this);
    } catch ( byucc.jhdl.base.WireValueException wve ) {}
    udn.drawRect(0,0,40,40);
    udn.setColor(COLOR_SET[color]);
    //udn.fillRect(1,1,28,38);
    udn.drawString("COLOR",1,10);
    //Add creative stuff--
}

Redirect.java

package model;

import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

public class Redirect extends Logic {
    public static CellInterface[] cell_interface = {
        in ("select",10),
        in ("value","WIDTH"),
        out ("output0","WIDTH"),
        out ("output1","WIDTH"),
        out ("output2","WIDTH"),
        out ("output3","WIDTH"),
        out ("output4","WIDTH"),
        out ("output5","WIDTH"),
        out ("output6","WIDTH"),
        out ("output7","WIDTH"),
        out ("output8","WIDTH"),
        out ("output9","WIDTH"),
        param("WIDTH",INTEGER),
    };

    public Redirect( Node parent,
        Wire select,
        Wire value,
        Wire output0,
        Wire output1,
        Wire output2,
        Wire output3,
        Wire output4,
        Wire output5,
        Wire output6,
        Wire output7,
        Wire output8,
        Wire output9
    ) {
        super(parent);
        int width = value.getWidth();
        bind("WIDTH",width);
        connect("select",select);
        connect("value",value);
        connect("output0",output0);
        connect("output1",output1);
        connect("output2",output2);
        connect("output3",output3);
        connect("output4",output4);
        connect("output5",output5);
        connect("output6",output6);
        connect("output7",output7);
        connect("output8",output8);
        connect("output9",output9);
    }
}
connect("output6", output6);
connect("output7", output7);
connect("output8", output8);
connect("output9", output9);

Wire zero = constant(width, 0, "zero");

mux_o(zero, value, select.gw(0), output0);
mux_o(zero, value, select.gw(1), output1);
mux_o(zero, value, select.gw(2), output2);
mux_o(zero, value, select.gw(3), output3);
mux_o(zero, value, select.gw(4), output4);
mux_o(zero, value, select.gw(5), output5);
mux_o(zero, value, select.gw(6), output6);
mux_o(zero, value, select.gw(7), output7);
mux_o(zero, value, select.gw(8), output8);
mux_o(zero, value, select.gw(9), output9);

RedirectSelect.java

package model;

import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

public class RedirectSelect extends Logic {
    public static CellInterface[] cell_interface = {
        in ("address", 4),
        out("select", 10),
    };

    public RedirectSelect( Node parent,
                           Wire address,
                           Wire select
    ) {
        super(parent);
        connect("address", address);

        connect("select", select);

        nor_o(address,
              select.gw(0));
        and_o(nor(address.range(3, 1)),
              address.gw(0),
              select.gw(1));
        and_o(nor(address.gw(3)),
              select.gw(2));
    }
}
SingleAssertState.java

```java
package model;

import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

public class SingleAssertState extends Logic {

```
public static CellInterface[] cell_interface =
{
    in("signalIn",1),
    out("signalOut",1),
};

public SingleAssertState( Node parent,
    Wire signalIn,
    Wire signalOut,
    String name
)
{
    super(parent,name);
    connect("signalIn",signalIn);
    connect("signalOut",signalOut);

    Wire state = wire( 2, "state" );
    reg_o( signalIn, state.gw(0), "state0" );
    reg_o( and(signalIn,state.gw(0)),state.gw(1),"state1" );
    and_o( state.gw(0), not(state.gw(1)), signalOut );
}

SolutionBlock.java

package model;

import byucc.jhdl.base.Browser;
import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

public class SolutionBlock extends Logic
    implements PGCommon/*, UserDefinedSchematic*/, Browser {
    public static CellInterface[] cell_interface =
    {
        in ("valueIn0",3),
        in ("valueIn1",3),
        in ("valueIn2",3),
        in ("valueIn3",3),
        in ("setValue",1),
        out("valueOut0",3),
        out("valueOut1",3),
        out("valueOut2",3),
        out("valueOut3",3),
        out("numRightColors",3),
        out("numRightColorsRightPlaces",3),
    };
    public SolutionBlock( Node parent,
Wire valueIn0,
Wire valueIn1,
Wire valueIn2,
Wire valueIn3,
Wire setValue,

Wire valueOut0,
Wire valueOut1,
Wire valueOut2,
Wire valueOut3,

Wire numRightColors,
Wire numRightColorsRightPlaces
)

super(parent);
connect("valueIn0",valueIn0);
connect("valueIn1",valueIn1);
connect("valueIn2",valueIn2);
connect("valueIn3",valueIn3);
connect("setValue",setValue);
connect("valueOut0",valueOut0);
connect("valueOut1",valueOut1);
connect("valueOut2",valueOut2);
connect("valueOut3",valueOut3);
connect("numRightColors",numRightColors);
connect("numRightColorsRightPlaces",numRightColorsRightPlaces);

Wire setValueInternalCtl = wire(1,"setValueInternalCtl");
Wire setValueInternal = or(setValue,setValueInternalCtl,"setValueInternal");
new SingleAssertState(this,setValue,setValueInternalCtl,"setValueStateMachine");

regce_o(valueIn0,setValueInternal,valueOut0,"valueReg0");
regce_o(valueIn1,setValueInternal,valueOut1,"valueReg1");
regce_o(valueIn2,setValueInternal,valueOut2,"valueReg2");
regce_o(valueIn3,setValueInternal,valueOut3,"valueReg3");

constant_o(numRightColors,4);
constant_o(numRightColorsRightPlaces,4);
}

class UserDefinedNode {
  public void initUserDefinedNode(UserDefinedNode udn) {
    udn.setPortSeperation(0);
    udn.setSize(100,70);
    //udn.addInPort(0,20,"c"); //add an implicit clock port.
    //udn.addOutPort(40,30,"value");
  }

class UserDefinedNode {
  public void paint(UserDefinedNode udn) {

```java
/*
int color = 0;
try {
    color = value.get((Browser)this);
} catch ( byucc.jhdl.base.WireValueException wve ) {} 
udn.drawRect(0,0,40,40);
udn.setColor(COLOR_SET[color]);
//udn.fillRect(1,1,28,38);
udn.drawString("COLOR",1,10);
//Add creative stuff--
*/
}

Sum4Bits.java

package model;

import byucc.jhdl.base.Browser;
import byucc.jhdl.base.CellInterface;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;

public class Sum4Bits extends Logic implements PGCommon/*, UserDefinedSchematic*/, Browser {
    public static CellInterface[] cell_interface = {
        in ( "bit0", 1 ),
        in ( "bit1", 1 ),
        in ( "bit2", 1 ),
        in ( "bit3", 1 ),
        out( "sum", 3 ),
    };
    public Sum4Bits( Node parent,
        Wire bit0,
        Wire bit1,
        Wire bit2,
        Wire bit3,
        Wire sum
    ) {
        super(parent);
        connect( "bit0", bit0 );
        connect( "bit1", bit1 );
        connect( "bit2", bit2 );
        connect( "bit3", bit3 );
        connect("sum",sum);
    }
```
and_o(bit0, bit1, bit2, bit3, sum.gw(2));

or_o(and(or(and(bit0, bit1),
and(bit2, bit3)),
not(sum.gw(2))),
and(or(bit2, bit3),
xor(bit0, bit1)),
sum.gw(1));

xor_o(bit0, bit1, bit2, bit3, sum.gw(0));

}
}

tbPG.java

package model;

import byucc.jhdl.apps.Stimulator.Stimulator;
import byucc.jhdl.base.Cell;
import byucc.jhdl.base.HWSystem;
import byucc.jhdl.base.Node;
import byucc.jhdl.base.ProgrammaticTestBench;
import byucc.jhdl.base.Wire;
import byucc.jhdl.Logic.Logic;
import byucc.jhdl.Logic.TechMapper;
import byucc.jhdl.Xilinx.Virtex.VirtexTechMapper;
import byucc.jhdl.Xilinx.Virtex2.Virtex2TechMapper;

import model.PG;
import controller.PGBroker;

public class tbPG extends Logic
    implements ProgrammaticTestBench {

    public Cell design;

    private Wire newGame;
    private Wire pegInput0;
    private Wire pegInput1;
    private Wire pegInput2;
    private Wire pegInput3;
    private Wire registerGuess;
    private Wire statusAddress;

    private Wire guess0;
    private Wire guess1;
    private Wire guess2;
private Wire guess3;
private Wire numRightColors;
private Wire numRightColorsRightPlaces;
private Wire gameOver;
private Wire youWin;

public tbPG(Node parent) {
    this( parent, null );
}

public tbPG(Node parent, TechMapper tm) {
    super(parent);
    setDefaultTechMapper( (null == tm)?
        new VirtexTechMapper(true) : tm);

    design = new PG( this,
        (newGame=(wire(1,"newGame"))),
        (pegInput0=(wire(3,"pegInput0"))),
        (pegInput1=(wire(3,"pegInput1"))),
        (pegInput2=(wire(3,"pegInput2"))),
        (pegInput3=(wire(3,"pegInput3"))),
        (registerGuess = (wire(1,"registerGuess"))),
        (statusAddress = (wire(4,"statusAddress"))),
        (guess0=(wire(3,"guess0"))),
        (guess1=(wire(3,"guess1"))),
        (guess2=(wire(3,"guess2"))),
        (guess3=(wire(3,"guess3"))),
        (numRightColors = (wire(3,"numRightColors"))),
        (numRightColorsRightPlaces = (wire(3,"numRightColorsRightPlaces"))),
        (gameOver=(wire(1,"gameOver"))),
        (youWin=(wire(1,"youWin"))) );

    Stimulator stimulator = new Stimulator(this);
    stimulator.addWire(newGame);
    stimulator.addWire(pegInput0);
    stimulator.addWire(pegInput1);
    stimulator.addWire(pegInput2);
    stimulator.addWire(pegInput3);
    stimulator.addWire(registerGuess);
    stimulator.addWire(registerGuess);

    private HWSys tem;
system = getSystem();
system.reset();
while (true) {
    system.cycle(1);
}

public void reset() {
}

private int currNumRightColors;
private int currNumRightColorsRightPlaces;
private int[] currGuess = new int[4];
private int currGameOver;
private int currYouWin;

public void clock() {
    currNumRightColors = numRightColors.get(this);
    currNumRightColorsRightPlaces =
        numRightColorsRightPlaces.get(this);
    currGuess[0] = guess0.get(this);
    currGuess[1] = guess1.get(this);
    currGuess[2] = guess2.get(this);
    currGuess[3] = guess3.get(this);
    currGameOver = gameOver.get(this);
    currYouWin = youWin.get(this);
}

public int getNumRightColors() {
    //return numRightColors.get(this);
    return currNumRightColors;
}

public int getNumRightPlaces() {
    //return numRightColorsRightPlaces.get(this);
    return currNumRightColorsRightPlaces;
}

public boolean getGameOver() {
    //int go = gameOver.get(this);
    return (currGameOver == 1);
}

public boolean getGameWon() {
    //int won = youWin.get(this);
    return (currYouWin == 1);
}

public int[] getGuess() {
    return currGuess;
}
C.4.2 Guessing Game Application View

This section includes the following source code files:

- ColorPlacementListener.java
- GameCanvas.java
- GameViewerFrame.java
- PGUtil.java

ColorPlacementListener.java

```java
package view;

import java.awt.Color;

/** Other classes use this to listen to where the user has placed a color
 * @author Anthony Slade */
public interface ColorPlacementListener {
    public void colorsSubmitted(GameCanvas source, int row, int[] guess);
}
```

GameCanvas.java

```java
package view;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import java.util.Collections;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
import javax.swing.JComponent;
import model.PGCommon;
```
/** Visual representation of the PG game.
 * @author Anthony L. Slade */

class GameCanvas extends JComponent
    implements PGCommon {

    public static void main(String[] args) {
        GameCanvas gc = new GameCanvas();
        gc.setFeedback(1,3,2);
        javax.swing.JFrame frame = new javax.swing.JFrame();
        frame.getContentPane().add(gc);
        frame.pack();
        frame.show();
    }

    private static final int CANVAS_WIDTH = 300;
    private static final int CANVAS_HEIGHT = 400;
    private static final int RECT_WIDTH = 20;
    private static final int RECT_HEIGHT = 20;

    private static final int BASE_Y = 50;
    private static final int SPACE_Y = 10;
    private static final int BASE_X = 75;
    private static final int SPACE_X = 20;

    private static final int SUBMIT_X = 220;
    private static final int SUBMIT_WIDTH = 70;
    private static final int SUBMIT_HEIGHT = 20;

    private static final int FEEDBACK_BASE_X = SUBMIT_X + 20;
    private static final int FEEDBACK_WIDTH = 10;
    private static final int FEEDBACK_HEIGHT = 10;
    private static final int FEEDBACK_SPACE_X = 5;
    private static final int FEEDBACK_SPACE_Y = 1;

    public GameCanvas() {
        Dimension size = new Dimension(CANVAS_WIDTH,
                                         CANVAS_HEIGHT);
        setMinimumSize(size);
        setPreferredSize(size);
        setBackground(Color.white);
        setOpaque(true);
        draggableRects = new ColoredRectangle[dataSetSize];
        setRectangles = Collections.synchronizedSet(new HashSet());
        for ( int pal = 0; pal < dataSetSize; ++pal ) {
            int x = RECT_WIDTH,
                y = (2*pal+2)*RECT_HEIGHT;
            Color color = COLOR_SET[pal];
            ColoredRectangle cr = new ColoredRectangle(x,y,
                                                      RECT_WIDTH,
                                                      RECT_HEIGHT,
                                                      color);
            setRectangles.add(cr);
            draggableRects[pal] = new ColoredRectangle(x,y,
RECT_WIDTH,
RECT_HEIGHT,
color);
}
spaces = new ColoredRectangle[numTries][numPegs];
for ( int row = 0; row < numTries; ++row ) {
    int y = BASE_Y + row*(RECT_HEIGHT+SPACE_Y);
    for ( int col = 0; col < numPegs; ++col ) {
        int x = BASE_X + col*(RECT_WIDTH+SPACE_X);
        spaces[row][col] = new ColoredRectangle(x,y,RECT_WIDTH,
            RECT_HEIGHT,null);
    }
}
currRow = 0;
setMouseListening();
draggedRect = intersectedRect = null;
listeners = Collections.synchronizedSet(new HashSet());
submitButton = null;
feedback = new int[numTries][2];
canvasFont = new Font("Times",Font.PLAIN,12);
solution = new ColoredRectangle[numPegs];
for ( int rect = 0; rect < numPegs; ++rect ) {
    int x = BASE_X + rect*(RECT_WIDTH+SPACE_X);
    solution[rect] = new ColoredRectangle(x,BASE_Y - 40,
        RECT_WIDTH,
        RECT_HEIGHT, null);
}
gameEnded = null;
waitingForFeedback = false;
}

public void addListener(ColorPlacementListener cpl) {
    listeners.add(cpl);
}

public void removeListener(ColorPlacementListener cpl) {
    listeners.remove(cpl);
}

/** Used to tell the view what the feedback is */
public void setFeedback(int row, int numColorsRight,
        int numPlacesRight) {
    try {
        feedback[row][0] = numColorsRight;
        feedback[row][1] = numPlacesRight;
    } catch (ArrayIndexOutOfBoundsException aioobe) {
    }
    waitingForFeedback = false;
}

/** Tells the viewer that the game has ended and gives the */
public void gameEnded(int[] actualSolution, boolean won) {
    if (won)
        gameEnded = "Game Won!";
    else
        gameEnded = "Game Over";
    for (int sol = 0; sol < solution.length; ++sol)
        solution[sol].color = COLOR_SET[actualSolution[sol]];
    submitButton = null;
    currRow = -1;
    revalidate(); repaint();
}

/** Outside class determines which row the user can
 * manipulate at the present time.
 * @param row the new row. */
public void setCurrentRow(int row) {
    currRow = row;
}

/* Resets everything for a new game */
public void newGame() {
    currRow = 0;
    for (int row = 0; row < numTries; ++row) {
        for (int col = 0; col < numPegs; ++col) {
            ColoredRectangle cr = spaces[row][col];
            cr.color = null;
            cr.drawColor = null;
            setRectangles.remove(cr);
        }
        feedback[row][0] = 0;
        feedback[row][1] = 0;
    }
    for (int si = 0; si < solution.length; ++si)
        solution[si].color = null;
    gameEnded = null;
    revalidate(); repaint();
}

/** Draws the component on the screen */
public void paint(Graphics gr) {
    gr.setFont(canvasFont);
    gr.setColor(Color.white);
    gr.fillRect(0,0,CANVAS_WIDTH,CANVAS_HEIGHT);
    gr.setColor(Color.black);
    paintSolution(gr);
    paintGrid(gr);
    paintSetRectangles(gr);
    paintSubmitButton(gr);
    paintFeedback(gr);
    paintDraggedRectangle(gr);
}

/** Draws the background grid. */
private void paintGrid(Graphics gr) {
    gr.setColor(Color.black);
    int startX = 2*RECT_WIDTH+10;
    for ( int row = 0; row < numTries; ++row ) {
        gr.drawString(""+(row+1),startX+5,
        BASE_Y + 15 + row*(RECT_HEIGHT+SPACE_Y));
        for ( int col = 0; col < numPegs; ++col ) {
            ColoredRectangle cr = spaces[row][col];
            if ( null != cr.color ) {
                continue;
            }
            gr.drawRect((int)cr.getX(),
            (int)cr.getY(),
            (int)cr.getWidth(),
            (int)cr.getHeight());
        }
        gr.drawRect(startX,0,1,CANVAS_HEIGHT);
        gr.drawRect(startX,BASE_Y-10,CANVAS_WIDTH-startX,1);
    }
    /** Draws the solution if it has been set */
    private void paintSolution(Graphics gr) {
        for ( int col = 0; col < numPegs; ++col ) {
            ColoredRectangle cr = solution[col];
            if ( null == cr.color ) {
                gr.setColor(Color.black);
                gr.drawRect((int)cr.getX(),
                (int)cr.getY(),
                (int)cr.getWidth(),
                (int)cr.getHeight());
                gr.setColor(Color.red);
                gr.drawString("?",(int)cr.getX()+8,
                (int)cr.getY()+15);
            } else {
                gr.setColor(cr.color);
                gr.fillRect((int)cr.getX(),
                (int)cr.getY(),
                (int)cr.getWidth(),
                (int)cr.getHeight());
            }
    }
    if ( null != gameEnded ) {
        ColoredRectangle cr = solution[numPegs-1];
        int x = (int)cr.getX()+RECT_WIDTH+SPACE_X-10;
        int y = (int)cr.getY()+15;
        gr.drawString(gameEnded,x,y);
    }
    /** Draws the rectangles that have their color set. */
    private void paintSetRectangles(Graphics gr) {
        for ( Iterator itr = setRectangles.iterator();

itr.hasNext(); ) {
    ColoredRectangle cr = (ColoredRectangle)itr.next();
    gr.setColor(cr.drawColor);
    gr.fillRect((int)cr.getX(),
                 (int)cr.getY(),
                 (int)cr.getWidth(),
                 (int)cr.getHeight());
}
}

/** Draws the dragged rectangle, if any. */
private void paintDraggedRectangle(Graphics gr) {
    if ( null == draggedRect )
        return;
    ColoredRectangle cr;
    if ( null != (cr=intersectedRect) ) {
        gr.setColor(cr.drawColor);
        gr.fillRect((int)cr.getX(),
                     (int)cr.getY(),
                     (int)cr.getWidth(),
                     (int)cr.getHeight());
    }
    cr = draggedRect;
    gr.setColor(cr.drawColor);
    gr.fillRect((int)cr.getX(),
                 (int)cr.getY(),
                 (int)cr.getWidth(),
                 (int)cr.getHeight());
}

/** Draws the submit button if there is one */
private void paintSubmitButton(Graphics gr) {
    if ( null == submitButton )
        return;
    gr.setColor(Color.black);
    int x = (int)submitButton.getX();
    int y = (int)submitButton.getY();
    int w = (int)submitButton.getWidth();
    int h = (int)submitButton.getHeight();
    gr.drawRect(x,y,w,h);
    if ( submitPressed )
        gr.setColor(Color.green.darker().darker());
    gr.drawRect(x+2,y+2,w-4,h-4);
    gr.drawString("Submit",x+15,y+15);
}

/** Draws the feedback if there is any. NOTE! This method * is currently hard-coded for four pegs. It needs to be * parameterized. */
private void paintFeedback(Graphics gr) {
    for (int row = 0; row < numTries; ++row) {
        int y = BASE_Y + row*(RECT_HEIGHT+SPACE_Y);
        int rightColors = feedback[row][0];
        int rightPlaces = feedback[row][1];
        ...
if ( row == currRow && waitingForFeedback ) {
  // this means we are waiting for feedback to come in
  gr.setColor(Color.gray);
  gr.drawString("?",4+FEEDBACK_BASE_X,
      y+FEEDBACK_HEIGHT);
  gr.drawString("?",4+FEEDBACK_BASE_X
      +FEEDBACK_WIDTH+FEEDBACK_SPACE_X,
      y+FEEDBACK_HEIGHT);
  gr.drawString("?",4+FEEDBACK_BASE_X,
      y+2*FEEDBACK_HEIGHT+FEEDBACK_SPACE_Y);
  gr.drawString("?",4+FEEDBACK_BASE_X+FEEDBACK_WIDTH
      +FEEDBACK_SPACE_X,
      y+2*FEEDBACK_HEIGHT+FEEDBACK_SPACE_Y);
} else if ( rightColors > 0 ) {
  if ( rightPlaces > 0 ) {
    gr.setColor(Color.magenta);
    gr.fillOval(FEEDBACK_BASE_X,y,
        FEEDBACK_WIDTH, FEEDBACK_HEIGHT);
    if ( rightPlaces > 1 ) {
      gr.fillOval(FEEDBACK_BASE_X+FEEDBACK_WIDTH
          +FEEDBACK_SPACE_X,
      y,FEEDBACK_WIDTH, FEEDBACK_HEIGHT);
    }
  }
  gr.setColor(Color.black);
  gr.drawOval(FEEDBACK_BASE_X,y,
      FEEDBACK_WIDTH, FEEDBACK_HEIGHT);
  if ( rightColors > 1 ) {
    gr.drawOval(FEEDBACK_BASE_X+FEEDBACK_WIDTH
        +FEEDBACK_SPACE_X,
    y,FEEDBACK_WIDTH, FEEDBACK_HEIGHT);
  }
  if ( rightColors > 2 ) {
    gr.drawOval(FEEDBACK_BASE_X,
        y+FEEDBACK_HEIGHT+FEEDBACK_SPACE_Y,
    FEEDBACK_WIDTH, FEEDBACK_HEIGHT);
  }
  if ( rightColors > 3 ) {
    gr.drawOval(FEEDBACK_BASE_X+FEEDBACK_WIDTH
        +FEEDBACK_SPACE_X,
    y+FEEDBACK_HEIGHT+FEEDBACK_SPACE_Y,
    FEEDBACK_WIDTH, FEEDBACK_HEIGHT);
  }
}
/** Updates the given rectangle with the given color. If all of the rectangles on the currRow are set, then the submitButton is created. */
private void dropColor( ColoredRectangle droppee, Color color ) {
    droppee.color = color;
    droppee.drawColor = color;
    setRectangles.add(droppee);
    updateListeners(droppee);
    boolean allSet = true;
    for ( int col = 0; col < numPegs; ++col ) {
        ColoredRectangle cr = spaces[currRow][col];
        if ( null == cr.color ) {
            allSet = false;
            break;
        }
    }
    if ( allSet ) {
        int y = BASE_Y + currRow*(RECT_HEIGHT+SPACE_Y);
        submitButton = new Rectangle(SUBMIT_X,y,
                                       SUBMIT_WIDTH,
                                       SUBMIT_HEIGHT);
    }
}

/** Sets up the dragging of squares around the board and clicking the submit button. */
private void setMouseListening() {
    addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent evt) {
            if ( currRow < 0 || currRow >= numTries || waitingForFeedback )
                return;
            Point pt = evt.getPoint();
            if ( null != submitButton && submitButton.contains(pt) ) {
                submitPressed = true;
                lastMousePoint = pt;
                revalidate();repaint();
            }
            for ( Iterator itr = setRectangles.iterator();
                 itr.hasNext(); ) {
                ColoredRectangle cr = (ColoredRectangle)itr.next();
                if ( cr.contains(pt) ) {
                    draggedRect = getDraggableForColor(cr.color);
                    draggedRect.setLocation(cr.getLocation());
                    lastMousePoint = pt;
                    return;
                }
            }
        }
    });
}
public void mouseReleased(MouseEvent evt) {
    if (intersectedRect != null) {
        dropColor(intersectedRect, draggedRect.color);
        intersectedRect = null;
    }
    if (null != draggedRect) {
        draggedRect = null;
        revalidate();
        repaint();
    }
    if (submitPressed
         && submitButton.contains(evt.getPoint())) {
        updateListeners();
        submitPressed = false;
        waitingForFeedback = true;
        revalidate(); repaint();
    }
    lastMousePoint = null;
}

addMouseListener(new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent evt) {
        if (currRow < 0 || currRow >= numTries)
            return;
        if (null == draggedRect)
            if (null == lastMousePoint)
                return;
        // In the middle of clicking the submit button
        if (submitButton.contains(evt.getPoint())) {
            if (!submitPressed) {
                submitPressed = true;
                revalidate(); repaint();
            }
        } else {
            if (submitPressed) {
                submitPressed = false;
                revalidate(); repaint();
            }
        }
        return;
    }  
    Point newPoint = evt.getPoint();
    int moveX = (int)(newPoint.getX()-lastMousePoint.getX());
    int moveY = (int)(newPoint.getY()-lastMousePoint.getY());
    draggedRect.translate(moveX, moveY);
    lastMousePoint = newPoint;
    ColoredRectangle currIntersected = intersectedRect;
    intersectedRect = null;
    for (int col = 0; col < numPegs; ++col) {
ColoredRectangle cr = spaces[currRow][col];
if ( cr.intersects(draggedRect) ) {
    intersectedRect = cr;
    cr.drawColor = Color.gray;
}
}
if ( null != currIntersected
    && currIntersected != intersectedRect ) {
    currIntersected.drawColor = currIntersected.color;
    revalidate();
    repaint();
}
}
/** Finds an appropriate draggable rectangle for the given * color. This system allows us to avoid building a new * ColoredRectangle for every new drag event. */
private ColoredRectangle getDraggableForColor(Color color) {
    int ind = PGUtil.getIndexForColor(color);
    try {
        return draggableRects[ind];
    } catch ( ArrayIndexOutOfBoundsException aioobe ) {
        return null;
    }
}
/** Tell registered listeners about a submission */
private void updateListeners() {
    int[] guessInts = new int[numPegs];
    for ( int peg = 0; peg < numPegs; ++peg ) {
        guessInts[peg] = PGUtil.getIndexForColor(spaces[currRow][peg].color);
    }
    for ( Iterator itr = listeners.iterator();
         itr.hasNext(); ) {
        ColorPlacementListener cpl =
            (ColorPlacementListener)itr.next();
        cpl.colorsSubmitted(this, currRow, guessInts);
    }
    submitButton = null;
}
/** Tell registered listeners about a change in color for a * space */
private void updateListeners(ColoredRectangle rect) {
    // for ( int col = 0; col < numPegs; ++col ) {
    //    if ( spaces[currRow][col] == rect ) {
    //        Color color = rect.color;
    //    for ( Iterator itr = listeners.iterator();
    //         itr.hasNext(); ) {
    //            ColorPlacementListener cpl =
            //                (ColorPlacementListener)itr.next();
    //            cpl.colorsSubmitted(this, currRow, guessInts);
    //    }
    //}
}
/** Contains the set of rectangles on the canvas that have
* had their colors set. This includes the palette. When
* a drag start event happens, this set of rectangles is
* checked to see if one of them is being dragged. */
private Set setRectangles;
/** Registered listeners*/
private Set listeners;
/** The set of spaces that the player can place a color
* in. These may also be contained in setRectangles. */
private ColoredRectangle[][] spaces;
/** Indicates current row that can be manipulated */
private int currRow;
/** Rectangle currently being dragged */
private ColoredRectangle draggedRect;
/** Set of ColoredRectangles already built and ready for
* dragging */
private ColoredRectangle[] draggableRects;
/** If a draggedRect is intersecting a valid play space, it
* will be remembered with this */
private ColoredRectangle intersectedRect;
/** Used to calculate where to move dragged rectangles */
private Point lastMousePoint;
/** Used to outline the submit button */
private Rectangle submitButton;
/** Use to tell if the submitButton is currently being
* pressed */
private boolean submitPressed;
/** Used to represent the progress feedback to the user */
private int[][] feedback;
/** Font used for painting this component */
private Font canvasFont;
/** Used to display the solution */
private ColoredRectangle[] solution;
/** Message for when the game is over */
private String gameEnded;
/** indicates if the values have been submitted and we’re
* waiting for feedback before we can do anything else */
private boolean waitingForFeedback;
}

/** Class to contain information about color, size and
* location of rectangles on the board. */
class ColoredRectangle extends Rectangle {
    ColoredRectangle(int x, int y, int width, int height, Color color) {
        super(x,y,width,height);
    }
}
GameViewerFrame.java

package view;

import byucc.jhdl.apps.Viewers.ViewerFrame;
import byucc.jhdl.util.cli.CLInterpreter;
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JMenuBar;
import javax.swing.JPanel;

/** This is the main viewer for a Peggie (Mastermind) game. *
 * @author Anthony L. Slade */
public class GameViewerFrame extends ViewerFrame {

    private CLInterpreter interp;
    private GameCanvas canvas;

    public GameViewerFrame(CLInterpreter interp,
                            GameCanvas canvas) {
        super(" //Peggie");
        this.interp = interp;
        this.canvas = canvas;
        buildAndShowFrame();
    }

    protected Container buildContentPane() {
        JPanel panel = new JPanel(new BorderLayout());
        JButton button = new JButton("New Game");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                interp.parseCommand("newgame");
            }
        });
        panel.add(button,BorderLayout.NORTH);
        panel.add(canvas,BorderLayout.CENTER);
        return panel;
    }

    protected JMenuBar buildMenuBar() {
        return null;
    }

    public void actionPerformed(ActionEvent evt) {}
}
PGUtil.java

```java
package view;

import java.awt.Color;
import model.PGCommon;

public class PGUtil implements PGCommon {
    static int getIndexForColor(Color color) {
        for (int col = 0; col < COLOR_SET.length; ++col) {
            if (COLOR_SET[col].equals(color)) {
                return col;
            }
        }
        return -1;
    }
}
```

C.4.3 Guessing Game Application Controller

This section includes the following source code files:

- CommandFeedback.java
- CommandGuess.java
- CommandNewGame.java
- Peggie.java
- PGBroker.java
- ViewListener.java

CommandFeedback.java

```java
package controller;

import byucc.jhdl.util.cli.CLICommand;
import byucc.jhdl.util.cli.CLIException;
import byucc.jhdl.util.cli.CLIInvalidUsageException;
import byucc.jhdl.util.cli.CLIInterpreter;

/** This command tells the PGBroker that it is time to give */
```

220
public class CommandFeedback implements CLICommand {

  public static final String CMD_FEEDBACK = "feedback";

  public CommandFeedback( PGBroker broker,
                           CLInterpreter interp ) {
    this.broker = broker;
    interp.registerCommand( CMD_FEEDBACK, this );
  }

  public Object execute( CLInterpreter parent,
                         String[] args )
    throws CLIException {
    if ( args.length < 3 )
      throw new CLIInvalidUsageException();
    int row = -1;
    int[] guess = new int[args.length-2];
    try {
      row = Integer.parseInt(args[1]);
      for ( int ai = 2; ai < args.length; ++ ai ) {
        guess[ai-2] = Integer.parseInt(args[ai]);
      }
    } catch ( NumberFormatException nfe ) {
      throw new CLIException("Unparsable integer value.");
    }
    broker.setViewFeedback(row, guess);
    return null;
  }

  public String getHelpText( String cmdName ) {
    return "Updates game viewer with feedback for a guess.";
  }

  public String getHelpType( String cmdName ) {
    return "PGBroker";
  }

  public String getUsageText( String cmdName ) {
    return "row guess0 guess1 ...";
  }

  private PGBroker broker;

  CommandGuess.java

  package controller;

  import byucc.jhdl.base.HWSystem;
  import byucc.jhdl.apps.Broker.Broker;
  import byucc.jhdl.apps.util.SimulatorThread;
import byucc.jhdl.apps.Viewers.ViewerFrame;
import byucc.jhdl.util.cli.CLICommand;
import byucc.jhdl.util.cli.CLIException;
import byucc.jhdl.util.cli.CLIInvalidUsageException;
import byucc.jhdl.util.cli.CLInterpreter;
import model.tbPG;
import view.ColorPlacementListener;
import view.GameCanvas;
import view.GameViewerFrame;

/** This command tells the PGBroker to set the hardware
* circuit and the game viewer to register a new guess.
* @author Anthony L. Slade */
public class CommandGuess implements CLICommand {
    public static final String CMD_GUESS = "guess";
    ViewerFrame viewerFrame = null;
    GameCanvas canvas;
    CLInterpreter interp;
    tbPG parent;
    public CommandGuess( PGBroker broker,
            CLInterpreter interp ) {
        this.broker = broker;
        interp.registerCommand( CMD_GUESS, this );
    }
    public Object execute( CLInterpreter parent,
            String[] args )
            throws CLIException {
            if ( args.length < 3 )
                throw new CLIInvalidUsageException();
            int row = -1;
            int[] guess = new int[args.length-2];
            try {
                row = Integer.parseInt(args[1]);
                for ( int ai = 2; ai < args.length; ++ ai ) {
                    guess[ai-2] = Integer.parseInt(args[ai]);
                }
            } catch ( NumberFormatException nfe ) {
                throw new CLIException("Unparsable integer value.");
            }
            broker.setGuess(row, guess);
            return null;
    }
    public String getHelpText( String cmdName ) {
        return "Registers a new guess in the game.";
    }
    public String getHelpType( String cmdName ) {
        return "PGBroker";
    }
    public String getUsageText( String cmdName ) {
        return "row guess0 guess1 ...";
    }
    private PGBroker broker;
**CommandNewGame.java**

```java
package controller;

import byucc.jhdl.base.HWSystem;
import byucc.jhdl.apps.Broker.Broker;
import byucc.jhdl.apps.util.SimulatorThread;
import byucc.jhdl.apps.Viewers.ViewerFrame;
import byucc.jhdl.util.cli.CLICommand;
import byucc.jhdl.util.cli.CLIEcision;
import byucc.jhdl.util.cli.CLInterpreter;
import model.tbPG;
import view.ColorPlacementListener;
import view.GameCanvas;
import view.GameViewerFrame;

/** This command tells the PGBroker to set the hardware
 circuit and the game viewer to start a new game.
 * @author Anthony L. Slade */
public class CommandNewGame implements CLICommand {
    public static final String CMD_NEW_GAME = "newgame";
    ViewerFrame viewerFrame = null;
    GameCanvas canvas;
    CLInterpreter interp;
    tbPG parent;

    public CommandNewGame( PGBroker broker,
            CLInterpreter interp ) {
        this.broker = broker;
        interp.registerCommand( CMD_NEW_GAME, this );
    }

    public Object execute( CLInterpreter parent,
            String[] args )
            throws CLIEcision {
        broker.newGame();
        return null;
    }

    public String getHelpText( String cmdName ) {
        return "Starts a new game";
    }

    public String getHelpType( String cmdName ) {
        return "PGBroker";
    }

    public String getUsageText( String cmdName ) {
        return ""
    }

    private PGBroker broker;
}

**Peggie.java**

import byucc.jhdl.base.HWSystem;
```

223
import model.tbPG;
import controller.PGBroker;

public class Peggie {
    public static void main(String argv[]) {
        HWSystem hws = new HWSystem();
        tbPG tb = new tbPG( hws );
        new PGBroker(tb,hws);
    }
}

PGBroker.java

package controller;

import byucc.jhdl.base.HWSystem;
import byucc.jhdl.apps.Broker.Broker;
import byucc.jhdl.apps.Stimulator.Stimulator;
import byucc.jhdl.apps.util.SimulatorThread;
import byucc.jhdl.apps.Viewers.ViewerFrame;
import byucc.jhdl.util.cli.CLInterpreter;
import model.tbPG;
import view.ColorPlacementListener;
import view.GameCanvas;
import view.GameViewerFrame;

/** The controller portion of the Peggie (Mastermind) game.
 * @author Anthony L. Slade */
public class PGBroker extends Broker {
    public PGBroker( tbPG parent, HWSystem system ) {
        super(system,new CLInterpreter());
        this.parent = parent;
        openGame();
    }

    protected void registerCLICommands(HWSystem system,
            CLInterpreter interp) {
        super.registerCLICommands(system,interp);
        new CommandNewGame(this, interp);
        new CommandGuess(this, interp);
        // registers put and get commands:
        Stimulator.registerCLICommands(interp);
        // remember the interpreter
        this.interp = interp;
        // create game view in this method for convenience only
        canvas = new GameCanvas();
        // make sure game view listener has command interpreter
        canvas.addListener(new ViewListener(interp));
    }
}
void newGame() {
    canvas.newGame();
    interp.parseCommandNoHistory("put_newGame_1");
    interp.parseCommandNoHistory("cycle_2");
    interp.parseCommandNoHistory("put_newGame_0");
    interp.parseCommandNoHistory("cycle_1");
}

private void openGame() {
    if (null != viewerFrame) {
        viewerFrame.setState(viewerFrame.NORMAL);
        viewerFrame.show();
        viewerFrame.toFront();
    } else {
        viewerFrame = new GameViewerFrame(interp, canvas);
    }
}

/** This class is used to defer updating the GameCanvas * until after all simulation/execution has occurred. */
private class CanvasUpdateThread extends SimulatorThread {
    CanvasUpdateThread(GameCanvas source, int row, int[] guess) {
        this.source = source;
        this.row = row;
        this.guess = guess;
    }

    public Object construct() { return null; }

    /** This method is called back on the Swing event thread * after the main thread is finished executing. */
    public void finished() {
        System.err.println("tbPG: updating GameCanvas now.");
        source.setFeedback(row,
            parent.getNumRightColors(),
            parent.getNumRightPlaces());
        if (parent.getGameOver()) {
            source.gameEnded(guess, false);
        } else if (parent.getGameWon()) {
            source.gameEnded(getSolution(), true);
        } else {
            source.setCurrentRow(row+1);
            source.revalidate(); source.repaint();
        }
        // make sure that the super class’ finished method is called too
        super.finished();
    }

    private GameCanvas source;
    private int row;
    private int[] guess;
}

void setGuess(int row, int[] guess) {
    for (int gi = 0; gi < guess.length; ++gi) {

viewFeedback(int row, int[] guess) {
    interp.parseCommandNoHistory("put_statusAddress\n"+row);
    interp.parseCommandNoHistory("cycle_2");
    // This thread will update the canvas, but only after any
    // previous Simulator threads (puts and cycles) have
    // completed.
    (new CanvasUpdateThread(canvas, row, guess)).start();
}

private int[] getSolution() {
    interp.parseCommandNoHistory("put_statusAddress_10");
    interp.parseCommandNoHistory("cycle_2");
    // this get is here so that we stall for the simulator
    // thread queue
    interp.parseCommandNoHistory("get_statusAddress");
    return parent.getGuess();
}

ViewListener.java

package controller;

import byucc.jhdl.util.cli.CLInterpreter;
import view.ColorPlacementListener;
import view.GameCanvas;

/** This listener receives events from the game view and
 * translates them into CLICommands to execute.
 * @author Anthony L. Slade */
public class ViewListener implements ColorPlacementListener {

    public ViewListener( CLInterpreter interp ) {
        this.interp = interp;
    }

    public void colorsSubmitted( GameCanvas source,
```java
    int row, int[] guess ) {
        StringBuffer cmdParams = new StringBuffer("_");
        cmdParams.append(row);
        for (int gi = 0; gi < guess.length; ++gi) {
            cmdParams.append("_"); cmdParams.append(guess[gi]);
        }
        interp.parseCommand(CommandGuess.CMD_GUESS
                + cmdParams.toString());
        interp.parseCommand(CommandFeedback.CMD_FEEDBACK
                + cmdParams.toString());
    }

    private CLIInterpreter interp;
```
Bibliography


