JHDLBits: An Open-Source Model for FPGA Design Automation

Alexandra Vanessa Poetter

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Dr. Peter M. Athanas, Chair
Dr. Cameron D. Patterson
Dr. Thomas L. Martin

August 23, 2004
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: JHDLBits, JHDL, JBits, FPGA, Design Automation,
Hardware Description Language

JHDLBits: An Open-Source Model for FPGA Design Automation

Alexandra Vanessa Poetter

Abstract

Today's Field Programmable Gate Array (FPGA) research community could use an extensible tool flow enabling designers to examine new algorithms and new methods of interacting with FPGA configurations. One such flow is JHDLBits, which integrates two prominent FPGA design environments: JHDL and JBits. JHDLBits offers the low-level access and control provided by JBits with the high-level structural circuit design of JHDL. Furthermore, the JHDLBits flow provides greater control of resource manipulation, placement, and routing, and gives researchers a *sandbox* to explore advanced interactions with FPGA configurations. This thesis presents the overall architecture of the open-source JHDLBits project. Details are provided on how the core components – JHDL, JBits3 for Virtex-II, the ADB connectivity database, and VTsim, a Virtex-II device simulator – are linked together to provide an integrated design environment. Strategies and philosophies of the open source movement are also examined to successfully establish the support for and involvement of the FPGA research community throughout the JHDLBits open source endeavor.

To my wonderful husband, Kevin, for always believing in me and showing me what a wonderful place Blacksburg is.

Acknowledgements

Thanks to my academic advisor, Dr. Athanas, for giving me the opportunity to join the Configurable Computing Lab and for all of his guidance throughout the JHDLBits project and during my graduate school career at Virginia Tech. I cannot thank him enough for the opportunities I have been allowed such as presenting JHDLBits at the FPL and FCCM conferences. Thanks for teaching an insightful Configurable Computing course which originally sparked my interest in JBits.

Thanks to Dr. Patterson for being a member of my committee, and for sharing his expert knowledge of FPGAs. Many of the decisions made during the JHDLBits project would not have occurred without his valuable input and guidance.

Thanks to Dr. Martin for being on my committee, and for teaching a great Digital Design II course. I would have never become interested in FPGA design had it not been for his course.

Thanks to my husband Kevin for encouraging me to go to graduate school, for his constant support throughout the last couple of years, and for always believing in me. I could not have accomplished this without him. Thanks for all the proofreading help he offered while I was writing this thesis.

Thanks to my parents, Rainer and Doris, for always encouraging me to believe I can achieve anything I set my mind to. Thanks for providing such rich and cultured experiences while growing up all over the world.

Thanks to my sister, Kathrin, for being my best friend, and for always listening on the phone.

Thanks to the rest of my family in Maryland, New York, and in Germany.

Thanks to Jesse Hunter for being a great friend, fellow JHDLBits team member, fellow CEL TA, and homework buddy for numerous classes. Graduate school would not have been the

same without him!

Thanks to Neil Steiner for always being there as a "consultant", even though he did not want to become involved. He has been a great inspiration and wealth of knowledge throughout this project.

Thanks to the JHDLBits team. Best of luck in the continued development of JHDLBits.

Thanks to current and former members of the CCM lab: Neil, Jesse, Tony, Stephen, Jon, Aditya, Dong Kwan, Josh, Dave, and Ryan. Best of luck in all future endeavors!

Thanks to all of my close friends in Blacksburg and in Colorado.

The JHDLBits project is supported by a grant from Xilinx, Inc.; JBits (Xilinx and Virginia Tech) and JHDL (Brigham Young University) were both funded under the DARPA ACS program.

Table of Contents

List of Figures

List of Tables

Glossary

A glossary of acronyms used throughout this thesis is presented below. The list is organized alphabetically.

API:	Application Programming Interface
ASIC:	Application Specific Integrated Circuit
BRAM:	Block SelectRAM
BSD:	Berkeley Software Distribution
CAD:	Computer Aided Design
CLB:	Configurable Logic Block
DARPA:	Defense Advanced Research Projects Agency
DCM:	Digital Clock Manager
DEC:	Digital Equipment Coporation
EDIF:	Electronic Design Interchange Format
FPGA:	Field Programmable Gate Array
FSF:	Free Software Foundation
GPL:	GNU General Public License
GUI:	Graphical User Interface
HDL:	Hardware Description Language
IDE:	Integrated Development Environment
IOB:	Input/Output Block
JERC:	Java Environment for Reconfigurable Computing
LE:	Logic Element
LGPL:	GNU Library or "Lesser" Public License
LSB:	Least Significant Bit
LUT:	Look Up Table
MSB:	Most Significant Bit

OSD: Open Source Definition
OSI: Open Source Inititative
OSS: Open Source Software
PCB: Printed Circuit Board
PLD: Programmable Logic Device
QUIP: Quartus University Interface Program
RAM: Random Access Memory
RTR: Run-Time Reconfiguration
UCF: User Constraints File
XHWIF: Xilinx HardWare InterFace

Chapter 1

Introduction

1.1 Overview

Designers involved in Field-Programmable Gate Array (FPGA) related research often require environments for exploring and evaluating new tools, new algorithms, and new ways to interact with FPGA configurations. This has often proven to be a difficult task. Exploratory environments have been created, such as VPR by Betz and Rose [1], that provide realistic models of FPGAs. With the infrastructure created by VPR, researchers can simulate the effects of placement enhancements on wire length. However, FPGA vendors tend to be secretive on the low-level architectural details of their products, making empirical confirmation of the simulation results impractical. Many languages, IDEs, and compilers have emerged in recent years that offer interesting environments for creating FPGA configurations, but most rely on the FPGA vendors' implementation flows to map, place, and route the final design.

The authors of JBits [2] address this problem to a certain degree by providing an Application Programming Interface (API) into the FPGA configuration. As a result, researchers are empowered with the ability to manipulate FPGA resources at the lowest level. For example, JBits allows researchers to develop their own FPGA placer and test it on real devices. JBits also enables researchers to interact with FPGAs in ways normally prohibited by the vendor's traditional mainstream tool flow.

While JBits is in many ways an *enabling technology* for exploring non-traditional uses of FPGAs, the abstraction presented to the developer is at a fairly low level, and mostly requires detailed architectural knowledge of the FPGA. In contrast, tools such as Brigham Young's JHDL [3] use a much higher level of abstraction, often making it a more conducive environment for large FPGA designs. JHDL is a structurally based Hardware Description Language (HDL) implemented within Java to describe the logic and connections in a digital logic circuit. The JHDL API provides a set of tools for debugging, simulating, and interfacing with a circuit created out of wires and basic elements.

1.2 Motivation

In this thesis, a software design automation package called JHDLBits is presented. JHDLBits strives to merge the prominent features of the JBits and JHDL research environments. JHDLBits blends the low-level control of JBits with the high-level design abstractions of JHDL. For instance, users can take advantage of programming specific FPGA resources by using JBits libraries without having to work entirely at the configuration level. Furthermore, full control over placement and routing is still possible. The JHDLBits project consists of a collection of tightly integrated components that together provide a means for creating, manipulating, and debugging FPGA configurations. More importantly, because most of the components of this project are open-source, researchers investigating architecture-specific placers, planners, routers, and compilers have the advantage of replacing the stock JHDLBits components at will.

JHDLBits integrates the JBits low-level bit manipulation capabilities into the JHDL integrated development environment (IDE). The extensible JHDL netlister has been modified to produce FPGA configurations directly. The JHDL debugger communicates to a target FPGA through the JBits Xilinx HardWare InterFace (XHWIF) [4]. JHDL is naturally extended so that instead of generating an Electronic Design Interchange Format (EDIF) file, which is run through the traditional vendor tools to generate a configuration file, the primitive and net information is extracted into a database that is easy for JBits to process. A

configuration file is created by elaborating, placing, and routing the corresponding JBits primitives all within the JHDLBits IDE.

1.3 Contributions

My contributions to the JHDLBits project are as follows:

- JHDLBits core code (JHDLBits Extractor, JHDLBits TechMapper, etc.) which is the common base for all JHDLBits designs.
- Creation of some of the primitives in the JHDLBits Primitive Library which serve as the building blocks for all JHDLBits designs.
- Tie-in of VTsim, a Virtex-II device simulator, which is tightly integrated into the JHDLBits flow.
- Research of the open source movement to determine the strategies for JHDLBits project continuity.
- Preparation of the JHDLBits website.
- Involved with testing of JHDLBits primitives on FPGA hardware.

1.4 Open Source Model

JHDLBits is an open-source endeavor striving to support and involve the FPGA research community by providing a tool to interact with FPGA configurations. The JHDLBits IDE will be hosted on SourceForge [5]. One part of the IDE is the JHDLBits core responsible for extracting information from a JHDL design and then using JBits calls to generate a configuration file. The rest of the IDE consists of components such as a placer, router, and device simulator. These components are easily extensible and can be replaced, allowing a user to plug in their own components as desired.

JHDLBits is entirely written in Java allowing the easy integration of JHDL and JBits. This integration allows the user to create designs using the Hardware Description Language and Debugger included with JHDL. The JHDLBits device simulator has also been integrated into the JHDL debugging environment. Figure 1.1 shows the relationships of the open source components that collectively make up the JHDLBits project.

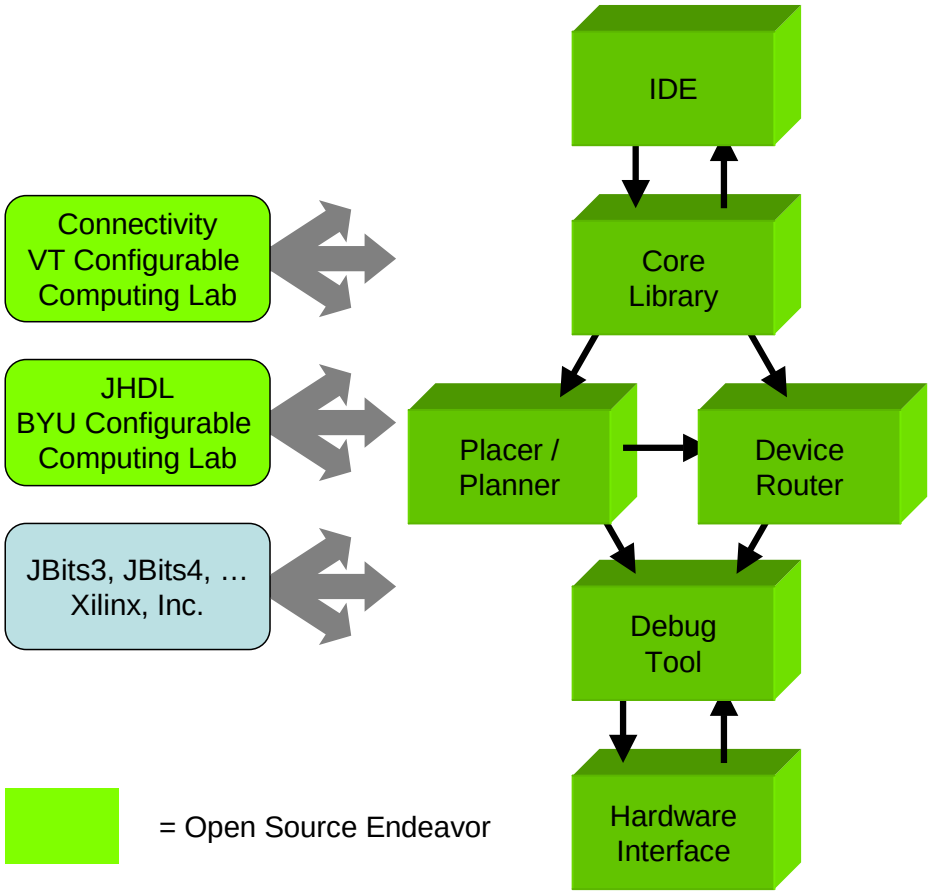

Figure 1.1: Relationships of the Open Source Constituents of JHDLBits

The history of the open source movement, along with strategies and philosophies that make a project successful, are also a focus of the JHDLBits development process. Open source projects such as Berkeley Software Distribution (BSD) Unix and Linux can be used as models

to influence the decision made by the JHDLBits team in preparation for an open source release. Following some of these strategies should pave the way for successful involvement from the research community.

1.5 Thesis Organization

The thesis is organized into seven chapters describing the background and details of the JHDLBits open source project. Chapter 2 provides the reader with history and background information about FPGAs, along with describing the related work of JHDL, JBits, ADB, and VTsim in the context of JHDLBits. Chapter 3 presents the philosophies and strategies of the open source movement to ensure a successful open source endeavor. Chapter 4 introduces the core JHDLBits components, and details how they are linked together to form a cohesive design environment. Chapter 5 provides the user with steps to create a high level design using JHDL, and gives examples of how JHDLBits is invoked. Chapter 6 details the proof of concept by showing JHDLBits output, along with describing how a user can take advantage of the extensibility of JHDLBits components. Chapter 7 summarizes the thesis, and discusses future directions of JHDLBits.

Chapter 2

Background and Related Work

2.1 Overview

This chapter presents a description of what an FPGA is and how it is used, along with tools currently available to enable FPGA development. Examples of such tools are JBits and JHDL, which can be brought together to form an integrated design environment in JHDLBits. Additional related information is also included on other components which are part of the JHDLBits project.

2.2 FPGA Information

Traditionally, custom logic designs required the use of an Application Specific Integrated Circuit (ASIC) comprised of discrete logic gates and flip-flops statically interconnected. Despite their high speed and high density, they are expensive to design and require a lengthy remanufacturing process if a change is required. However, the development of Programmable Logic Devices (PLDs) allow the functionality of a part to change after manufacturing. These designs are built from complex devices that consist of an uncommitted array of logic gates and memory elements that can be configured by the user to implement a complete logic de-

sign [6]. An FPGA is a PLD featuring very high logic capacity and an extensive interconnect network. Some of the salient features exhibited by FPGAs are [7]:

- The masking layers used in the fabrication process are not customized for a particular design implementation.
- There is a method to program and possibly re-program the basic logic blocks and interconnects.
- The core is a regular array of programmable logic cells.
- A routing matrix surrounds the basic logic blocks.
- Programmable I/Os run along the perimeter of the array.
- It exhibits extremely small design turn-around times.

The architecture of an FPGA consists of arrays of basic logic blocks. Each array unit includes a logic block and interfaces to the routing resources through a connection block (CB) and switch box (SB). This array structure is replicated in a tile format, as shown in Figure 2.1 reproduced from [8].

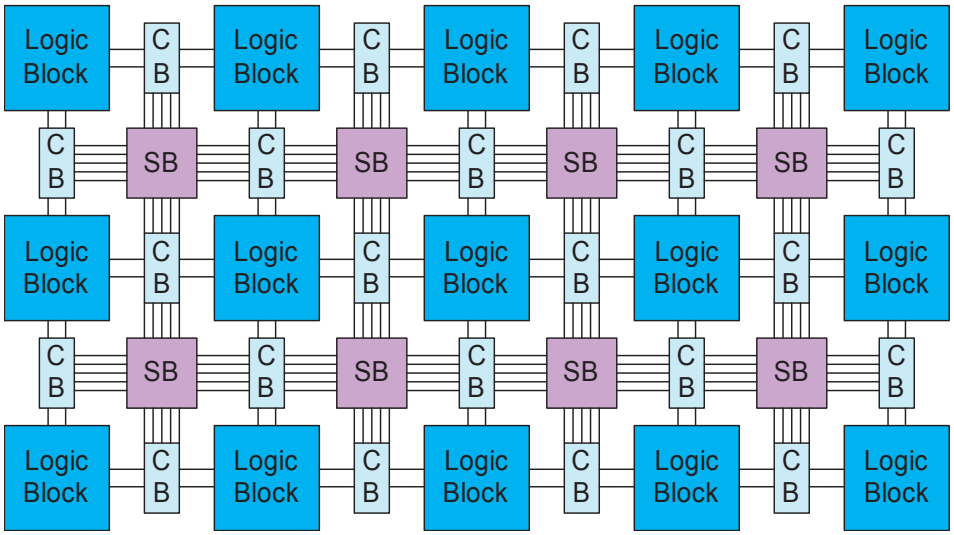

Figure 2.1: FPGA Tile Structure

2.2.1 Xilinx FPGAs

Xilinx is the largest developer and manufacturer of FPGAs. The first Xilinx FPGA family, the XC2000 series, was introduced in 1985. Since 1985, many more generations such as the XC3000, XC4000, Virtex, Spartan-II, Spartan-3, Virtex-II, Virtex-II Pro, and Virtex-4 families have been introduced. Altera, Xilinx's current primary competitor in the marketplace, has offered a similar progression of FPGA offerings. Even though JHDLBits could be applied to other FPGA architectures, Xilinx FPGAs were chosen out of convenience.

Because JHDLBits works with the Xilinx Virtex-II family, this series will be focused on. Table 2.1, as modified from [9], gives an overview of the number of logic cells, multipliers, memory, and input/output (I/O) channels available to the user in the various Virtex-II devices.

Table 2.1: Xilinx Virtex-II Platform FPGA

Feature/ Product	**Logic Cells**	**BRAM (Kbits)**	**18x18 Multipliers**	**Digital Clk Mgmt Blks**	**Max Dist RAM Kb**	**Max Avail User I/O**
XC2V40	576	72	4	4	8	88
XC2V80	1,152	144	8	4	16	120
XC2V250	3,456	432	24	8	48	200
XC2V500	6,912	576	32	8	96	264
XC2V1000	11,520	720	40	8	160	432
XC2V1500	17,280	864	48	8	240	528
XC2V2000	24,192	1,008	56	8	336	624
XC2V3000	32,256	1,728	96	12	448	720
XC2V4000	51,840	2,160	120	12	720	912
XC2V6000	76,032	2,592	144	12	1,056	1,104
XC2V8000	104,882	3,024	168	12	1,456	1,108

The Virtex-II architecture features various functional tile types arranged in an array structure, as shown in Figure 2.2 taken from [10]. The tile types consist of Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), Block SelectRAM (BRAM) memory modules, Multiplier Blocks, and Digital Clock Manager (DCM) blocks.

The CLBs are made up of four logic slices and two 3-state buffers. Each logic slice is equivalent and contains two function generators (F&G). The function generators are configurable as 4-input look-up tables (LUTs), as 16-bit shift registers, or as 16-bit distributed SelectRAM memory. Each CLB has internal fast interconnect and externally connects to a switch matrix to access general routing resources. The IOBs interface between the package pins and the internal configurable logic. The Block SelectRAM memory modules provide 18Kbit of dual-port RAM, and the Mulitplier Block is a dedicated 18x18-bit multiplier. The DCM blocks along with the global clock multiplexer provide design solutions for high-speed clocking [10].

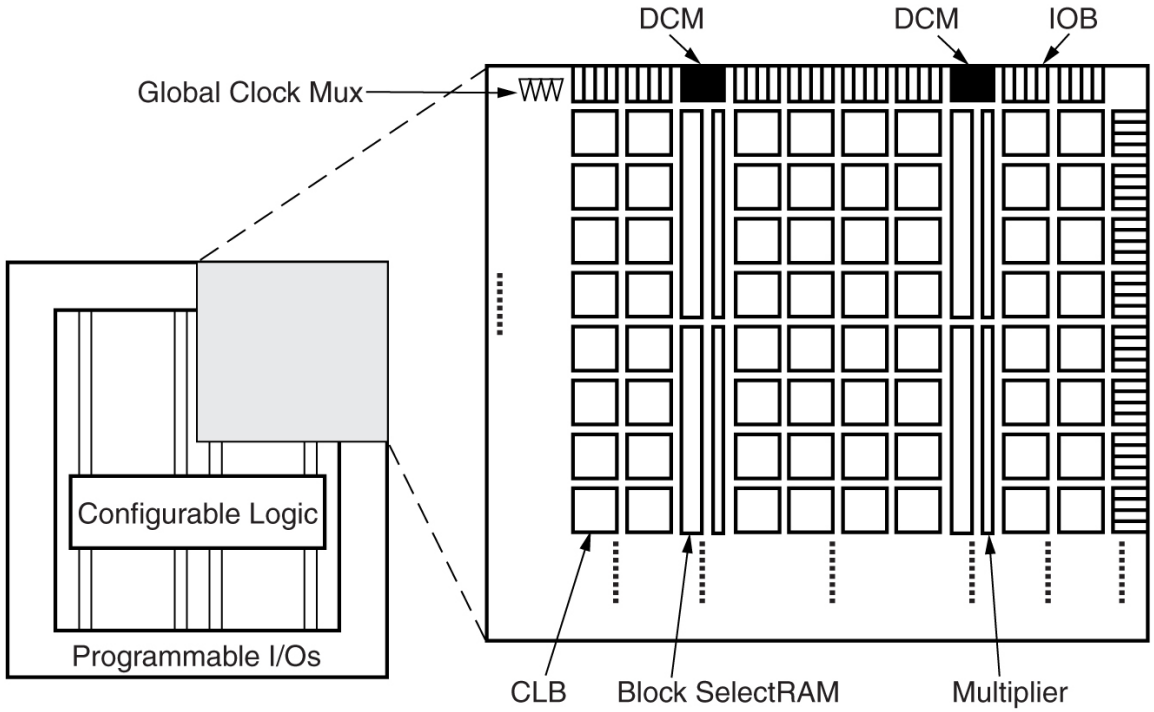

Figure 2.2: Virtex-II Architecture Overview

2.2.2 Bitstream Generation

To define the behavior of the FPGA hardware, a proprietary software data file is downloaded to the chip in the form of a bitstream. A bitstream is a binary sequence used to control the

on-off state of various pass transistors and transmission gates (active switches).

A programming image (bitstream) for Xilinx Virtex-II devices is typically generated through the use of various Xilinx tools. The design flow usually starts with a user-created VHDL or Verilog source file describing the functionality to be implemented in the FPGA hardware. Xilinx ISE [11] design tools allow the user to synthesize, translate, map, route, and generate a bitstream for a design. Alternatively, a source file can be synthesized using Synplicity's Synplify Pro [12] tool suite. The resulting file can then be imported into the Xilinx ISE tools to map, place, route, and generate a bitstream for a design.

Once generated, the bitstream can be loaded into the configuration memory of the FPGA hardware.

2.3 History of Automated FPGA Design Tools

2.3.1 JERC

The Java Environment for Reconfigurable Computing (JERC) is the predecessor to JBits. JERC [13] is a software environment consisting of a standard Java compiler and a set of libraries for reconfigurable coprocessor applications. The platform for the JERC environment is the Xilinx XC6200 Development System [14], shown in Figure 2.3, which consists of a Xilinx XC6216 FPGA on a PCI board.

JERC is based on multiple levels of abstraction to help a user construct circuit designs. The lowest level provides the ability to program all of the reconfigurable logic cells and routing switches, including the clock routing in the XC6200. However, this layer requires the user to have a detailed knowledge of the internal FPGA architecture, and is intended to only be used to build up the remaining layers. The next layer of abstraction is made up of basic logic elements such as AND, OR, etc. along with registers and flip-flops. The clock routing is also abstracted in this layer.

JERC was one of the tools created when Xilinx released the XC6200DS, the first open FPGA

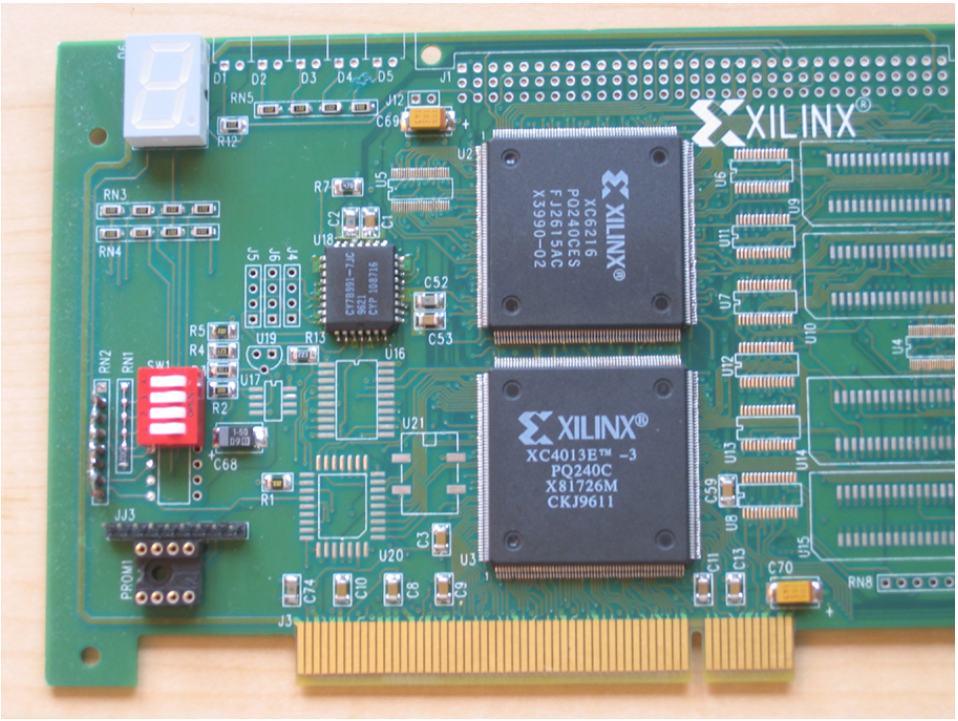

Figure 2.3: Xilinx XC6200 Development Board

architecture that allowed for reconfigurable computing.

2.3.2 Fast Integrated Tools for Circuit Design with FPGAs

Fast integrated tools for circuit design with FPGAs [15] have been developed at the Swiss Federal Institute of Technology. The system design tightly integrates the FPGA design process in a single environment, starting with the initial circuit specification in an HDL and finishing with the bitstream generation. The front-end consists of a technology-independent framework, while the back-end is technology-dependent. Back-end processes have been developed for the Xilinx XC6200 architecture and for the Atmel AT6000 [16] architecture.

As the basis of the tool, a universal circuit representation (data structure) is defined in the core of the framework. This means that several tools such as the HDL compiler, placer, router, and bitstream generator all share the same circuit data structure. The framework

for the design specification uses a compiler for the hardware description language Lola [17]. Lola supports parameterizable cores such as N-bit Adders and allows the user to constrain placement of circuit components. Other features of the framework include a layout editor, design checker, and automatic placer. A designer also has influence over the routing process, by allowing the user to constrain the routing to certain areas of the chip. The XC6200 back-end process consists of technology-dependent mapping, placement, and routing.

During analysis, the resulting design cycle is one to two orders of magnitude faster than the vendor-supplied tools. This is attributed to the universal data structure, which allows tools to run faster and with lower memory usage. Using the framework above gives the user tight control of the FPGA implementation process.

2.3.3 Quartus II University Interface Program

The Quartus II University Interface Program (QUIP) [18] developed by Altera is a toolkit that enables researches to access the Quartus II CAD suite at different stages of the implementation flow. Quartus II is Altera's mainstream FPGA tool flow. The QUIP toolkit allows users to integrate their own CAD tool components and ideas into a complete FPGA CAD flow.

QUIP describes Altera's FPGA devices, the interfaces by which data can be input at various points in the CAD flow, and data formats in which data can be output from the Quartus II software. Following are examples of the CAD tool components a user can build into the mainstream flow:

- Replace the Quartus II HDL elaboration with your own HDL elaboration.
- Replace the Quartus II technology mapping algorithm with a new one.
- Replace the Quartus II placement algorithm with your own.
- Add a floorplanner to the CAD flow.
- Add a global router to the CAD flow.

- Perform physical synthesis.
- Develop an engineering change order (ECO) flow.

Advantages of QUIP include allowing the user to plug in circuits written in various hardware description languages such as VHDL and Verilog, or even in higher-level formats like Simulink and Matlab. The Quartus II software flow also gives the user access to accurate timing analysis and the programming files (bitstreams) for real devices. Altera is releasing all the details of their devices so users can investigate CAD algorithms for the more complex features that are normally abstracted away.

2.4 JHDL

The Configurable Computing Laboratory at Brigham Young University has developed a structural Java-based Hardware Description Language (JHDL). JHDL is an open-source set of FPGA development tools that allow a user to design the structure and layout of a circuit, debug and then netlist the circuit, as represented in Figure 2.4. A netlist for a design contains information on the primitive blocks of a design and corresponding connections. Java was chosen because it is easy to use, object-oriented, platform-independent, has built-in documentation capabilities, and has a rich set of graphical user interface (GUI) APIs that are integral to the language. The primary distinction of JHDL is the creation of a single integrated API that allows the designer to express circuit organizations that dynamically change over time [3].

JHDL is a structural HDL - that is, circuits are described by structurally instantiating lower-level building blocks. JHDL provides object classes used to build up circuit structure. These include [19]:

- Static cells, such as Boolean gates, registers, etc.
- Parameterizable modules, such as multipliers, counters, etc.
- Generic, platform-independent APIs.

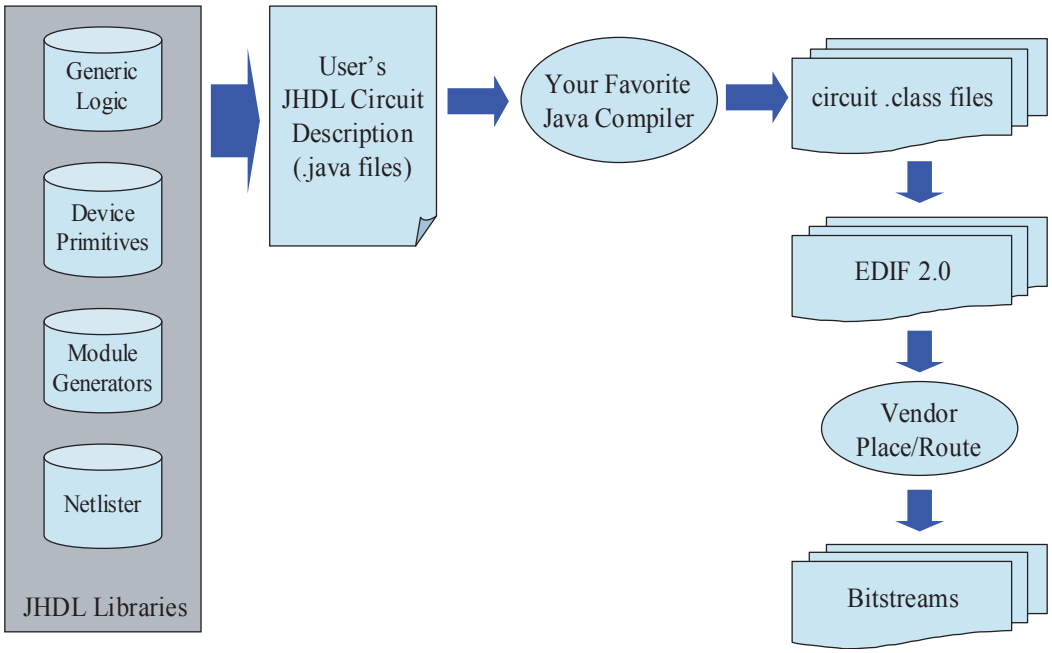

Figure 2.4: JHDL Design Flow

These object classes are used for creating basic circuit elements and the corresponding interconnect. In JHDL, two basic Java classes form the basis for all circuits: `Logic` and `Wire`. Designers create a new logic cell in their design by extending `Logic` (creating a new class), defining a cell interface, and defining the architecture of the cell. Wires support an API for creation and manipulation: users can create single- or multi-bit wires, and concatenate or extract wires from existing wires [20].

The body of a design instantiates predefined circuit modules selected from the JHDL design libraries. JHDL libraries offer the user a number of levels of abstraction ranging from low-level design using primitive library elements to higher-level design using module generators. The first level of abstraction `(byucc.jhdl.Xilinx)` instantiates exact circuit primitives for the FPGA technology targeted. This creates the best performance results and packing density for the part, but is not portable to other FPGA technologies. The second level of abstraction `(byucc.jhdl.Logic)` is a layer of technology-independent routines on top of the primitive libraries that can be targeted and optimized to a specific technology at build time. This provides the user a functional interface for creating logic, while hiding the tech-

nology dependencies. The third level of abstraction (`byucc.jhdl.Logic.Modules`) layers on top of `Logic` by providing more predefined circuit building blocks for users to incorporate. This package consists of counters, multipliers, memories, etc. that are portable to any technology, but do not take advantage of the technology-specific features for optimization. The fourth level of abstraction (`byucc.jhdl.Logic.Virtex2.Modules`) takes advantage of any technology-specific features to give the design the smallest/fastest circuit possible. The fifth level of abstraction (`byucc.Contrib.Logic`) allows JHDL users to contribute their own packages to be included in future releases [21].

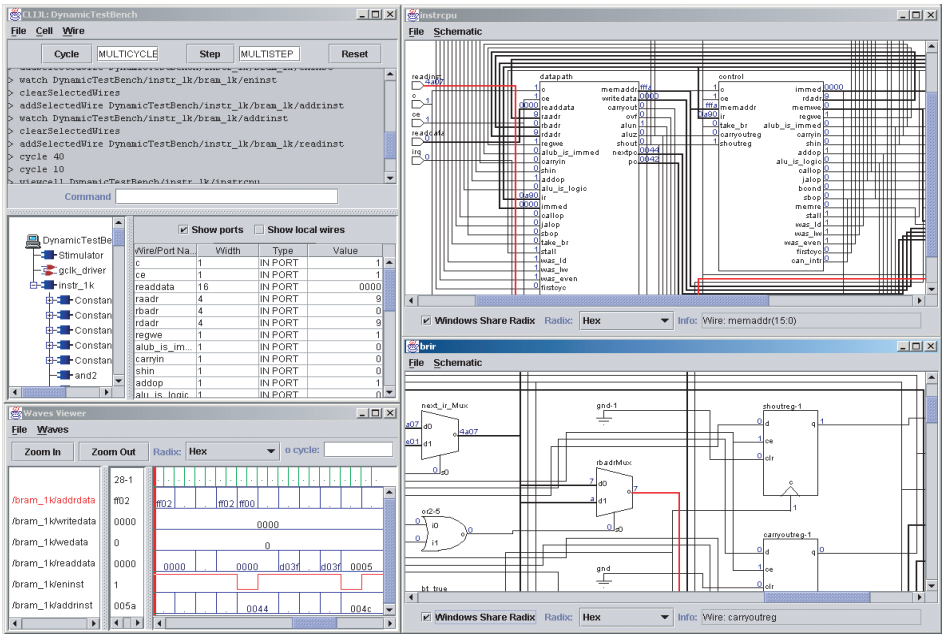

Figure 2.5: JHDL Graphical User Interface Screen Shot

An important feature of JHDL is its ability to operate in either simulation or hardware mode. In simulation mode, the values of all elements in the circuit data structure are computed by the JHDL simulator. In hardware mode, starting execution of the circuit downloads a bitstream to the FPGA platform, and stepping the clock triggers a clock step on the actual hardware. Between clock steps, bitstream readback extracts the hardware state, which is back annotated into the circuit data structure for graphical display. As a result, the same

GUI interface and tools can be used for debugging in either in simulation mode or hardware execution mode. This simplifies the transition from simulation to hardware debug, and streamlines the development process. Figure 2.5 presents a screen capture of the JHDL GUI used in either simulation or hardware mode.

2.5 JBits

JBits is a collection of Java classes that provide an API to access the Xilinx FPGA configuration bitstream, namely every configurable resource in the hardware. The interface currently operates on either bitstreams generated by Xilinx design tools, or on bitstreams extracted from the FPGA hardware. The configurable resources accessed by JBits include look-up tables (LUTs), flip-flops, and routing elements [2]. These device resources can be programmed and probed individually at run-time, even with the FPGA active in a working system. Through this mechanism, JBits supports the run-time observation and reconfiguration of Xilinx FPGAs. The interface to the hardware is implemented by XHWIF, the Xilinx HardWare InterFace standard. XHWIF [4] is a Java interface for communicating with FPGA-based Printed Circuit Boards (PCBs), i.e. to describe the board and to send data on and off the board. Methods are included for reading and writing bitstreams, along with describing the kinds and numbers of FPGAs on the board. The entire JBits API suite provides FPGA designers with a good alternative to some of the standard vendor-based design flows.

Multiple versions of JBits have been released since 1998 to support various FPGA families. The JBits3 SDK [22] is the latest release from Xilinx and provides support for the Virtex-II FPGA family. This API allows access to all of the bit-level configurable resources, but unlike previous versions does not include a device simulator, router, or support for placement. Prior versions also included higher levels of abstraction which were made of parameterizable cores such as adders, counters, multipliers, and standard logic functions. The JBits design flow

starts with an input bitstream and user developed Java code made up of calls to JBits libraries. This is similar to circuits written in JHDL. Next, the code is compiled, executed, and the output is downloaded to the FPGA in the form of a bitstream. Figure 2.6 illustrates the JBits design flow.

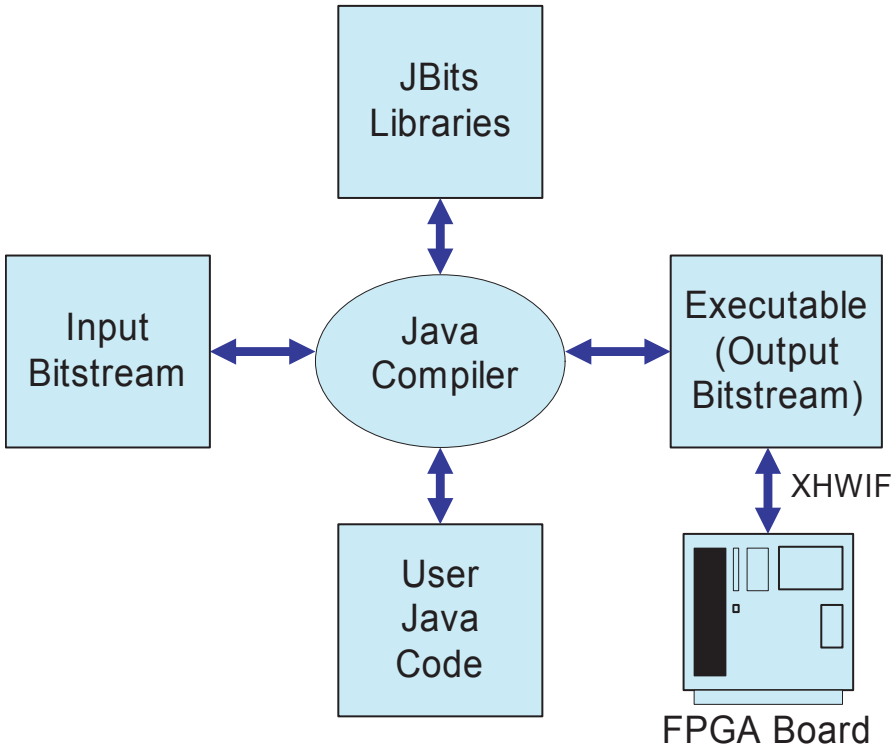

Figure 2.6: JBits Design Flow

Although JBits is an alternative to the standard design flow, there are still some drawbacks with the API. JBits is quite low-level in nature, such that all the FPGA resources must be explicitly stated in the design source code. As a result, JBits is more useful for very structured circuits. The low-level resources programmed in JBits also require the user to have a very detailed understanding of the internal architecture of the hardware. The authors of JBits dealt with this issue in prior versions with higher levels of abstraction or cores so the user did not have to design the circuit at the lowest bit-level, but these abstractions are not included in JBits3. The JBits interface also does not provide timing analysis.

2.6 ADB

ADB is an Alternate Wire Database that can be used for routing, tracing, and browsing in Xilinx Virtex, Virtex-E, Virtex-II, and Virtex-II Pro FPGAs. The main design objectives of ADB are as follows [23]:

- 100% coverage of device wiring.
- Good database runtime performance.
- Rapid database initialization.
- Compact representation of data on disk.
- Compact representation of data in memory.
- Compatibility with JBits.

The exhaustive coverage is derived from the same proprietary data that the Xilinx mainstream tools use, and presents an improvement over the coverage available in past and present JBits wire databases. In addition, the database files used by ADB are more than an order of magnitude smaller than their counterparts in the mainstream tools. The performance and compactness may be especially useful in embedded systems that have to dynamically reconfigure themselves [24].

ADB can be run as a standalone tool that provides services to client tools such as browsers, routers, tracers, and placers. As a supplemental connectivity database, ADB also interfaces with JBits to provide wiring information and routing, unrouting, and tracing services. These interfaces are publicly exposed, so they can be extended or replaced as necessary by the user [25]. Since JBits3 no longer includes JRoute [26] – an autorouter present in prior versions of JBits – an ADB-based router is included with JHDLBits. The router is based on a robust search algorithm enhanced with heuristics suited to the Virtex-II FPGA family. The unrouter allows functionality normally not found in static tool flows, by allowing the user to disconnect existing nets, in order to reconnect other nets at runtime. The ADB-tracer

is also used by VTsim, as described in Section 2.7 below, to infer connectivity information from a configuration bitstream.

ADB may be of interest to researchers wanting to evaluate routing algorithms with wiring data from commercial FPGAs; however, ADB currently provides the only routing option available to JBits users. To plug the ADB router into the JHDLBits project, the router simply implements `RouterInterface` as defined by JBits, without concern for its inner workings.

2.7 VTsim

VTsim is an event-driven device simulator for Xilinx Virtex-II devices modeled at the CLB level and supports multiple clocks. The primary goal of VTsim is to create a Java-based virtual FPGA device that accurately models and simulates all of the resources within an FPGA. Currently, VTsim provides resource coverage of approximately 90%, including support for the majority of logic used in an FPGA design [27].

VTsim can be executed as part of three different design flows. The first design flow involves VTsim run in standalone mode, where it can take any valid Virtex-II bitstream as input. The second flow allows a user to generate a bitstream from a JBits3 design, which is then passed to VTsim as the input. The third flow involves VTsim as part of the open-source JHDLBits design flow. VTsim is tied into the JHDL GUI simulator, as described further in Chapter 4, Section 4.5.6. VTsim relies on a tracer supplied by ADB and on JBits3 as part of all three design flows.

The simulation process executed by VTsim proceeds as follows. VTsim first invokes the ADB tracer on the bitstream to be simulated. After the tracer builds a database of all internal connections, the simulator constructs a netlist from this information. An optional JHDLBits simulation file can be used to provide the simulator with specific information such as net names and placement information. Next, the tiles appearing in the netlist are configured using JBits. Once the virtual device has been configured, all non-clocked elements are evaluated to to ensure a stable device along with proper values on each simulation net. A

simulation cycle then begins by triggering the system clock. This event starts the evaluation of all clocked elements within the CLBs that are connected to that clock, followed by the evaluation of all non-clocked elements. Upon completion of a simulation cycle, the simulator can either write the updated results back to a bitstream, or allow GUI access to the modified information [28]. Figure 2.7 illustrates the process described above.

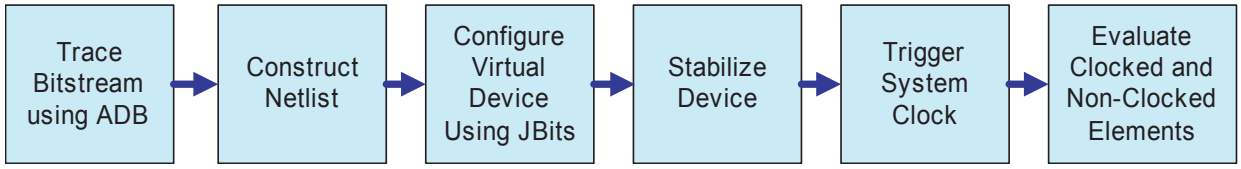

Figure 2.7: VTsim Simulation Process

VTsim was developed due to the fact that JBits3 does not include a device simulator. The previous version, JBits2.8, included VirtexDS, a device simulator for the Xilinx Virtex family [29]. During analysis VTsim exhibited rapid response, relatively low memory usage, and was found to outperform VirtexDS by up to 9000% [27].

2.8 Summary

This chapter presented background information relating to JHDLBits, focusing on the tools JHDLBits is dependent upon along with other automated FPGA design tools. The next chapter discusses the important features that make an open source project such as JHDLBits successful.

Chapter 3

Open Source Process

3.1 Overview

A goal of JHDLBits is to become a self-sustaining project, updated and maintained by a sizable user base. To accomplish this, JHDLBits will be released as an open-source project hosted on the SourceForge website [5]. In order to make this release successful, the open source software development process was researched. This chapter presents information on the history of open source, and on the strategies and philosophies that make an open source project successful. The Open Source Definition is also given, along with the licenses that are compliant with this definition. A prediction of the strategies and license that JHDLBits will use is also discussed.

3.2 History of Open Source

This section provides a brief history of the Open Source movement. It starts with the early accounts of software sharing and the founding of the Free Software Foundation. The section ends with the forming of the modern Open Source Initiative.

3.2.1 The Early Days

"In the beginning, there were Real Programmers [30]." These programmers usually came out of strong engineering or physics background. From the end of World War II to the early 1970s, the Real Programmers were the dominant technical culture in computing. In 1961, MIT acquired the first Digital Equipment Corporation's (DEC) PDP-1 and formed the beginnings of the hacker culture. They invented programming tools, slang, and an entire surrounding culture. The influence of the MIT laboratory spread even further after 1969, the first year of the ARPAnet. ARPAnet was the first transcontinental high speed computer network built by the Defense Department as an experiment in digital communications. It brought together universities, defense contractors, and research laboratories. This link gave way for researchers to exchange information, thus greatly increasing the idea of collaborative work. It also brought together hackers from all over the United States [30].

Richard Stallman started his career as a member of the MIT Artificial Intelligence Laboratory in 1971, where a software-sharing community existed for many years. This lab used a time-sharing operating system called ITS (the Incompatible Time-sharing System) that was internally developed for the DEC PDP-10, one of the largest computers of the time, first released in 1967. The software they developed was available to any university or company if they were interested [31].

3.2.2 Unix

In 1969, the same year as ARPAnet's birth, another Bell Labs hacker named Ken Thompson invented Unix. He had been working on a time-sharing operating system called Multics, which internally hid its complexities, making them invisible to the user and even to most programmers. Bell Labs pulled out of the project, and Thompson started missing the environment of Multics. He started implementing a combination of ideas from Multics along with some of his own ideas on a DEC PDP-7 that he had scavenged [30].

Another hacker named Dennis Ritchie invented a new language called *C* for use with Thompson's early version of Unix. Thompson and Ritchie were among the first to determine that

hardware and compiler technology had advanced enough that an entire operating system could be written in C, and by 1974 the entire environment had been ported to several machines of different types [30].

By this time, Unix had dedicated users within AT&T, but no larger group of users existed outside the company. This changed when Thompson and Ritchie presented a paper on Unix in late 1973 that was published in the summer of 1974. Following this, requests for copies of the operating system flooded in. This should have been viewed as a huge business opportunity for AT&T, but lawyers argued that the company had no intention of pursuing software as a business. The early Unix license was minimal and stated the software came *as is* with no support or bug fixes. These terms separated AT&T from any business interest. The terms of the license then gave users the initiative to share support and fixes with one another [32].

By 1980, Unix had spread to a large number of universities and research sites, thus giving access to hackers worldwide.

3.2.3 The Berkeley Software Distribution

The Berkeley Software Distribution (BSD) was established in 1977 at the University of California at Berkeley. BSD Unix was one of the most effective and well-known distribution channels for Unix and related software. The project was headed by Bill Joy, who later founded Sun Microsystems. The group modified and improved the Unix system. They also redistributed it to others who added their own enhancements, thus making BSD Unix very powerful. The Defense Advanced Research Projects Agency (DARPA) even chose to use BSD Unix to link together ARPAnet nodes that later became the Internet. In addition, the BSD group developed a TCP/IP protocol suite and mail utilities that contributed to the Internet [33].

Over its history, the BSD project forked into additional projects such as FreeBSD, OpenBSD, and NetBSD. This happened since the BSD code was available for users and companies to take and commercialize; however, it was left up to the user or company whether or not the modified source code would then be released to the public again. This attitude contributed to

the downfall of BSD. Large quantities of good source code was developed within proprietary implementations and then not released. Some people also made money relying on the work of volunteers, but in turn gave nothing back to the community [34].

In 1984, AT&T suffered the break-up of its monopoly, and then sought to commercialize Unix. A lengthy court battle between AT&T and BSD over copyright violations resulted in many claims and counter-claims by both parties. These claims were finally resolved during the early 1990s [33].

3.2.4 The Free Software Foundation

After Digital Equipment Corporation discontinued the PDP-10 series, and all of the programs that made up ITS were no longer useful, Richard Stallman resigned from MIT and founded the Free Software Foundation (FSF) as a nonprofit organization in 1984. His goal was to produce an entirely free operating system that anyone could download, use, modify, and distribute freely. Based on his training and the general type of users he would work with, he used Unix as the operating system model he would follow; however, he wanted to distance himself from the proprietary attitude that AT&T displayed, so the project was named the recursive acronym GNU (GNU's not Unix). He also defined the word *free* as used in his Free Software Foundation. Free did not mean zero price, but means *freedom*. To make this freedom effective in the community, users would be given full access to the source code. The four freedoms essential to Stallman's vision were as follows [32]:

- Freedom to run the program for any purpose.
- Freedom to study how the program works and to modify it to suit your needs.
- Freedom to redistribute copies, either gratis or for a monetary fee.
- Freedom to change and improve the program and to redistribute modified versions of the program to the public so others can benefit from your improvements.

Richard Stallman also developed the General Public License (GPL), which will be discussed in Section 3.6.1. The GPL license makes sure that free software and any derivative works

from free software will remain free. Some of the software that came out of the GNU project includes the GNU Emacs text editor, GCC compiler, and GDB debugger [32]. The public got the wrong idea of what free meant even though Richard Stallman tried hard to convey that free meant freedom. Many companies rejected the idea of free software since they were in the business of making money, not adding to our knowledge.

3.2.5 Linux

Linus Torvalds, a Helsinki university student, started the Linux operating system in 1991. Linux has in turn become the *poster child* of open source software. He modeled his operating system on Minix, a Unix clone developed by Andrew Tannenbaum. The source code for Minix was available, but any changes required Tannenbaum's permission [33].

Torvalds' goal with Linux was to create a Unix-like operating system for the IBM PC 386 series with help from other programmers worldwide. Some estimates show that more than a thousand developers helped with the Linux kernel development, while more than 40,000 developers have worked on the overall Linux operating system. It is also estimated that Linux has more than 25 million users. It is currently the most widely ported operating system available on the PC platform [33].

Linus Torvalds claims that Linux represents the largest collaborative project in the history of the world. It should be noted that a typical Linux distribution is dependent on tools and utilities that were developed by both the BSD and FSF [33].

3.2.6 Open Source Initiative

In the spring of 1997, many leaders in the free software community met to discuss how to get over this notion that free software meant zero price. These discussions led to coining the term *Open Source*. This meeting is also where the groundwork for the Open Source Definition as defined in Section 3.3 below was laid out. This definition allowed for more liberties with licensing than the GPL allows [35]. This also started the foundation of the

Open Source Initiative (OSI).

3.3 Open Source Definition

The term *Open Source* is too broad and descriptive to be a trademark under United States law. In general, Open Source Software is software distributed under the terms that comply with the Open Source Definition. The Open Source Definition (OSD) is maintained by the Open Source Initiative (OSI). This definition is not considered to be a license, but defines how a product's *terms of distribution* can be measured. [36].

The Open Source Definition [37] is included below:

Introduction Open Source does not just mean access to the source code. The distribution terms of open-source software must comply with the following criteria:

1. **Free Redistribution** The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.
2. **Source Code** The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.
3. **Derived Works** The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

4. **Integrity of The Author's Source Code** The license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.

5. **No Discrimination Against Persons or Groups** The license must not discriminate against any person or group of persons.

6. **No Discrimination Against Fields of Endeavor** The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

7. **Distribution of License** The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

8. **License Must Not Be Specific to a Product** The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.

9. **License Must Not Contaminate Other Software** The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.

10. **License Must Be Technology-Neutral** No provision of the license may be predicated on any individual technology or style of interface.

3.4 Open Source Philosophies and Strategies

Eric Raymond was involved in the open source community and in developing Unix for a number of years. He thought that he understood the open source development process. He *"believed that the most important software needed to be built like cathedrals, carefully crafted by individual wizards or small band of mages working in splendid isolation, with no beta to be released before its time [38]."*

It was a surprise that Linux became successful under a completely different development process. Linus Torvald's style was based on the following principles [38]:

- Release early and often.
- Delegate everything you can.
- Be open to the point of promiscuity.

"No quiet, reverent cathedral-building here - rather, the Linux community seemed to resemble a great babbling bazaar of different agendas and approaches (aptly symbolized by the Linux archive sites, which would take submissions from anyone) out of which a coherent and stable system could seemingly emerge only by a succession of miracles [38]."

Raymond started trying to understand why the bazaar style seemed to work so well, and formed his own open source project that was developed in the bazaar style in 1996. The name of the project was *Fetchmail* and ended up being a significant success.

The following are the main principles Raymond used in developing his project which were published in *The Cathedral and the Bazaar* [38]:

1. Every good work of software starts by scratching a developer's personal itch.
2. Good programmers know what to write. Great ones know what to rewrite (and reuse).
3. Plan to throw one away; you will, anyhow.

4. If you have the right attitude, interesting problems will find you.
5. When you lose interest in a program, your last duty to it is to hand it off to a competent successor.
6. Treating your users as co-developers is your least-hassle route to rapid code development and effective debugging.
7. Release early. Release often. And listen to your customers.
8. Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone.
9. Smart data structures and dumb code works a lot better than the other way around.
10. If you treat your beta-testers as if they're your most valuable resource, they will respond by becoming your most valuable resource.
11. The next best thing to having good ideas is recognizing good ideas from your users. Sometimes the latter is better.
12. Often, the most striking and innovative solutions come from realizing that your concept problem was wrong.
13. Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away.
14. Any tool should be useful in the expected way, but a truly great tool lends itself to uses you never expected.
15. When writing gateway software of any kind, take pains to disturb the data stream as little as possible - and *never* throw away information unless the recipient forces you to!
16. When your language is nowhere near Turing-complete, syntactic sugar can be your friend.
17. A security system is only as secure as its secret. Beware of pseudo-secrets.
18. To solve an interesting problem, start by finding a problem that is interesting to you.

19. Provided the development coordinator has a medium at least as good as the Internet, and knows how to lead without coercion, many heads are inevitably better than one.

Another philosophy to an open source development process starts with the same ideal process, but then forks depending on the license that the software is licensed under. The ideal process is made up of the following fundamentals [39]:

- The key element is voluntary participation and voluntary selection of tasks.
- It revolves around a core code base.
- Source code for that code is available freely. Anyone can obtain it, usually over the Internet.
- Anyone can modify the code, freely, for his or her own use.

From here on, the process differs depending on how the software is licensed. BSD-style licenses are not very constraining, so the user if free to do whatever they please. The GPL-style license is much more constraining. These licenses will be discussed in further detail in Section 3.6.

The key to open source success is only partially based on what developers do for themselves with the code. It is also based on what developers contribute back to the collective project. This differs once again on how the software is licensed. Following are the strategies used by BSD-style projects [39]:

- Small team of developers.
- Outside users may modify the source code for their own purposes.
- Report bugs to the core team.
- Sometimes suggest new features or approaches to problems that might be helpful.

- Core development team does not generally rely heavily on code that is written by users.
- No regularized process for submitting code to the core team.
- Open source since the source code is free.
- Not vitally collaborative.

On the other hand, the strategies of Linux-style (licensed under GPL) projects are as follows [39]:

- General user base can do *check-ins* to the code.
- No distinction between developers and users.
- Not just bug reports, but real code modifications and new features.
- Actively encourages extensions and improvements.
- Lowers the barriers for individuals to join debugging and development.
- Process for reviewing submissions is ordered and methodical.
- If a code change is rejected, the user has the right to create a *new* open source project which incorporates the change, and then invite others to join (forking the code base).

Some developers believe that the essential freedom of open source development is the right to fork the code at any time.

3.5 JHDLBits Strategy

JHDLBits will be considering all of the strategies above in determining the most successful course of action in proceeding with an open source release. Since this project will remain at Virginia Tech, the core development will most likely continue there; however, the team will rely on code developed by the FPGA research and design community. Following are some preliminary strategies that the JHDLBits team will follow:

- Use a combination of BSD-style and Linux-style.
- Use a small core team of developers.
- Users will report bugs to core team.
- If a user becomes active, they can request permission for Subversion (version control) access.
- Develop and publish a process for submitting code changes.
- Develop and publish a process for reviewing submissions.
- Rely on code written by general users.

3.6 Licenses

Choosing a license for an open source project is not a step to be taken lightly. The complications involved with a poorly chosen license can create a large burden on a project. The OSI maintains a list of the licenses [40] that have been reviewed and found to be compliant with the OSD defined in Section 3.3. To keep this section as informative as possible, only the most widely used licenses will be discussed in detail. These consist of the GNU General Public License (GPL), the GNU Library or *Lesser* Public License (LGPL), the Berkeley Software Distribution (BSD) License, and the MIT License.

3.6.1 GPL and LGPL

As mentioned above in Section 3.2, the GPL and LGPL were created by Richard Stallman's FSF to protect his four envisioned freedoms.

Users may modify software licensed under GPL in any way they see fit; however, if a modified version is publicly released, it must be distributed under the same terms of the GPL [36]. It is not permitted under the GPL to combine a free program with a non-free program unless

the entire combination is then released as free software under the GPL. This explains the GPL's most notable trait, its *viral* nature since it *infects* other software [32].

GPL is based on a method called *copyleft*. Copyleft uses copyright law, but flips it over to serve the opposite of its usual purpose: instead of a means of privatizing software, it becomes a way of keeping software free. For an effective copyleft, all modified versions must also remain free [31]. This was designed to keep developers from *hoarding* software code under any conditions.

The FSF created a second license, the LGPL. This license provides a non-viral alternative to the GPL and was designed to be used with programming language libraries. A library in this context is a collection of routines that other applications can use. If a program only calls or links to the library versus actually containing code from it, then it is not covered under the LGPL. It is covered if the new program actually contains code or extends code from the library [36]. Software written under LGPL can also be incorporated into proprietary software.

3.6.2 BSD

The BSD License originated from software that was funded by monetary grants from the United States government. Since the U.S. citizens had already paid for the software with their taxes, they were given permission to do nearly anything with the software [41].

The BSD License is pretty much at the opposite end of the scale from GPL, and defines relatively few constraints on the terms of distribution for modified versions of the software. The three basic provisions and disclaimer taken from the BSD License Template on the OSI website are [42]:

> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the organization nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

3.6.3 MIT

The MIT License was originally created for the X Windows graphics system and also has few constraints. It only requires that the copyright notice and permission remain intact on full or substantial copies of the software [36]. This license is thus equivalent to the BSD License, except for the no-endorsement final clause [42].

3.7 JHDLBits License

ADB which was introduced in Section 2.6 has started the process of obtaining a license through Virginia Tech, but it has not been finalized at the date of this publication. This license is derived from the GPL and MIT licenses. Based on this, JHDLBits would most likely be licensed in a similar manner. Some of the constraints of the proposed ADB license are:

- Must include copyright notice and license on any copy or modification of this Software.
- If a section of the modified code is not derived directly from this Software, then this section of code can be released as a separate work, and the license doesn't apply.
- Nobody will be held liable for any damages that occur from using this Software.
- As is, no warranty.
- Neither the name of the organization nor the names of its contributors may be used to endorse or promote products derived from this Software without specific prior written permission.

3.8 Summary

This chapter provided the relevant background information required to understand some of the strategies that the JHDLBits project intends to follow to produce a successful open source project. To understand the underlying design and code of JHDLBits, Chapter 4 will present the design approach taken, along with describing the core components that make up the JHDLBits project.

Chapter 4

JHDLBits Design

4.1 Overview

Chapter 2 provided the background information required for a reader to understand the sections presented in this chapter. First, the JHDLBits design approach will be discussed followed by details of how JHDL and JBits are tied into JHDLBits. Finally, the core components of the JHDLBits flow are examined.

4.2 Design Approach

The early stages of the JHDLBits project involved exploration on how to make the current JBits3 release more usable and flexible to the research community. As discussed earlier in Section 2.5, JBits3 was released without a router, placer, simulator, or higher levels of abstraction. The libraries available were made up of very low-level calls to access the configurable resources available on a Virtex-II FPGA. As part of the scope for this project a router, placer, and simulator were to be written; however, it was decided that writing lower level primitives such as basic logic gates and flip-flops, followed by higher level cores would be very time consuming. In past JBits releases, Xilinx Research Labs took several years

to develop a new release while having multiple engineers working on the project, including the Virginia Tech Configurable Computing Lab, under the support of the DARPA Adaptive Computing Systems program. In contrast, the JHDLBits project is quite constrained. We had only the part-time manpower of four graduate students and two professors and very limited funding.

The JHDLBits team brainstormed for a few weeks to determine the best course of action. It was recalled that JHDL, as discussed in Section 2.4, supported Virtex-II devices, was written in Java, and already contained built-in cores made up of lower level primitives. The initial approach was to determine whether or not it would be possible to create a flow that transformed the JHDL cores into JBits cores. After spending many hours of looking through the JHDL libraries and classes, it was determined that the best way of *hooking into* JHDL was during the netlisting phase, as described in Section 4.3 below.

Next, decisions were made on the how to proceed with a placer, router, and simulator. To begin with, a simple greedy placer was to be written. Since ADB, as discussed in Section 2.6, contained a router and an interface to JBits, these portions were already available. A complete device level simulator was also to be written.

For the proof of concept the JHDLBits team selected a simple JHDL design that did not contain any top-level I/Os or clock pins.

4.3 JHDL Tie-In

As mentioned above, the place to tie into the JHDL flow is during the netlisting phase. Instead of producing an EDIF file that is run through the Xilinx mainstream tools, JHDLBits extracts the portions of the JHDL netlist required to generate a bitstream using JBits. This is demonstrated in Figure 4.1 below.

When netlisting a JHDL design, the user has the option of producing a flat or hierarchical netlist. To keep the JHDL to JBits design flow simple during the proof of concept phase, a flat netlist is used.

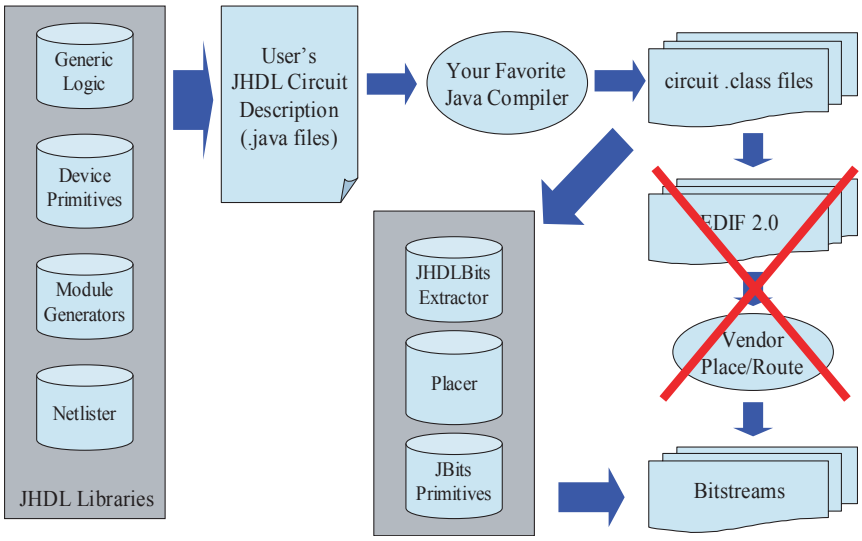

Figure 4.1: Modified JHDL Flow

Internally, a JHDL design is made up of many Class objects called `Cells`. Determining what level of JHDL abstraction should be extracted into JBits objects is difficult because it may limit the type of designs a user could develop within the JHDLBits flow. To allow the user the full range of JHDL libraries in constructing circuits, the lowest level primitives will be created in JBits.

While looking through the JHDL libraries, there are classes available to the user to include placement directives for circuits to be constructed. This seems like a good start for the JHDLBits placer construction, however these placement directives are ignored by JHDL. The mainstream tools are evidently much further advanced in placement and routing than in previous Xilinx FPGA families making such directives unnecessary.

As discussed in Section 2.4, the JHDL simulation environment appears the same to the user in either software or hardware simulation mode. It is therefore a valuable asset for the JHDLBits flow to allow VTsim to interface with this environment. In hardware mode `Cells` that extend the `ExternallyUpdateable` class are read back from the actual hardware, and the updated values are displayed in the GUI. However, in this mode only memory elements such as flip-flops are actually extracted from the hardware, and the rest of the circuit is still simulated behaviorally.

4.4 JBits Tie-In

A goal of the JHDLBits project is to enable JBits to be used as part of the JHDLBits flow, but also to allow a designer to still create designs in a JBits-only flow. As part of either of the design flows, a library of primitives was developed that builds upon the lower level JBits calls to configure logic and resources in the FPGA fabric. In addition, a bridging object, the `Bitstream` class, was created to allow abstract access to both the JBits and ADB objects, enabling the creation of primitives without dependencies on architecture-specific classes. The `Net` class was extended to allow the connection of primitives by maintaining a list of the source and sink pins that form a net [20].

The library of JHDLBits primitives was started based on the primitives that were extracted from a JHDL design. For completeness, each JHDL primitive requires a corresponding JBits primitive, allowing for a quick and direct data conversion with low memory usage. Currently the library is made up of the following primitives:

- basic logic gates, such as inv, and2, and3b2, or3, xor4, etc.
- flip-flops, such as fdc, fdce, etc.
- gnd and vcc
- adder
- muxcy and xorcy
- bufg
- ibuf and obuf
- 18x18-bit multiplier
- RAMB16_S36

Each primitive in the library is in its own file and organized as part of a tree structure. At the top level is the `ULPrimitive` class, which each primitive must extend. At the same

level as this class is a file called `PrimitiveRegistry` that contains the path names for the entire tree structure. This makes it easy to include primitives into the tree that may be stored in a completely separate directory (i.e. the file path can just be added to the `PrimitiveRegistry`). At the second level, the primitives are divided into directories of tile types. For example, all primitives that are based on LUT configurations are part of the `CENTER` tile type directory. All input and output related primitives are part of the `IOIS` tile type directory. Different configurations of a Block SelectRAM are in the `BRAM` tile type directory. The clock and multiplier primitives are not inside a directory since they are the only primitives that exist within that tile type. Figure 4.2 shows an illustration of the JHDLBits primitive class hierarchy.

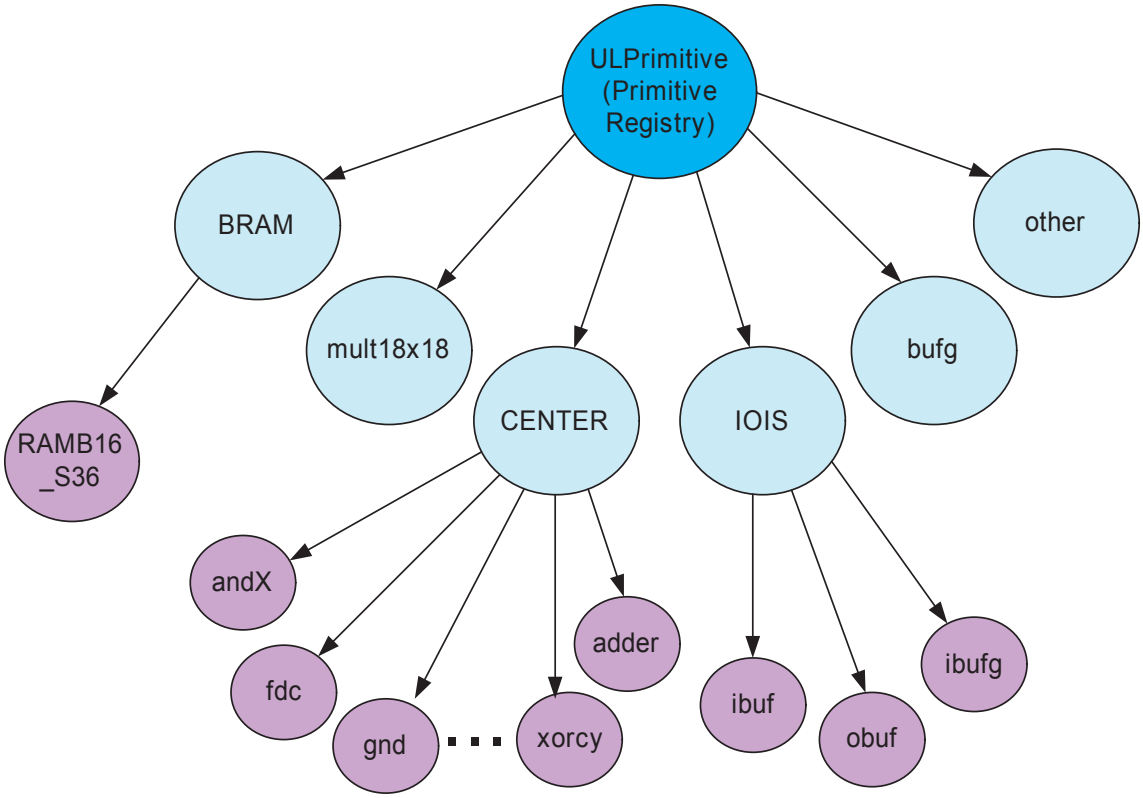

Figure 4.2: JHDLBits Primitive Class Hierarchy

Each primitive is an object, meaning each is a separate Java class. The primitives have two constructors specifically designed for the JHDL-to-JBits flow, with the other constructors

catering to a JBits-only design flow. A primitive requires two pieces of information to be implemented: a list of input and output nets, and placement information. The primitive uses the placement location to assign source and sink pins to the associated nets, and then accesses the JBits object from the `Bitstream` class to configure the internal logic and resources using JBits calls.

Figures 4.3 and 4.4 below show example code of a 4-input OR primitive. This primitive is part of the `CENTER` tile type since an OR gate can be programmed by using a LUT. This example can be used as a template for future primitive designs that may occur as part of the open source initiative.

In programming a LUT primitive, the user must first come up with a truth table to determine the output function, as shown in Lines 3-23. This output function is then inverted to form the stream to be programmed as the contents of the LUT from least significant bit (LSB) to most significant bit (MSB) as declared in Line 29. The PORT_ORDER variable defines the order of the list of input and output nets to be passed to the primitive. The HEIGHT and WIDTH variables are set in Lines 26-27 to later be used in the placement algorithm. They define the size of a primitive in terms of logic elements. The first constructor shown on Lines 30-37 was used in the early stages of the JHDLBits project. At that time, the level of granularity for placement was at the slice level; however, currently the second constructor on Lines 38-44 is used when creating a JBits primitive. This constructor allows the placement algorithm to define the CLB, slice, and LE locations for a primitive. The remaining constructors are used as part of the JBits-only flow. The last function on Lines 71-73, named `implement()`, calls the super class' `implement` function.

Figure 4.5 shows a portion of the code from the super class the previous OR example extended. The code details how the `implementLUT4()` method uses JBits calls to configure the internal resources of the FPGA. Lines 12-13 show how the LUT contents and output are configured based on the CLB row and column values, slice, and LE value of F. Lines 16-20 show how the sink and source net wires are configured for the F LE in a particular slice. Note that all of this code is replicated for the G LE with the appropriate changes to the wire arguments.

```
1  package JHDLBits.Virtex2.ULPrimitives.CENTER;
2  import JHDLBits.ArchIndependent.Net;
3  /* Four input OR gate on G1, G2, G3, G4 Truth Table
4     G4 G3 G2 G1 D
5     0  0  0  0  0
6     0  0  0  1  1
7     0  0  1  0  1
8     0  0  1  1  1
9     0  1  0  0  1
10    0  1  0  1  1
11    0  1  1  0  1
12    0  1  1  1  1
13    1  0  0  0  1
14    1  0  0  1  1
15    1  0  1  0  1
16    1  0  1  1  1
17    1  1  0  0  1
18    1  1  0  1  1
19    1  1  1  0  1
20    1  1  1  1  1
21     // LUT contents (inverted and ordered from LSB to MSB)
22     1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
23 */
24 public class or4 extends CENTER {
25     public static final String[] PORT_ORDER = {"i0","i1","i2","i3","o"};
26     public static final int HEIGHT = 1;
27     public static final int WIDTH = 1;
28     // LUT contents (inverted and ordered from LSB to MSB)
29     private static final int[] LUTor4 = {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
30     // Constructor with no LE granularity specified... default to G
31     public or4(Net[] net, int clbRow, int clbCol, int slice) {
32         super.net = net;
33         super.LUTValue = LUTor4;
34         // specify the LE as G lut since not specified
35         super.place(clbRow, clbCol, slice, G);
36         implement();
37     }
```

Figure 4.3: 4-Bit OR JBits Primitive Part 1

```
38      // Constructor allowing specification of LE granularity
39      public or4(Net[] net, int clbRow, int clbCol, int slice, int LE) {
40          super.net = net;
41          super.LUTValue = LUTor4;
42          super.place(clbRow, clbCol, slice, LE);
43          implement();
44      }
45      // Constructor allowing the user to create an instance of the primitive without
46      // specifying placement directives. The user or another file will need to call:
47      // 'primtive'.place(clbRow, clbCol, slice, LE)
48      public or4(Net[] net) {
49          super.net = net;
50          super.LUTValue = LUTor4;
51      }
52
53      // Constructor allowing specification of LE granularity with seperate nets
54      public or4(Net a, Net b, Net c, Net d, Net out, int clbRow, int clbCol, int slice,
55              int LE) {
56          Net[] netA = {a,b,c,d,out};
57          super.net = netA;
58          super.LUTValue = LUTor4;
59          super.place(clbRow, clbCol, slice, LE);
60          implement();
61      }
62
63      // Constructor with no LE granularity specified... default to G with seperate nets
64      public or4(Net a, Net b, Net c, Net d, Net out, int clbRow, int clbCol, int slice) {
65          Net[] netA = {a,b,c,d,out};
66          super.net = netA;
67          super.LUTValue = LUTor4;
68          super.place(clbRow, clbCol, slice, G);
69          implement();
70      }
71      public void implement() {
72          super.implementLUT4();
73      }
74  }
```

Figure 4.4: 4-Bit OR JBits Primitive Part 2

```
public abstract class CENTER extends ULPrimitive {
    ⋮
    // The implement function uses specified jbits calls to set the bitstream
    public void implementLUT4() {
        final int i0 = 0;
        final int i1 = 1;
        final int i2 = 2;
        final int i3 = 3;
        final int o = 4;
        JBits jbits = Bitstream.getJBits();
        switch(LE) {
        case F:
            jbits.setCLBBits(clbRow, clbCol, LUT.CONTENTS[slice][LUT.F], LUTValue);
            jbits.setCLBBits(clbRow, clbCol, FXMUX.CONFIG[slice], FXMUX.F);
            switch(slice) {
            case 0:
                net[i0].addSink(bitstream.newPin(Pin.CLB,clbRow,clbCol,Wire.F1_B0));
                net[i1].addSink(bitstream.newPin(Pin.CLB,clbRow,clbCol,Wire.F2_B0));
                net[i2].addSink(bitstream.newPin(Pin.CLB,clbRow,clbCol,Wire.F3_B0));
                net[i3].addSink(bitstream.newPin(Pin.CLB,clbRow,clbCol,Wire.F4_B0));
                net[o].addSource(bitstream.newPin(Pin.CLB,clbRow,clbCol,Wire.X0));
                break;
            case 1:
                net[i0].addSink(bitstream.newPin(Pin.CLB,clbRow,clbCol,Wire.F1_B1));
                net[i1].addSink(bitstream.newPin(Pin.CLB,clbRow,clbCol,Wire.F2_B1));
                net[i2].addSink(bitstream.newPin(Pin.CLB,clbRow,clbCol,Wire.F3_B1));
                net[i3].addSink(bitstream.newPin(Pin.CLB,clbRow,clbCol,Wire.F4_B1));
                net[o].addSource(bitstream.newPin(Pin.CLB,clbRow,clbCol,Wire.X1));
                break;
            case 2:
                //same as above with F1_B2 - F4_B2
            case 3:
                //same as above with F1_B3 - F4_B3
            }
            //same as above for G
    ⋮
```

Figure 4.5: CENTER Primitive Code

4.5 JHDLBits Components

The traditional JHDL design flow produces an EDIF file, which is used by the Xilinx implementation tools to generate a bitstream. The JHDLBits flow instead relies upon JBits extensions to generate a bitstream. A high-level design is created and simulated with JHDL libraries and graphical debugging tools. The JHDL primitives, instances, and nets are extracted and mapped to equivalent JBits primitives and nets. JBits primitives are placed using either placement directives or an extendable placement algorithm, and then routed using ADB. A bitstream is then generated, along with net and simulator files. Figure 4.6 shows the steps involved in the JHDLBits design flow, which are explained in further detail in the following subsections.

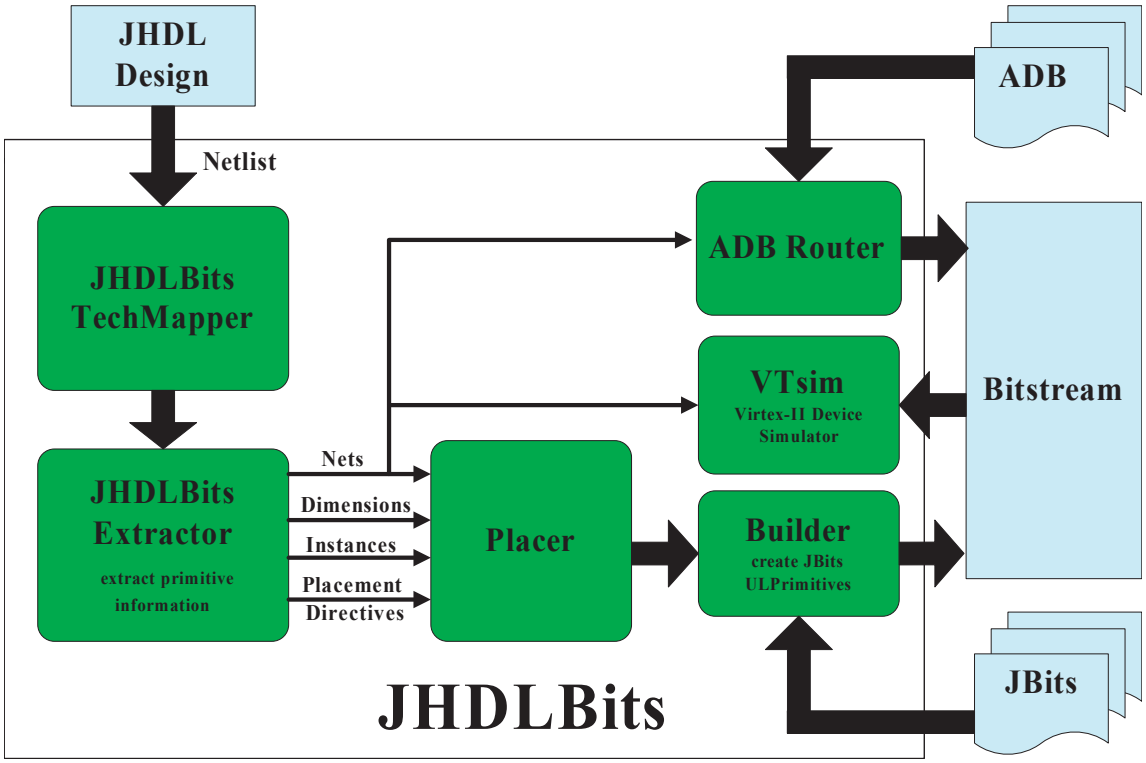

Figure 4.6: JHDLBits Design Flow

4.5.1 JHDLBits TechMapper

The first step in the JHDLBits design flow is the `JHDLBitsTechMapper`, which functions as the interface into the JHDL flow. The `JHDLBitsTechMapper` extends the JHDL `Virtex2TechMapper` class that is used by JHDL during the netlisting phase to map primitives to architecture-dependent resources by calling the `EDIFNetlister` class. Instead of using the `EDIFNetlister`, the `JHDLBitsTechMapper` invokes the `JHDLBitsExtractor` as its default netlister. Since `JHDLBitsTechMapper` provides the external interface into the JHDL flow, there are multiple constructors available to the user which allow passing the target device and package along with input and output bitstreams. These constructors are discussed in further detail in Section 5.3. The `JHDLBitsTechMapper` also provides access to the `JHDLBitsExtractor` object which is used throughout the rest of the JHDLBits design flow.

The JHDL `Virtex2TechMapper` has been optimized to the Virtex-II architecture through the use of helper functions such as ADD and SUBTRACT. For example, these functions take a JHDL add circuit and map it to very low level resources to take advantage of the high-speed adder resources available in the Virtex-II architecture. These resources are at a much lower level than the other primitives found during netlisting. It was difficult to create JBits calls for these very low level resources, so the `JHDLBitsTechMapper` overrides these helper functions by calling our own ADDER primitive, which also takes advantage of the high-speed resources.

4.5.2 JHDLBits Extractor

The `JHDLBitsExtractor` uses JHDL libraries such as `Cell` and `Net` classes to extract primitive, instance, and net information from a JHDL design. The `JHDLBitsExtractor` is invoked when the `JHDLBitsTechMapper netlist()` method is called, which then calls the `JHDLBitsExtractor expand()` method.

The `expand()` method is a recursive function that starts at the top level cell of a JHDL design and then iterates through all of the children cells until all terminating primitive cells are encountered. Figure 4.7 illustrates the `expand()` code in the form of a flowchart.

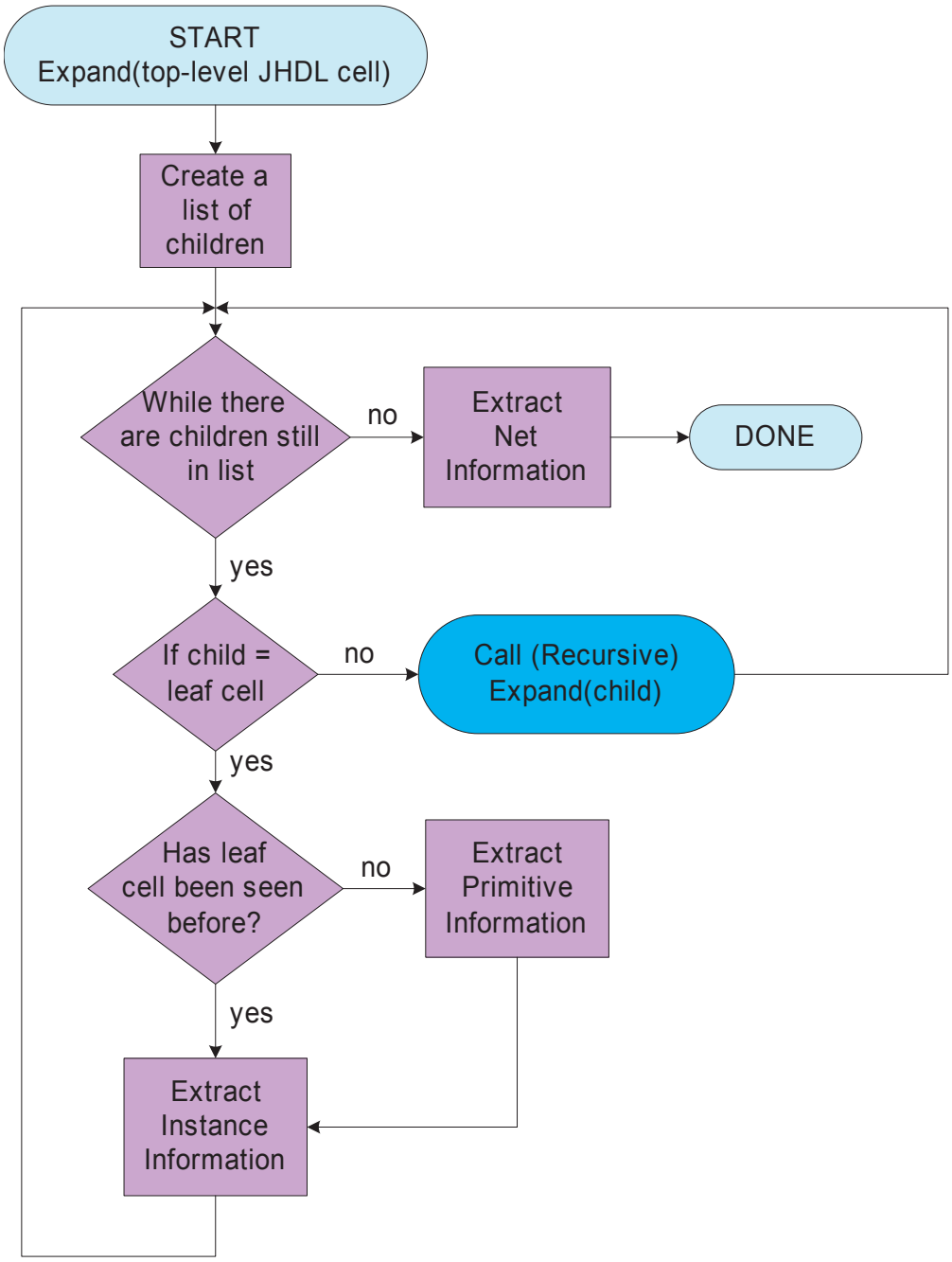

Figure 4.7: JHDLBitsExtractor expand() Flowchart

Figure 4.8 below is a portion of code from the `expand()` function. Line 14 is reached once a cell cannot be expanded anymore. This primitive cell is then checked to see if it has been encountered before. If not, then the primitive information is extracted in the `extractPrimitive()` function. An instance is an occurrence of a primitive, so the instance information is always extracted in the `extractInstance()` function shown in Line 17.

```
    ⋮
1   protected void expand(Cell c, NetlistWriter output) {
2
3       String cellname = c.getCellName(); // defaults to class name
4       CellList children;
5       // list of top level cell's children
6       children = c.getFlatNetlistableChildren();
7
8       Cell child;
9       // iterate through all children
10      for (children.init(); ! children.atEnd(); children.next()) {
11          child = children.getCell();
12          expand(child, output);
13          // primitive cell hasn't been encountered before, so extractPrimitive
14          if (!primitiveMap.containsKey(child.getCellName()))
15              extractPrimitive(child);
16          // always extractInstance
17          extractInstance(child,c);
18      }
19      // all children have been iterated
20      if (!children.empty()) {
21          extractNet(c);
    ⋮
```

Figure 4.8: Portion of JHDLBitsExtractor expand() function

The `JHDLBitsExtractor` maintains four major data structures which are stored as `HashMap` collections. The first primitive `HashMap` (established in the `extractPrimitive()` function) contains the primitive name along with the names of all ports and corresponding directions. At this point, the JBits primitive of the same name is also accessed (not created yet) in order

to save the corresponding JBits class, height, width, port order, and tile type information from the primitive as discussed in Section 4.4.

The second instance `HashMap` (established in the `extractInstance()` function) contains the corresponding primitive (found in the previous `HashMap`), instance name relative to the parent, and JHDL cell. The instance name allows us to maintain the hierarchy of the cell for later development.

Once the primitives and associated instances have been extracted, all of the net information is extracted from the top-level JHDL `Net` class in the `extractNet()` function shown on Line 21 in Figure 4.8 above. This function extracts the sink and source ports for each net. These ports also have an instance associated with them, so the instance `HashMap` then has the additional net information stored. The net `HashMap` contains the ports of each net, along with a JBits net of the same name.

Now that the extraction process has been completed, there is some additional code in the `JHDLBitsExtractor` which cleans up all of the information stored. JHDL has an FMAP primitive that contains information about which primitives should be *packed* together while placing. At this time, JHDLBits is ignoring these primitives, so they have been purged from the list of instances. Also, there are some nets that contain just one port. These nets are mainly associated with a GND or VCC primitive. If a net with only one port is found, this net and associated instances are also purged.

Finally, the instance `HashMap` can now be iterated in order to specify placement of the instances.

4.5.3 JHDLBits Placer

During the JHDLBits proof of concept phase, a very simple greedy placement algorithm was developed. Primitives are placed in the FPGA fabric from left to right, until the row fills up and then the next column in used. As mentioned above, the instance `HashMap` is iterated to place each instance in a specific CLB Row, CLB Column, slice, and LE location. Once this occurs, the assigned locations are added into the instance `HashMap` for later use. It is of note that all instances are currently placed in a random fashion, i.e. related primitives aren't placed near each other since the `HashMap` is in no specific order.

A more sophisticated placement algorithm is being developed. This algorithm will take advantage of the hierarchy information stored to place related primitives near each other, generally making the routing much more efficient.

4.5.4 JBits Primitive Instantiation

Once again, the instance `HashMap` is iterated to create a JBits primitive for each instance. Each instance contains a list of input and output nets, along with placement information, with which the JBits constructor can be called. Figure 4.9 shows the portion of the code that instantiates a JBits primitive. Lines 3 and 5 show how the current instance and primitive of that instance are obtained from the `HashMaps`. The primitive is required to access the port_order and JBits class information. Next, the input and output nets of the instance are converted into a net array shown in Lines 7-14. The JBits primitive instantiation function is called in Lines 15-17. This function calls the JBits primitive constructor.

The assigned coordinate information is also maintained for an instance that is `ExternallyUpdateable`, which refers to a JHDL cell (flip-flops, memories, etc.) that changes value and can be read back using the JHDL simulator in hardware execution mode. This is shown in Lines 19-28.

```
for(int i=0;i<instanceKeys.length;i++) {
    // get current instance
    ExInstance curInstance = (ExInstance) instanceMap.get(instanceKeys[i]);
    // get primitive of current instance (required for port_order and JBits class)
    ExPrimitive curPrimitive = curInstance.primitive;
    // create a JBits Net based on port_order length
    JHDLBits.ArchIndependent.Net[] curJbitsNet =
        new JHDLBits.ArchIndependent.Net[curPrimitive.port_order.length];
    // convert all instance input and output nets into a net array
    for(int k=0;k<curPrimitive.port_order.length;k++) {
        //get net object based on port ref
        curJbitsNet[k] = ((ExNet) curInstance.nets.get(curPrimitive.
            port_order[k])).jbitsNet;
    }
    // create a JBits primitive
    createObject(curPrimitive.jbitsClass, curJbitsNet, curInstance.row,
        curInstance.col, curInstance.index, curInstance.le);

    // check if instance is of type ExternallyUpdateable; get placement locations
    // this is later used by VTsim which needs tile row & col, not CLB row & col
    if (curInstance.JHDLCell instanceof ExternallyUpdateable) {
        int tileRow = curJbitsNet[curPrimitive.getMemPortIndex()].source.getTileRow();
        int tileCol = curJbitsNet[curPrimitive.getMemPortIndex()].source.getTileCol();
        UpdateablePlacement update = new UpdateablePlacement(tileRow, tileCol,
            curInstance.index, curInstance.le);

        updateablePlacementMap.put(curInstance.JHDLCell, update);
    }
}
```

Figure 4.9: JBits Primitive Instantiation Code from JHDLBitsExtractor

4.5.5 Router and Bitstream Generation

Once all of the JBits primitives have been created, a router utilizing the connectivity information in ADB routes all nets connected to the JBits primitives. This is allowed through the JBits `Net` class by giving access to the router object. After the design has been success-

fully routed, JBits is used to generate a bitstream. The JBits `Bitstream` class has access to the JBits object, so the `JHDLBitsExtractor` can call `bits.writeFull()` to write a full bitstream.

4.5.6 VTsim

JHDLBits and VTsim tie into the JHDL simulator through the JHDL `HardwareInterface`. VTsim is given the bitstream generated, along with coordinate information for the cells that are `ExternallyUpdateable`, and returns values for these cells after each clock cycle. The updated values are displayed in the JHDL GUI interface. As noted earlier, only the values read back from the hardware show the results of simulation through VTsim. The rest of the circuit elements are still modeled behaviorally. In the future, it would be valuable to have all circuit elements show the simulation results from VTsim.

In addition to being tied into the JHDL simulator, VTsim was also used to test the bitstreams originating from the JHDLBits flow. Designs were created in JHDL and then run through the JHDLBits flow to generate a bitstream, and were also run through the traditional JHDL flow to generate a second bitstream. Both of these bitstreams were simulated with VTsim, and the results compared. If the results did not match, the JBits primitives were modified until the results did match. Figure 4.10 taken from [27] illustrates the bitstream verification process.

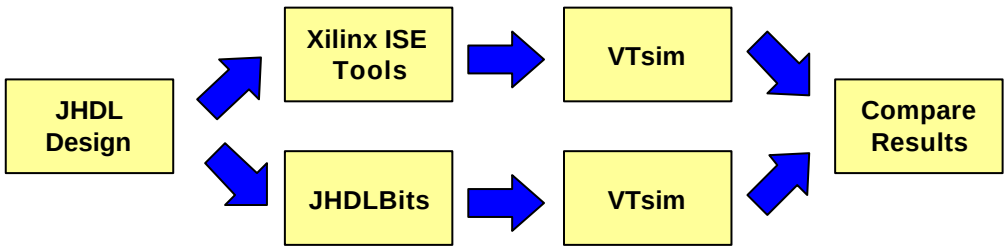

Figure 4.10: Bitstream Verification Process

4.6 Partial Reconfiguration

Partial reconfiguration will be supported in JHDLBits through an incremental design flow. In an incremental design methodology, a user has completed part of a design, and would like to *lock it down* by no longer having to repeatedly place and route this part of the design. This would be accomplished by including a reserved area with defined inputs and outputs into the current JHDL design. This design is run through the `JHDLBitsExtractor`, but the placer and router are constrained to avoid the reserved area. At a later time, the user can then include additional logic into the reserved area, and once again run through the `JHDLBitsExtractor` in order to generate a bitstream [20].

4.7 Summary

This chapter presented details of how JHDLBits was developed along with an in depth view of the JHDLBits design flow. The information above described the inner workings of JHDLBits which are generally transparent to a user. To get a better understanding of how to use JHDLBits to generate a bitstream for a JHDL design, Chapter 5 will provide examples to illustrate the options available and steps involved.

Chapter 5

JHDLBits Usage

5.1 Overview

To take advantage of the features offered by the JHDLBits flow, a user will need a better understanding of how to get started and create designs. This chapter will take the user through example JHDL designs to become familiar with available JHDL libraries. Next, key methods and a simple example showing the minimally essential JHDLBits needed in any design are given. Finally, a more complex design is presented that shows how various JHDLBits components can be invoked such as the JHDLBits GUI or VTsim.

5.2 Example JHDL Designs

Since the JHDLBits flow relies upon a user to first create a design using the JHDL libraries, it is important to understand the various parts of a JHDL design. First, a simple design will be explained and then a second design will demonstrate how to incorporate the first simple design as a building block.

The first example is a FullAdder circuit, shown in Figure 5.1 below. The circuit is constructed with AND, OR, and XOR gates.

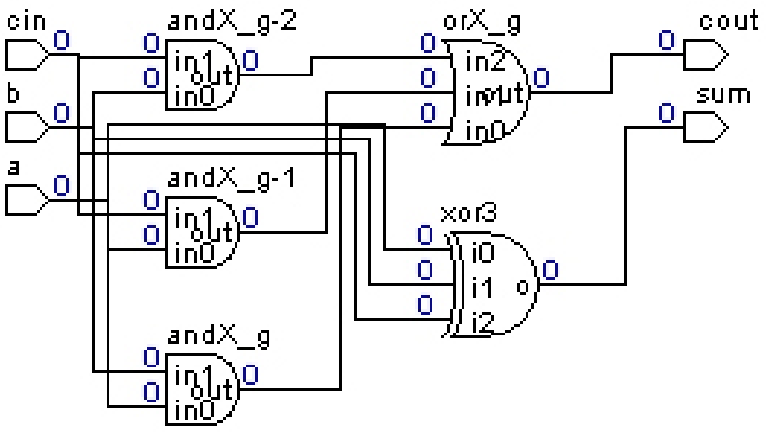

Figure 5.1: JHDL FullAdder Design

Figure 5.2 illustrates the JHDL code written for the FullAdder circuit above. Since JHDL is a Java-based HDL, every circuit in JHDL is an object. Wires, logic gates, and other circuit building blocks are also objects. To be able to instantiate these JHDL building blocks, we import `Logic` and `base` class files, as shown in Lines 2-3. Unless custom designs are created, these classes should always be imported.

Since the FullAdder circuit is an object, it will need to be declared as a class as shown in Line 6. Notice that this class extends `Logic` which is considered to be the parent class and contains methods to build circuits. By extending the parent class, the user will have access to all the parent class functions.

Each circuit object should have output and usually input ports. Ports are used to connect wires to the circuit object when it is instantiated. These ports are described in the `CellInterface`. The static object `cell_interface` describes an array of these ports. Each port must be described by a direction, a name, and a width. For example: `in(‘‘a’’, 1)` describes an input port named “a” that is 1 bit wide. Output ports are described with out(name,width) [43]. Lines 9-15 show the port declarations for the FullAdder circuit.

```
1  // Import the base libraries for JHDL design
2  import byucc.jhdl.base.*;
3  import byucc.jhdl.Logic.*;
4
5  // Our FullAdder is a Java class.  Begin the class declaration here
6  public class FullAdder extends Logic {
7
8    // This is cell's interface (ports)
9    public static CellInterface[] cell_interface = {
10     in("a", 1),
11     in("b", 1),
12     in("cin", 1),
13     out("sum", 1),
14     out("cout", 1)
15   };
16
17   // This is the constructor - it gets called when a new FullAdder is desired
18   public FullAdder(Node parent, Wire a, Wire b,
19                    Wire cin, Wire sum, Wire cout) {
20
21     // Since we extend Logic, always have to call its constructor as the first thing
22     super(parent);
23
24     // Connect the wires passed in as parameters to the constructor to
25     // the ports from the CellInterface above.
26     connect("a", a);
27     connect("b", b);
28     connect("cin", cin);
29     connect("sum", sum);
30     connect("cout", cout);
31
32     // Build our logic as a collection of gates.
33     or_o( and(a,b), and(a,cin), and(b,cin), cout ); /* cout is output */
34     xor_o( a, b, cin, sum );                        /* sum is output */
35
36   }
37 }
```

Figure 5.2: JHDL FullAdder Example

Now that the interface to the FullAdder circuit has been declared, a constructor must be written for the circuit. The example constructor is shown in Lines 18-36. The constructor is called when the circuit is built, and should be a public method.

For the FullAdder circuit, the constructor takes six parameters. The first parameter tells the constructor who the parent of the circuit is. This maintains hierarchy within all of the circuit building blocks. The remaining parameters are the wires that should be connected to its ports. As part of the constructor, the command on Line 22 is always required. This calls the constructor of the parent class that was extended (`Logic` in this case). Next, the wires passed into the constructor are connected to the ports defined in the `CellInterface`, as seen in Lines 24-30.

The last step in the constructor involves defining the logical function of the circuit. Lines 32-34 show that the logic is defined by making calls to `and(), or_o()`, and `xor_o()`. The `Logic` class provides two types of functions to instantiate gates: `gate()` and `gate_o()`. The first function returns the output wire, whereas in the second function, the output wire is one of the parameters. To demonstrate this Line 34 could also have been written as: `sum = xor(a, b, cin)`. The FullAdder constructor is now complete.

To review, the steps described above are [43]:

1. Import the required packages.
2. Declare your circuit as a class which extends `Logic`.
3. Declare the cell interface (ports).
4. Begin the constructor definition for the circuit.
5. Call super(parent).
6. Do connect() calls to hook the constructor parameter wires up to the ports.
7. Insert your logic using calls such as `or_o()` and `and()`.

The JHDL FullAdder circuit construction is now complete. The next example will build up on this example by using it as a sub-block. Figure 5.3 shows an NBitAdder with a width of four, implying that it is built with four FullAdder circuits. Since the JHDL circuit is parameterizable, the width is determined at run-time.

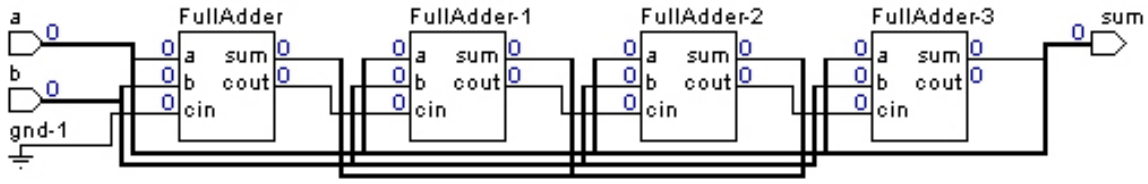

Figure 5.3: JHDL NBitAdder Design

Similar to the FullAdder code discussed, the NBitAdder code shown in Figure 5.4 imports the required packages, declares the circuit as a class, and then defines the ports. The difference here is that the ports can be parameterized to be any specified width. The `param()` command in Line 12 defines width to be an integer input parameter. Subsequent differences occur in the constructor in Lines 17-20. The value for width is set at run-time, and passed in as a parameter. The width must be bound to a value before any `connect()` calls can be made.

Since the FullAdder circuits must be called n times, internal carry wires must be instantiated to connect these circuits within the cell. The `Logic` class provides a method to instantiate wires, as seen in Line 26. Wires can be made any width in JHDL and their bits can be accessed individually through the use of the `gw()` call. Finally, Lines 27-35 show the loop created to instantiate and connect n FullAdder circuits. For more examples, please visit the JHDL website at http://www.jhdl.org.

```
1  // Import the base libraries for JHDL design
2  import byucc.jhdl.base.*;
3  import byucc.jhdl.Logic.*;
4  // Our NBitAdder is a Java class.  Begin the class declaration here
5  public class NBitAdder extends Logic {
6      // This is cell's interface (ports)
7      // The parameter WIDTH is set at run-time
8      public static CellInterface[] cell_interface = {
9          in("a", "WIDTH"),
10         in("b", "WIDTH"),
11         out("sum", "WIDTH"),
12         param("WIDTH", INTEGER)
13     };
14     // This is the constructor - it gets called when a new NBitAdder is desired
15     public NBitAdder(Node parent, Wire a, Wire b, Wire sum) {
16         super(parent);
17         // get width value from an input wire
18         int width = a.getWidth();
19         // width must be bound to a value before connect calls
20         bind("WIDTH", width);
21         // Connect the wires passed in as parameters to the constructor to
22         // the ports from the CellInterface above
23         connect("a", a);
24         connect("b", b);
25         connect("sum", sum);
26         Wire carries = wire(width);  // Intermediate carry wires
27         // Build our logic out of N FullAdder circuits
28         for (int i=0; i < width; i++) {
29             if (i==0)
30                 new FullAdder(this, a.gw(i), b.gw(i), gnd(),
31                     sum.gw(i), carries.gw(0));
32             else
33                 new FullAdder(this, a.gw(i), b.gw(i), carries.gw(i-1),
34                     sum.gw(i), carries.gw(i));
35         }
36     }
37 }
```

Figure 5.4: JHDL NBitAdder Example

5.3 Key Methods to Invoke JHDLBits

In the traditional JHDL design flow, a user has two options to netlist and exercise (assign values to inputs) a circuit. The first option allows the user to open up the JHDL simulation GUI. From within the GUI, the user types commands to netlist the top-level circuit and assign values to the inputs. The circuit can then be simulated by the cycling the clock from the GUI.

The second option is in the form of a programmatic testbench, which builds your circuit, sends stimulus to your circuit, and steps the clock, all using procedural code. A user can also check the output for correctness in the testbench. Examples of programmatic testbenches will be explained later in this chapter.

To use JHDLBits, a user *must* write a programmatic testbench to have access to the techmapper object that the `Logic` class uses to netlist the circuit. The techmapper used by JHDL netlists the top-level circuit and produces the results to an EDIF file. Instead of writing to an EDIF file, JHDLBits uses the `JHDLBitsTechMapper` class as decribed in Section 4.5.1 to netlist the circuit, instantiate JBits primitives, and generate a bitstream.

As part of the programmatic testbench, the `JHDLBitsTechMapper` object is instantiated as the default techmapper. Below is a summary of the `JHDLBitsTechMapper` constructors available to the user depending on the input parameters required to achieve the desired result. Table 5.1 details the constructor parameters. The usage of each constructor will be discussed using examples later in the chapter.

JHDLBitsTechMapper(Device device, String inputFile, String outputFile):

JHDLBitsTechMapper(Device device, Package devicePackage, String inputFile, String outputFile, String ucfFile):

JHDLBitsTechMapper(Device device, String inputFile, String outputFile, ProgressFeedbackInterface pfi):

JHDLBitsTechMapper(Device device, Package devicePackage, String inputFile, String outputFile, String ucfFile, ProgressFeedbackInterface pfi):

Table 5.1: JHDLBits TechMapper Constructor Parameters

Parameter	**Description**	**Example**	**Required/ Optional**
Device *device*	Virtex-II device	XC2V40	Required
Package *devicePackage*	Package type	XC2V40_FG256	Optional
String *inputFile*	Input bitstream	null-2v40.bit	Required
String *outputFile*	Output bitstream	NBitAdder-2v40.bit	Required
String *ucfFile*	User Constraints File	NBitAdder-2v40.ucf	Optional
ProgressFeedbackInterface *pfi*	GUI/Screen Output/ Log file Object	OutputStreamFeedback *osf*	Optional

After the `JHDLBitsTechMapper` is instantiated, the user will have to define `JHDLBitsTechMapper` as the default techmapper used by JHDL's `Logic` class. The following method will accomplish this:

Logic.setDefaultTechMapper(*tm*): Defines `JHDLBitsTechMapper` as the default techmapper used by JHDL's `Logic` class. The *tm* input parameter is the `JHDLBitsTechMapper` object.

The final method the user needs to invoke JHDLBits is the `netlist()` method shown below which is called by the `JHDLBitsTechMapper` object.

tm.netlist(*myNAdd*, *true*, *"dummy.txt"*): The first parameter is the cell to be netlisted. The next parameter is a boolean that determines whether the netlist should be flat (true) or hierarchical (false). The last parameter is required by JHDL since normally

an EDIF file is produced as output. In our case, this string can be treated as a dummy string since no output will be produced (however, passing *null* will cause exception errors).

5.4 Design using JHDLBits Flow

The previous section detailed the methods required to invoke JHDLBits from a top-level testbench file. This section now shows the user how to write a testbench file. The first testbench shown below in Figure 5.5 includes the bare mininum code required to run JHDLBits. This code instantiates the `JHDLBitsTechMapper` which generates a bitstream file; however, the code shown below does not produce any output such as extraction progress statements.

The example below is a testbench written for the NBitAdder circuit described in Figures 5.3 and 5.4 above. First a user must import all the necessary libraries. Line 11 shows that the class `NBitAdderMin_tb` extends the JHDL `Logic` class and also implements the `TestBench` class that interacts with the JHDL simulator. The `TestBench` class is a JHDL interface that describes a top level cell for generating test data to drive the circuit.

Since this testbench file is the top-level file created, a main function must also be written as shown in Lines 12-16. The main function instantiates the `NBitAdderMin_tb` object. Note that this constructor requires a parent cell, so JHDL uses a `HWSystem` as the top-level node for the circuit. All circuit-level issues, such as stepping and cycling a clock are taken care of in this class.

The constructor class is then written in Lines 17-36. The user must specify the names of the input and output bitstreams that are to be used. Next, the `JHDLBitsTechMapper` object is instantiated with device and bitstream parameters in Lines 24-25. The `JHDLBitsTechMapper` must also be set as the default techmapper as shown in Line 27. The user then creates the top level wires or I/O ports. Using these wires, an NBitAdder circuit is called. Finally, the techmapper's `netlist` method is called in Line 35 to start the netlisting process.

```
1  //  JHDLBits (C) 2004 Virginia Tech.  All Rights Reserved.
2  //  Written by the Virginia Tech Configurable Computing Lab
3  import JHDLBits.Virtex2.Extractor.*;
4  import JHDLBits.Util.FileUtil;
5  import byucc.jhdl.base.*;
6  import byucc.jhdl.Logic.*;
7  import byucc.jhdl.examples.*;
8  import com.xilinx.JBits.Virtex2.XC2V40;
9  import java.io.File;
10
11 public class NBitAdderMin_tb extends Logic implements TestBench {
12     public static void main(String args[]) {
13         // instantiate an NBitAdder testbench object
14         // HWSystem is considered the top-level cell
15         NBitAdderMin_tb tb = new NBitAdderMin_tb(new HWSystem());
16     }
17     public NBitAdderMin_tb(Node parent) {
18         super(parent);
19         // define the input and output bitstreams that we intend to use
20         File inputBitstream = new File(FileUtil.NullBitstreamsPath,"null-2v40.bit");
21         File outputBitstream = new File("NBitAdderMin-2v40.bit");
22         // instantiate and install the JHDLBits tech mapper
23         // pass the device, input file, and output file as parameters
24         JHDLBitsTechMapper tm = new JHDLBitsTechMapper(new XC2V40(),
25             inputBitstream.getPath(),outputBitstream.getPath());
26         // Logic class needs to know which tech mapper is used
27         Logic.setDefaultTechMapper(tm);
28         // top level I/O
29         Wire a = wire(4, "a");
30         Wire b = wire(4, "b");
31         Wire sum = wire(4, "sum");
32         // instantiate NBitAdder circuit
33         Cell myNAdd = new NBitAdder(this, a, b, sum);
34         // netlist NBitAdder cell
35         tm.netlist(myNAdd, true, "testnbitadd.txt");
36     }
37 }
```

Figure 5.5: JHDLBits NBitAdder Testbench File (Bare Minimum Required)

If a user wishes to have information presented to the screen and/or to a log file while the extraction process is running, the lines shown in Figure 5.6 can be added to the testbench file above. Note that the `JHDLBitsTechMapper` object instantiation in Lines 12-13 uses a different constructor than in the previous example. The output information on the screen or log file is explained in Chapter 6 Section 6.2.

```
    ⋮
1   import JHDLBits.ArchIndependent.OutputStreamFeedback;
2   import java.io.FileOutputStream;
3
    ⋮
4       public NBitAdderMin_tb(Node parent) {
5           super(parent);
6
7           // create a feedback path to System.out
8           OutputStreamFeedback osf = new OutputStreamFeedback(System.out);
9           try { osf.addStream(new FileOutputStream("NBitAdderMin-2v40.log")); }
10          catch(java.io.FileNotFoundException fnfe) {}
11
    ⋮
12          JHDLBitsTechMapper tm = new JHDLBitsTechMapper(new XC2V40(),
13              inputBitstream.getPath(),outputBitstream.getPath(),osf);
    ⋮
```

Figure 5.6: JHDLBits NBitAdder Testbench File (Screen Output/Log File)

5.5 More Complex Design and JHDLBits Testbench Examples

Now that the user should have a good feel and understanding of how to write a JHDL design along with a testbench to invoke JHDLBits, a slightly more complex JHDL design is shown in Figure 5.7 below. The design is an NBitCounter with a width of four consisting of four FullAdder circuits and four flip-flops. This design was chosen to use more of the JHDLBits primitives available to the designer.

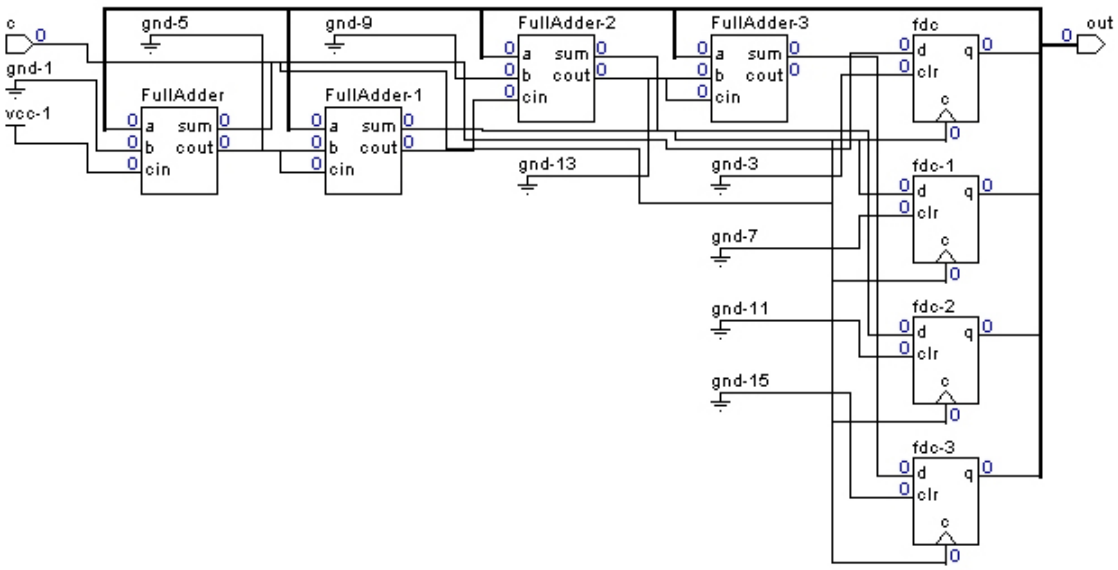

Figure 5.7: JHDL NBitCounter Design

The code for the NBitCounter design above is shown in Figure 5.8. This code is once again parameterizable at run-time. The comments in the code explain the steps of the JHDL design.

```
1  //  JHDLBits (C) 2004 Virginia Tech.  All Rights Reserved.
2  import byucc.jhdl.base.*;
3  import byucc.jhdl.Logic.*;
4  import byucc.jhdl.examples.*;
5  import byucc.jhdl.Logic.LogicGates;
6  /** Simple NBitCounter JHDL Design.  Modified from NBitCounter
7    * found in byucc.jhdl.examples.  Uses JHDL FullAdder cell found
8    * in byucc.jhdl.examples.
9    */
10 public class NBitCounter extends Logic {
11     public static CellInterface[] cell_interface = {
12         out("out", "WIDTH"),
13         param("WIDTH", INTEGER)
14     };
15     /**
16       * Constructs the NBitCounter object.
17       * @param parent The parent cell.
18       * @param out The NBitCounter output wire.
19       */
20     public NBitCounter(Node parent, Wire out) {
21         super(parent);
22         int width = out.getWidth();
23         bind("WIDTH", width);
24         connect("out", out);
25         Wire carries = wire(width);  // intermediate carry wires.
26         Wire sum = wire(width);      // intermediate sum wires
27         for (int i=0; i < width; i++) {
28             if (i==0)
29                 new FullAdder(this, out.gw(i), gnd(), vcc(),
30                     sum.gw(i), carries.gw(0));
31             else
32                 new FullAdder(this, out.gw(i), gnd(), carries.gw(i-1),
33                     sum.gw(i), carries.gw(i));
34             reg_o(sum.gw(i), out.gw(i));
35         }
36     }
37 }
```

Figure 5.8: JHDLBits NBitCounter Example

The rest of this section will cover three different testbench examples for the NBitCounter above. The first example will show the user how the JHDLBits GUI can be invoked. The second example will demonstrate how VTsim is tied into the testbench file. The final example will show how a user constraints file (UCF) can be integrated to create top-level I/O ports so that the JHDLBits design can be used on FPGA hardware.

The JHDLBits GUI was created as part of a demo for the tool. A snapshot of the GUI is shown in Figure 5.9 below. The output in the GUI is the same as that sent to the screen or log file as described in Section 5.4; however, the GUI may be beneficial to a designer that would like to step through the extraction process one part at a time to aid debugging. The GUI has a RUN button that completes the entire extraction process in one step. This can also be RESET and run again, or the user can opt to QUIT. The buttons along the left part of the screen each represent a step in the extraction process that can be run individually.

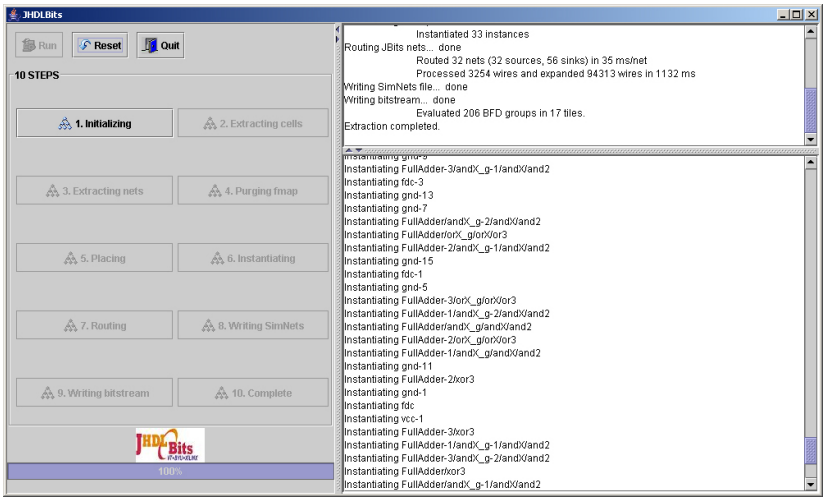

Figure 5.9: JHDLBits GUI Snapshot

The JHDLBits GUI can be invoked from a testbench as shown in the following code. Part 1 of the code is displayed in Figure 5.10. Behind the scenes, the GUI is written in Java Swing, and additional libraries need to be imported as found on Lines 13-15. Libraries to run the GUI are also required and shown on Lines 3 and 4. To allow the JHDLBits GUI to have buttons such as RUN, the testbench must implement the Java `Runnable` interface as shown on Line 16. An additional Java interface `ResetRunnable` is also required in order to service a RESET button. This interface is created on Line 30.

```
1  //  JHDLBits (C) 2004 Virginia Tech.  All Rights Reserved.
2  import JHDLBits.Virtex2.Extractor.*;
3  import JHDLBits.GUI.Virtex2.JHDLBitsDemo.*;
4  import JHDLBits.ArchIndependent.ProgressFeedbackInterface;
5  import JHDLBits.Util.FileUtil;
6  import byucc.jhdl.base.*;
7  import byucc.jhdl.Logic.*;
8  import byucc.jhdl.Xilinx.Virtex2.*;
9  import com.xilinx.JBits.Virtex2.XC2V40;
10 import java.util.*;
11 import java.math.*;
12 import java.io.*;
13 import java.awt.*;
14 import java.awt.event.*;
15 import javax.swing.*;
16 public class NBitCounterGUI_tb extends Logic implements TestBench, Runnable {
17     JHDLBitsTechMapper tm;
18     Cell myNCcount;
19     JHDLBitsDemo frame;
20     public static boolean resetClicked = false;
21     public static void main(String args[]) {
22         HWSystem hw = new HWSystem();
23         NBitCounterGUI_tb tb = new NBitCounterGUI_tb(hw);
24     }
25     public NBitCounterGUI_tb(Node parent) {
26         super(parent);
27         int window_width = 1000;
28         int window_height = 600;
29         // create an object that can reset this test bench
30         ResetRunnable resetter = new ResetRunnable();
31         frame = new JHDLBitsDemo(this,resetter);
32         WindowListener l = new WindowAdapter() {
33             public void windowClosing(WindowEvent e) {
34                 System.exit(0);
35             }
36         };
       ⋮
```

Figure 5.10: JHDLBits NBitCounter Testbench File (GUI) Part 1

Figure 5.11 shows Part 2 of the NBitCounter testbench using the JHDLBits GUI. Lines 37-40 set up the GUI window parameters such as size and bringing the window to the forefront of the user's screen. The `run()` method on Lines 46-51 gets executed whenever the user presses RUN in the GUI window. The `run()` method within `ResetRunnable` on Lines 52-62 takes care of instantiating the `JHDLBitsTechMapper` object. This method is called whenever a user presses RESET on the GUI window. This then resets the techmapper, JBits, and ADB router objects, thus removing all previous extracted information.

```
    ⋮
37          frame.setSize(window_width, window_height);
38          frame.setLocation(0, 0);
39          frame.addWindowListener(l);
40          frame.setVisible(true);
41          // let the resetter create the JHDLBitsTechMapper
42          resetter.run();
43          Wire out = wire(4, "out");      // instantiate top-level I/O
44          myncount = new NBitCounter(this, out);  // create NBitCounter cell
45      }
46      // extract the design
47      public void run() {
48          if(!resetClicked){
49              tm.netlist(myncount, true, "testnbitcount.edn");
50          }
51      }
52      // reset the extractor (and JBits and ADB and the Net class)
53      public class ResetRunnable implements Runnable {
54          public void run() {
55              File inputBitstream = new File(FileUtil.NullBitstreamsPath,"null-2v40.bit");
56              File outputBitstream = new File("NBitCounter-2v40.bit");
57              // instantiate and install the JHDLBits tech mapper
58              tm = new JHDLBitsTechMapper(new XC2V40(),inputBitstream.getPath(),
59                  outputBitstream.getPath(),frame.getProcessPanel());
60              Logic.setDefaultTechMapper(tm);
61          }
62      }
63  }
```

Figure 5.11: JHDLBits NBitCounter Testbench File (GUI) Part 2

Another testbench example is included here to show the user how to tie in VTsim, the device simulator component of the JHDLBits project. Figure 5.12 shows the first part of the testbench code that integrates with VTsim.

JHDL allows values that are considered memory elements to be read back from the actual hardware. This is the approach used to integrate VTsim. Using the testbench, the user can put JHDL in hardware mode allowing values to be read back from the simulator.

Two library files need to be imported to use the simulator. The first is the library for VTsim in Line 3, and the second is the JHDL circuit visualization tool (CVT) in Line 8, which allows the user to see the values of a circuit changing in a schematic viewer. Line 13 shows that the main class created needs to implement `HardwareInterface`. This interface has a few methods that need to be created and which will be explained in Part 2 of the code. Next, Lines 21 and 22 put the JHDL circuit into hardware mode. Two other lines of code the user has not seen before are shown in Lines 32 and 33. In the first, the cycle count is initialized before the circuit is clocked. In the second, a boolean is set to make sure that VTsim is only instantiated once.

Figure 5.13 shows Part 2 of the code used to integrate VTsim. The methods shown below must be implemented as part of the `HardwareInterface` in JHDL. When a user cycles the clock from the CVT GUI, the method `stepHardwareClock(int cycles)` is called as shown in Lines 40-48. The first time the circuit is clocked, VTsim is instantiated and the desired number of clock cycles are simulated. The method `getModuleName()` is required and shown on Line 38, but only returns a string.

Finally, the method `getHardwareState` is called by the `HardwareInterface`. This call happens transparently to the user. In this method, a `JHDLBitsExtractor` object is required to access the `HashMap` that contains all of the cells that are `ExternallyUpdateable`. These are all the JHDL cells which feature the hardware readback capability.

All of the cells in the `HashMap` are iterated to access placement information. This placement information is then passed to VTsim, which will then return the updated values after the clock cycles passed to `stepHardwareClock()` have been executed. The updated values are then reflected in the schematic viewer.

```
1  //  JHDLBits (C) 2004 Virginia Tech.  All Rights Reserved.
2  import JHDLBits.Virtex2.Extractor.*;
3  import JHDLBits.Virtex2.Simulator.*;
4  import JHDLBits.Util.FileUtil;
5  import byucc.jhdl.base.*;
6  import byucc.jhdl.Logic.*;
7  import byucc.jhdl.Xilinx.Virtex2.*;
8  import byucc.jhdl.apps.Viewers.cvt.cvtFrame;
9  import com.xilinx.JBits.Virtex2.XC2V40;
10 import java.util.*;
11 import java.math.*;
12 import java.io.*;
13 public class NBitCounterVTsim_tb extends Logic implements TestBench, HardwareInterface {
14     JHDLBitsTechMapper tm;
15     VTDS vtds;
16     boolean callSim;
17     int cycleCount;
18     public static void main(String args[]) {
19         HWSystem hw = new HWSystem();
20         NBitCounterVTsim_tb tb = new NBitCounterVTsim_tb(hw);
21         hw.setHardwareMode(true);       // tell JHDL to run in hardware mode
22         new cvtFrame(tb);               // JHDL circuit visualization tool
23     }
24     public NBitCounterVTsim_tb(Node parent) {
25         super(parent);
26         File inputBitstream = new File(FileUtil.NullBitstreamsPath,"null-2v40.bit");
27         File outputBitstream = new File("NBitCounter-2v40.bit");
28         // instantiate and install the JHDLBits tech mapper
29         tm = new JHDLBitsTechMapper(new XC2V40(),inputBitstream.getPath(),
30             outputBitstream.getPath());
31         Logic.setDefaultTechMapper(tm);
32         cycleCount = -1;        // initialize cycle count
33         callSim = true;         // boolean used to instantiate the simulator one time
34         Wire out = wire(4, "out");      // create top-level I/O
35         Cell myNCount = new NBitCounter(this, out); // instantiate NBitCounter cell
36         tm.netlist(myNCount, true, "dummy.txt");    // invoke the JHDLBits extractor
37     }
```

Figure 5.12: JHDLBits NBitCounter Testbench File (VTsim) Part 1

```
38      public String getModuleName() { return "JHDLBits Device Simulator"; }
39
40      // steps the hardware clock by number of cycles passed
41      public void stepHardwareClock( int cycles ) {
42          if (callSim) {                  // call VTsim only once
43              vtds = new VTDS();          // instantiate VTsim
44              callSim = false;
45          }
46          cycleCount += cycles;           //# of cycles entered on JHDL cvt
47          vtds.clockUpdate(cycles);       // simulate the # of clock cycles
48      }
49       /** Returns the hardware state
50       * @param eCells the {@link ExternallyUpdateable} cells to look at
51       * @param leCells the {@link LargeExternallyUpdateable} cells to look at
52       * @param cCells the {@link Checkpointable} cells to look at
53       * @return the hardware state  */
54       public StateObject getHardwareState( ExternallyUpdateable[] eCells,
55                      LargeExternallyUpdateable[] leCells,
56                      Checkpointable[] cCells ) {
57          int values[] = new int[eCells.length];      // initialize arrays
58          int largeValues[][] = new int[leCells.length][];
59          for (int j=0; j < leCells.length; j++) {
60              largeValues[j] = new int[leCells.length];
61          }
62          Serializable checks[] = new Serializable[0];
63          // get the extractor object from the JHDLBitsTechMapper
64          JHDLBitsExtractor extractor = tm.getJHDLBitsExtractor();
65          HashMap updateMap = extractor.getUpdateablePlacement();
66          for (int i=0; i < eCells.length; i++) {
67              UpdateablePlacement update = (UpdateablePlacement) updateMap.get(eCells[i]);
68              values[i] = vtds.getFFValueFromJBitsTileCoordinates(update.tileRow,
69                  update.tileCol,update.slice,update.le);
70          }
71          StateObject myState = new StateObject(cycleCount,0,values,largeValues,checks);
72          return(myState);        // return updated values to HardwareInterface
73       }                          // to be reflected in GUI
74  }
```

Figure 5.13: JHDLBits NBitCounter Testbench File (VTsim) Part 2

The last testbench example shown for the NBitCounter involves integrating a user written UCF file. A UCF file, as in the Xilinx Foundation mainstream tool flow, is used for mapping top-level signals in the JHDL design to external pins/pads on the FPGA. This is necessary when a user wants to load the generated bitstream onto a FPGA board. Figure 5.14 below shows an example UCF file that will be used for the NBitCounter testbench in Figure 5.15. For simplicity, the NBitCounter was parameterized for a width of four. The UCF files shows the four top level nets that are connected to pins on the FPGA, along with a clock pin.

```
1 net out<0> loc = N9;
2 net out<1> loc = P9;
3 net out<2> loc = R9;
4 net out<3> loc = T9;
5 net ibufg_clk_in loc = T8;
```

Figure 5.14: 4-BitCounter UCF File

Following are some of the differences to note in this testbench to integrate the UCF file. The first thing a user must do is import a library for the package of the device used as shown in Line 9. The user also needs to specify the path for the UCF as shown in Lines 23 and 24. The `JHDLBitsTechMapper` constructor now includes the package type and UCF path. The last main difference to consider is that the `tm` object needs to know that the top level pads need to be included in the netlist as shown in Lines 29-32.

```
1  //  JHDLBits (C) 2004 Virginia Tech.  All Rights Reserved.
2  //  Written by the Virginia Tech Configurable Computing Lab
3  import JHDLBits.Virtex2.Extractor.*;
4  import JHDLBits.Util.FileUtil;
5  import JHDLBits.ArchIndependent.OutputStreamFeedback;
6  import byucc.jhdl.base.*;
7  import byucc.jhdl.Logic.*;
8  import com.xilinx.JBits.Virtex2.XC2V40;
9  import com.xilinx.JBits.Virtex2.Packages.XC2V40_FG256;
10 import java.io.File;
11 import java.io.FileOutputStream;
12
13 public class NBitCounterUCF_tb extends Logic implements TestBench {
14     public static void main(String args[]) {
15         HWSystem hw = new HWSystem();           //create a hardware system
16         // instantiate and NBitCounter (with UCF) test bench
17         NBitCounterUCF_tb tb = new NBitCounterUCF_tb(hw);
18     }
19     public NBitCounterUCF_tb(Node parent) {
20         super(parent);
21         File inputBitstream = new File(FileUtil.NullBitstreamsPath,"null-2v40.bit");
22         File outputBitstream = new File("NBitCounterUCF-2v40.bit");
23         // define the UCF file that we intend to use
24         File ucfFile = new File(FileUtil.JHDLExamplesPath,"NBitCounter-2v40.ucf");
25         // instantiate and install the JHDLBits tech mapper
26         JHDLBitsTechMapper tm = new JHDLBitsTechMapper(new XC2V40(),new XC2V40_FG256(),
27             inputBitstream.getPath(),outputBitstream.getPath(),ucfFile.getPath());
28         Logic.setDefaultTechMapper(tm);
29         // tell the tech mapper to insert top level ports and clock pads
30         tm.setInsertPads(true);
31         tm.setInsertTopLevelPorts(true);
32         tm.setInsertTopLevelClockPad(true);
33         Wire out = wire(4,"out");                // create top-level I/O
34         Cell cell = new NBitCounter(this, out);  // instantiate NBitCounter cell
35         tm.netlist(cell,true,"dummy.txt");       // invoke the JHDLBits extractor
36     }
37 }
```

Figure 5.15: JHDLBits NBitCounter Testbench File (UCF)

5.6 Summary

This chapter provided information on how to create JHDL designs and write testbench files to invoke JHDLBits. The key methods required in the testbench were also described. The last three testbench files shown for the NBitCounter design showed how to integrate additional JHDLBits components. These examples were explained as separate files, but the user can incorporate them all into one file as desired. The next chapter will detail the results of running these testbench files. The output information generated by JHDLBits, along with running a JHDLBits design on FPGA hardware will be discussed.

Chapter 6

Results

6.1 Overview

The JHDLBits proof of concept, as discussed in Chapter 4 Section 4.2, was successful in that a bitstream was generated for a simple JHDL design using JBits calls. Since then, top-level I/O and clock ports have also been incorporated. This chapter details the JHDLBits results by first describing the output generated by JHDLBits. Next, the extensibility of JHDLBits will be discussed. The successful tie-in of several components such as a placer, ADB router, and VTsim are notable achievements. JHDLBits generated bitstreams are also in the process of being tested on FPGA hardware, and the status of this testing will be mentioned. Finally, the current limitations of JHDLBits will be documented.

6.2 JHDLBits Output

This section describes the output generated by JHDLBits for the GUI, screen, and log file. The output described is generated from the NBitCounter (width of 4) design used throughout Chapter 5. The intention is to give the user additional information to aid in debugging a design.

Figure 6.1 shows a portion of the JHDLBits output during initial set-up, extraction of JHDL logic cells, and the extraction of JHDL nets. Following the initialization of JBits and ADB objects in lines 1 and 2, the JHDL logic cells are extracted in lines 3-23.

While extracting the top-level NBitCounter cell recursively, the first primitive that is found is a GND primitive. Since this primitive has not been encountered before, the primitive and instance information are stored. Recall that an instance is an occurrence of a primitive. Notice that on Line 10 another GND primitive is encountered, but this time only the instance information is extracted. This line also shows that the instance name retains the hierarchy information, which can later be used by a more sophisticated placement algorithm to place related cells near each other. Finally, Line 22 shows that 83 instances for 10 primitive types were extracted in the NBitCounter design.

Lines 24-33 give details of the JHDL nets extraction. These nets represent the connectivity for all the instances extracted above. When a net is extracted, the name along with the source and sink information is stored. Line 32 shows that a total of 34 nets are extracted for the NBitCounter design.

Figure 6.2 shows the portion of the JHDLBits process that includes purging unused primitives, placing and instantiating JBits primitives, routing nets, writing the VTsim SimNets file, and generating a bitstream.

Lines 34-40 show the purged instances that are not supported as JBits primitives. The JHDL libraries include a primitive called FMAP which contains directives for the Xilinx mainstream tools to pack together certain related instances in one tile. This primitive is not supported by JHDLBits because our future placement algorithm will be responsible for packing related instances together. Therefore all instances of FMAP are purged. These FMAP instances have GND instances associated with them that are also purged. Line 39 shows that 44 FMAP instances were purged in this design.

Next, Lines 41-49 detail the locations where the placement algorithm has placed some of the instances. For example, Line 44 places an OR3 instance in CLB Row 0 and Column 0, in slice 1, and LE 0. Line 48 shows that 39 instances were placed for the NBitCounter design.

Once all the instances have been placed, the corresponding 39 JBits primitives can be in-

```
Initializing JBits and ADB...
    done
Extracting JHDL logic cells...
    Extracting primitive gnd
    Extracting instance gnd-1
    Extracting primitive vcc
    Extracting instance vcc-1
    Extracting primitive and2
    Extracting instance FullAdder/andX_g/andX/and2
    Extracting instance FullAdder/andX_g/andX/gnd-1
    Extracting instance FullAdder/andX_g/andX/gnd-3
    Extracting primitive fmap
    Extracting instance FullAdder/andX_g/andX/fmap
    Extracting instance FullAdder/andX_g-1/andX/and2
    Extracting instance FullAdder/andX_g-1/andX/gnd-1
    Extracting instance FullAdder/andX_g-2/andX/gnd-3
    Extracting instance FullAdder/andX_g-2/andX/fmap
    Extracting primitive or3
    ⋮
    Extracting primitive obuf
    Extracting instance out_OBUF_2
    Extracting instance out_OBUF_3
    Extracted 83 instances of 10 types
    done
Extracting JHDL nets...
    Extracting net out_OPAD_OUT
    ⋮
    Extracting net FullAdder-3/and_out
    Extracting net FullAdder-3/andX_g/andX/gnd
    Extracting net FullAdder-3/and_out-2
    Extracting net FullAdder-3/andX_g-2/andX/gnd
    Extracting net FullAdder-3/andX_g-2/andX/gnd-2
    Extracting net FullAdder-3/orX_g/orX/gnd
    Extracted 68 nets
    ⋮
    done
```

Figure 6.1: JHDLBits Output Part 1

```
Purging fmap directives...
    Purging FullAdder-3/andX_g-1/andX/fmap
    Purging FullAdder-3/andX_g-1/andX/gnd-1
    Purging FullAdder-3/andX_g-1/andX/gnd-3
    Purging FullAdder/andX_g-2/andX/fmap
    ⋮
    Purged 44 fmap instances
    done
Placing JBits primitives...
    Placing gnd at CLB [0][0] slice 0, LE 0
    Placing and2 at CLB [0][0] slice 0, LE 1
    Placing or3 at CLB [0][0] slice 1, LE 0
    Placing and2 at CLB [0][0] slice 1, LE 1
    Placing and2 at CLB [0][0] slice 2, LE 0
    Placing xor3 at CLB [0][0] slice 2, LE 1
    ⋮
    Placed 39 instances in 3 ms
    done
Instantiating JBits primitives...
    Instantiating FullAdder-2/andX_g/andX/and2
    Instantiating FullAdder-1/orX_g/orX/or3
    Instantiating FullAdder-2/andX_g-2/andX/and2
    Instantiating out_OBUF_2
    ⋮
    Instantiated 39 instances
    done
Routing JBits nets...
    Routed 34 nets (34 sources, 65 sinks) in 11 ms/net
    Processed 1832 wires and expanded 76607 wires in 379 ms
    done
Writing SimNets file...
    done
Writing bitstream...
    Evaluated 212 BFD groups in 28 tiles.
    done
Extraction completed.
```

Figure 6.2: JHDLBits Output Part 2

stantiated as shown in Lines 50-56. Internally, JBits calls are made to configure the FPGA resources based on the placement information. This includes setting the internal logic along with the input and output connections for that instance. Lines 57-60 then show how ADB is used to route the internal connections.

Lines 61-66 show the last few steps of the JHDLBits process. On Line 61, the `SimNets` file is written. This file is used by VTsim, the device simulator. It contains a list of all the nets along with source and sink information for each net. This allows the simulator to continue using the same names established in this design, since the bitstream file, alternatively used for simulation, is stripped of all symbols and names. Next, line 63 shows that a bitstream has been generated. A total of 28 tiles in the FPGA were used. Finally, line 66 states that the JHDLBits extraction process has been completed.

6.3 Tie-in of Components

An important goal of the JHDLBits project was to create a JHDLBits core that was extensible in the sense that users could easily plug in their own components. The JHDLBits core is simply responsible for extracting information from a JHDL design and then using JBits calls to generate a bitstream. The other components, including the placer, router, and device simulator, can be selected to meet the user's specific needs. Currently, the JHDLBits project uses a simple placer, the ADB router, and the VTsim device simulator; however, users are free to develop alternatives.

6.3.1 Placer

Figure 6.3 shows the structure of the placement API. The main class is the `Placer` class, which is defined as an abstract class. Each placement algorithm written must extend and define the abstract methods of the `Placer` class. As mentioned before, the JHDLBits project currently includes the `SimplePlacer` class, but the `MacroPlacer` is currently being developed.

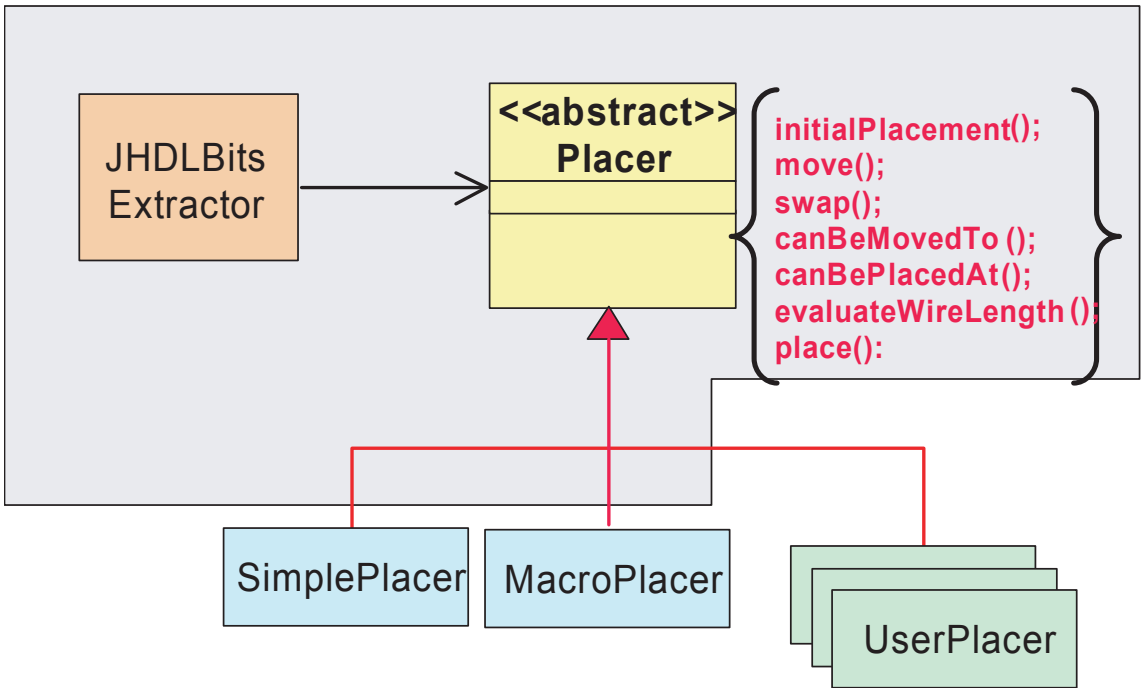

Figure 6.3: JHDLBits and Placer Tie-In

The placer object, whether it be the default placer included with JHDLBits or a user-created placer, is currently instantiated in the `JHDLBitsExtractor` class. This does not allow easy access to the object from a user standpoint. The proposed change for this would be to include methods within the `JHDLBitsTechMapper` class that allow the user to opt for either the default or a user created placer. These methods in the `JHDLBitsTechMapper` class could then be called from the testbench. The `JHDLBitsExtractor` could now access the placer object through the `JHDLBitsTechMapper`. It might be worthwhile to send the placer object onto the JHDLBits `Bitstream` class which currently stores the JBits and ADB objects. Most objects would then be stored in one central location.

6.3.2 Router

The JHDLBits project currently uses the router included with ADB. The ADB router has been tied into JHDLBits using the `RouterInterface`, as defined by JBits. The `RouterInterface` links ADB with JBits, thus giving the JHDLBits flow access through the JHDLBits `Bitstream` class.

If a user would like to write their own router, they would have to implement the `RouterInterface` and all of the corresponding methods to be able to tie into JBits. More information on the JBits `RouterInterface` can be found in the JBits Javadocs (`com.xilinx.JBits.ArchIndependent`) that can be downloaded as part of the JBits3 SDK [22]. In addition to implementing the `RouterInterface`, a user implemented router would have to either incorporate the wire database of ADB or the wire database of JBits.

The router object is currently accessed in the `JHDLBitsExtractor` class, the same way as described with the placer object above. The same proposed strategy of including methods in the `JHDLBitsTechMapper` class should be implemented for the router object. This object should also be stored in the JHDLBits `Bitstream` class.

JHDLBits is built upon the ADB wire database even though there is a wire database included with JBits. Because of this, ADB is required for JHDLBits to function even if the ADB router is not used.

6.3.3 Device Simulator

VTsim, a Virtex-II device simulator, is another component plugged into the JHDLBits flow. The user also has the option of replacing this component; however, VTsim provides very good performance when compared to VirtexDS (the device simulator written for JBits2.8), of which the user may wish to take advantage. Figures 6.4 and 6.5 taken with permission from [27] show comparisons between VTsim and VirtexDS for a simple design and a more complex design.

Figure 6.4 represents a simple 10-bit counter design which was run for 300,000 cycles. For

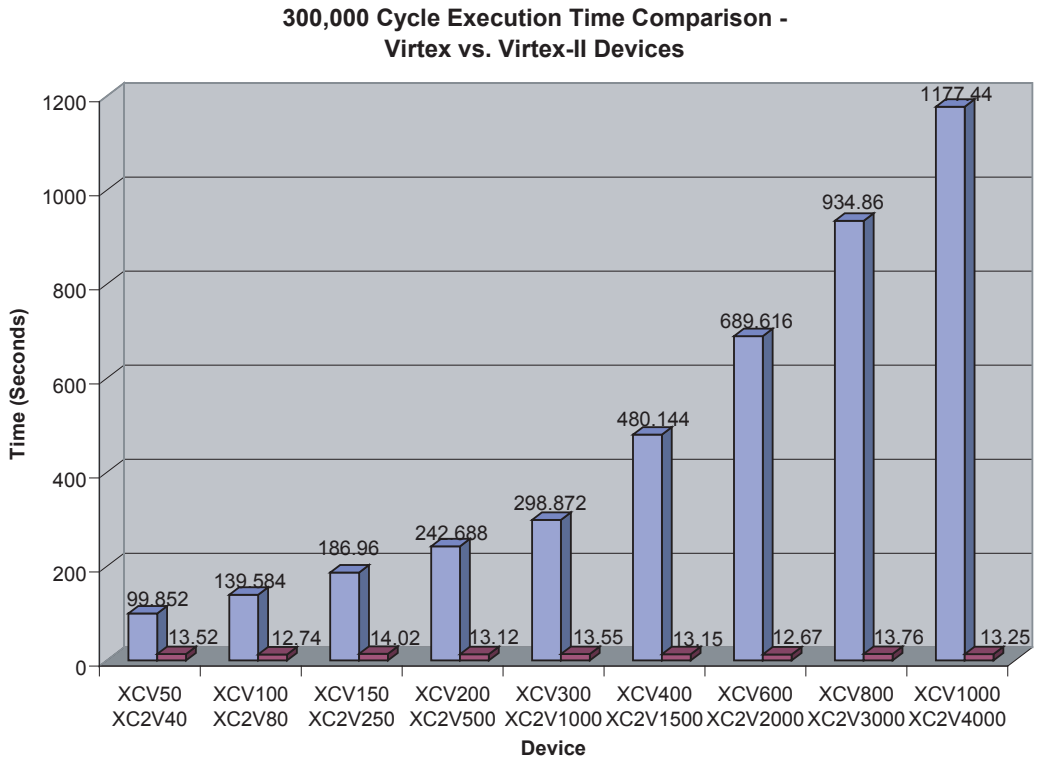

Figure 6.4: VTsim versus VirtexDS for a Simple Design [27]

the largest device in each family, VTsim (shown on the right) took approximately 13 seconds, while VirtexDS (shown on the left) took approximately 19.5 minutes. In this case, VTsim is about 90 times faster than VirtexDS for the same design.

Figure 6.5 displays results for a more complex design, where the entire chip was filled with logic gates and run for 1,000 cycles. VTsim took approximately 81 seconds, while VirtexDS took approximately 170 seconds. For the complex design, VTsim was still two times faster that VirtexDS.

As discussed in Chapter 5 Section 5.5, VTsim can be tied into the JHDLBits flow through the use of a testbench file. This incorporates the JHDL `HardwareInterface` libraries to allow the user to view results from VTsim in the JHDL GUI simulation and debugging environment.

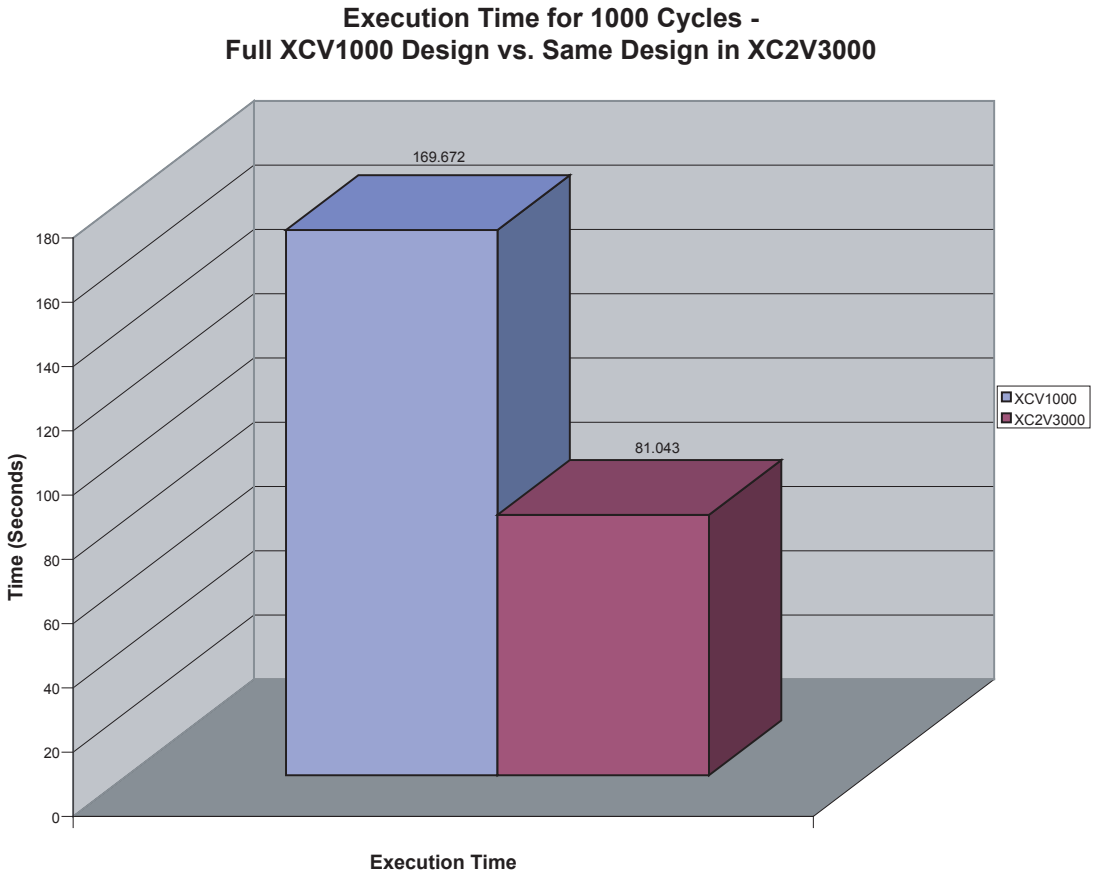

Figure 6.5: VTsim versus VirtexDS for a Complex Design [27]

Figure 6.6 shows a schematic view of the NBitCounter (width of 4) design. Updated values coming directly from VTsim are displayed for the four flip-flops below. The NBitCounter is currently at a count of 3. The non-memory elements (i.e. simple logic) are still simulated behaviorally based on the values read back from VTsim.

Figure 6.7 is another simulation view available in the JHDL GUI environment. Instead of displaying simulation values in a schematic, a waveform view is shown. The values for the `out` signal are being updated from VTsim.

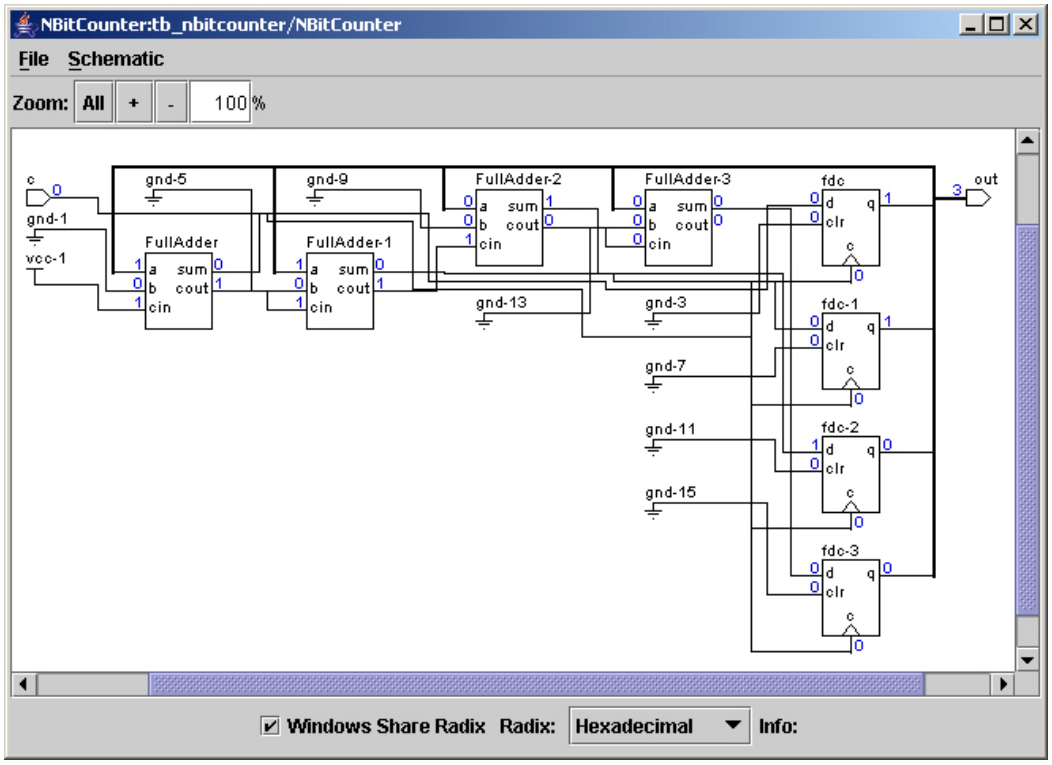

Figure 6.6: JHDL and VTsim NBitCounter Simulation Schematic View

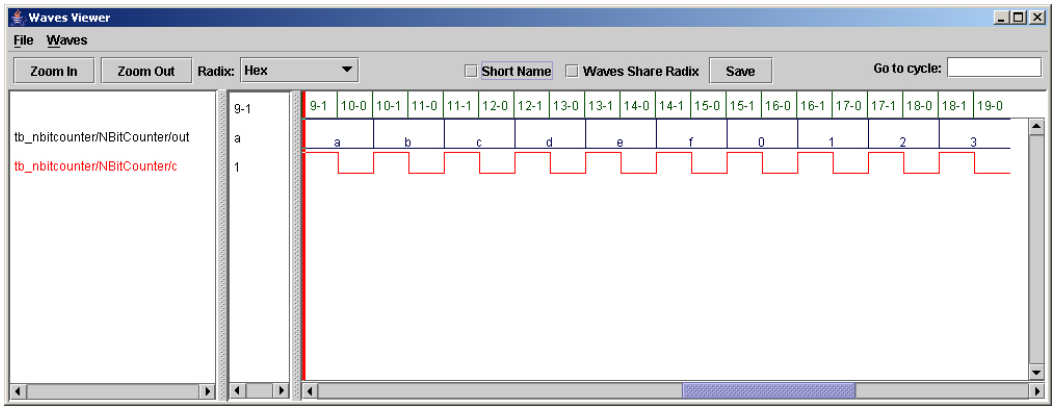

Figure 6.7: JHDL and VTsim NBitCounter Simulation Waveform View

6.4 FPGA Hardware

To ensure that a JHDLBits design is functionally correct, a user can simulate the design using VTsim; however, a developer may also wish to run the design in hardware. Testing JHDLBits primitives and designs in hardware is an important part of the JHDLBits development process, and ultimately the goal of the project.

At the time of this publication, the JHDLBits team is working on testing various primitives in hardware. As a reminder, the JHDLBits primitives are created using JBits3 calls to program resources.

It has been verified that the input, output, and clock JHDLBits primitives work correctly. Additional primitives such as flip-flops also are functioning. All of these primitives are part of a 3-bit counter design, which is being tested using a Memec V2MB1000 development board that features a Xilinx XC2V1000 FPGA.

6.5 Limitations

JHDLBits will hopefully be useful to the FPGA design and research communities. To make JHDLBits a very powerful design automation tool, additional work is required.

For JHDLBits to be successful, additional work needs to be completed in expanding the JHDLBits primitive library. Developing more primitives would allow the user higher complexity JHDL designs. This is an area where the open source nature of JHDLBits will hopefully have an impact. The research community could become involved in writing additional primitives as they are needed for JHDL designs. A newly written primitive would have to extend the JHDLBits `ULPrimitive` class. If a user would not like to include self-written primitives in the JHDLBits tree, the file path pointing to the location of the primitive(s) written could be passed in the function `addULPrimitives()`. This function is part of the `ULPrimitive` package, which could then locate and access the primitives from a different directory.

As mentioned above, additional designs and primitives need to be tested on hardware to verify that the primitives function as expected. Communicating with the target hardware currently only involves downloading the bitstream generated. Future capabilities should also include the ability to communicate with a development board through the Xilinx HardWare InterFace (XHWIF). This interface allows a user to readback the bitstream from a board. XHWIF is currently not functioning correctly, and we are unsure whether the problems lie with JBits or our usage of XHWIF commands.

The JHDLBits team has also tried to analyze performance of JHDLBits such as memory usage. A stress test consisting of a design using most of the FPGA resources is a good indicator of the memory usage of JHDLBits. This design was created, but could not be routed due to limitations with the current simple placement algorithm. The algorithm places primitives in a random order from left to right on the FPGA. This structure created many routes that ran left to right on the chip, exhausting all of the available routing resources without completing the design. To remedy this, a more sophisticated placement algorithm is being developed. This hierarchical algorithm will allow related primitives to be placed near each other permitting easier debugging as well as optimizing routing performance.

Another important capability of JHDLBits would include the ability to take advantage of the partial reconfiguration functions available through JBits. At the time of this publication, ideas of how to address partial reconfiguration have been discussed, but no development has been initiated.

6.6 Summary

This chapter detailed some of the results achieved throughout the JHDLBits project, along with some of the current limitations. The next chapter will summarize this thesis, and present some ideas for future directions to ensure success of the JHDLBits open source project in the community.

Chapter 7

Conclusion

7.1 Summary

This thesis presented JHDLBits, an open source FPGA design tool that merges the features of JHDL and JBits. JHDLBits offers a user the low-level control of FPGA resources provided by JBits, while still being able to create a design at a higher abstract level. JHDLBits has proven to successfully generate bitstreams for a variety of JHDL designs.

The JHDLBits API has also been thoughtfully developed to allow for the extensibility of multiple components. This offers a designer the flexibility of writing their own components such as a placer or router specific to their individual project needs.

JHDLBits strives to be an open source project hosted on SourceForge, which will hopefully draw on the collective knowledge and generate valuable feedback from the FPGA design community.

7.2 Future Directions

A goal of the JHDLBits project has been to retain the properties and philosophies of JBits relating to the partial and run-time reconfiguration of an FPGA device. Following this principle, it is important to take advantage of these abilities through the JHDLBits design flow. This area will be addressed in the near future.

Another goal is to allow for partitioning of the design, so that the user can run part of the JHDL design through the mainstream tools and part through JBits via the JHDLBits Extractor. The developers would also like to investigate using JHDLBits in an embedded system, possibly using a small subset of JHDL and ADB rewritten in a language other than Java to reduce memory usage.

Finally, in preparation for an open source release, additional documentation needs to be written to allow a user a smooth start in using JHDLBits. Many decisions regarding the strategies presented in Chapter 3 still need to be made. The JHDLBits source code along with additional information will be found at http://sourceforge.net/project/jhdlbits.

Bibliography

[1] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," in *Proceedings of the 7th International Conference on Field-Programmable Logic and Applications, FPL '97*, pp. 213–222, August 1997.

[2] S. A. Guccione and D. Levi, "XBI: A Java-Based Interface to FPGA Hardware," in *Configurable Computing: Technology and Applications, Proceedings of SPIE* (J. Schewel, ed.), vol. 3526, (Bellingham, Washington), pp. 97–102, November 1998.

[3] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, "A CAD Suite for High-Performance FPGA Design," in *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 1999*, pp. 12–24, April 1999.

[4] S. A. Guccione, D. Levi, and P. Sundararajan, "JBits: A Java-based Interface for Reconfigurable Computing," in *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.

[5] Open Source Development Network, Inc. ("OSDN"), "SourceForge.net: the world's largest open source software development website." http://www.sourceforge.net/.

[6] S. D. Brown, "An Overview of Technology, Architecture and CAD Tools for Programmable Logic Devices," in *Proceedings of the IEEE Custom Integrated Circuits Conference, 1994*, pp. 69–76, May 1994.

[7] H. Verma, "Field Programmable Gate Arrays," *IEEE Potentials*, vol. 18, pp. 34–36, October–November 1999.

[8] P.-T. Wang, K.-N. Chen, and Y.-T. Lai, "A High Performance FPGA with Hierarchical Interconnection Structure," in *1994 IEEE International Symposium on Circuits and Systems (ISCAS '94)*, vol. 4, pp. 239–242, June 1994.

[9] Xilinx, Inc., "Xilinx FPGA Product Tables." http://www.xilinx.com/products/tables/fpga.htm#v2.

[10] Xilinx, Inc., "Virtex-II Platform FPGAs: Complete Data Sheet." http://direct.xilinx.com/bvdocs/publications/ds031.pdf, April 2004.

[11] Xilinx, Inc., "Design Tools Center." http://www.xilinx.com/products/design_resources/design_tool/index.htm.

[12] Synplicity, Inc., "Synplify & Synplify Pro." http://www.synplicity.com/products/synplifypro/index.html.

[13] E. Lechner and S. A. Guccione, "The Java Environment for Reconfigurable Computing," in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, FPL '97*, pp. 284–293, Springer-Verlag, 1997.

[14] Xilinx, Inc., "XC6200 Development System," 1997.

[15] S. W. Gehring and S. H.-M. Ludwig, "Fast Integrated Tools for Circuit Design with FPGAs," in *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, pp. 133–139, ACM Press, 1998.

[16] Atmel, "Configurable Logic: Design & Application Book," 1995.

[17] N. Wirth, "The Language Lola and Programmable Devices in Teaching Digital Circuit Design," in *Proceedings of the 2nd International Andrei Ershov Memorial Conference*, Springer Verlag, 1996.

[18] Altera, "Quartus II University Interface Program (QUIP)." http://www.altera.com/education/univ/quip/quip-overview.html.

[19] Brigham Young University, "JHDL Home Page." http://www.jhdl.org/.

[20] A. Poetter, J. Hunter, C. Patterson, P. Athanas, B. Nelson, and N. Steiner, "JHDLBits: The Merging of Two Worlds," in *Proceedings of the 14th International Workshop on Field-Programmable Logic and Applications, FPL '04*, (Antwerp, Belgium), August 2004.

[21] Brigham Young University, "Introduction to Creating Logic Descriptions with JHDL." http://www.jhdl.org/docs/docs/usersManual/intro.html.

[22] Xilinx, Inc., "JBits 3.0 SDK for Virtex-II." http://www.xilinx.com/labs/projects/jbits/.

[23] N. Steiner, "A Standalone Wire Database for Routing and Tracing in Xilinx Virtex, Virtex-E, and Virtex-II FPGAs," Master's thesis, Virginia Tech, August 2002.

[24] R. Fong, S. Harper, and P. Athanas, "A Versatile Framework for FPGA Field Updates: An Application of Partial Self-Reconfiguration," in *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping*, (San Diego, California), p. 117, June 2003.

[25] N. Steiner and P. Athanas, "An Alternate Wire Database for Xilinx FPGAs," in *Proceedings of the Twelfth Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2004*, (Napa, California), April 2004.

[26] Eric Keller, "JRoute: A Run-Time Routing API for FPGA Hardware," in *Parallel and Distributed Processing, Lecture Notes in Computer Science* (J. Romlin et al., ed.), pp. 874–881, Springer-Verlag, Berlin, May 2000.

[27] J. Hunter, "A Device-Level FPGA Simulator," Master's thesis, Virginia Tech, June 2004.

[28] J. Hunter, P. Athanas, and C. Patterson, "VTSim: A Virtex-II Device Simulator," in *Proceedings of the 2004 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'04)*, (Las Vegas, Nevada), June 2004.

[29] S. P. McMillian, B. J. Blodget, and S. A. Guccione, "VirtexDS: A Device Simulator for Virtex," in *Reconfigurable Technology: FPGAs for Computing and Applications II, Proceedings of SPIE*, vol. 4212, (Bellingham, Washington), pp. 50–56, November 2000.

[30] C. DiBona, S. Ockman, and M. Stone, eds., *Open Sources: Voices from the Open Source Revolution*, ch. A Brief History of Hackerdom, pp. 19–29. O'Reilly and Associates, Inc., 1999.

[31] C. DiBona, S. Ockman, and M. Stone, eds., *Open Sources: Voices from the Open Source Revolution*, ch. The GNU Operating System and the Free Software Movement, pp. 53–70. O'Reilly and Associates, Inc., 1999.

[32] S. Weber, *The Success of Open Source*, ch. The Early History of Open Source, pp. 20–53. Harvard University Press, 2004.

[33] J. Feller and B. Fitzgerald, *Understanding Open Source Software Development*, ch. A history of Open Source Software, pp. 27–40. Addison-Wesley, 2002.

[34] S. Weber, *The Success of Open Source*, ch. A Maturing Model of Production, pp. 94–127. Harvard University Press, 2004.

[35] C. DiBona, S. Ockman, and M. Stone, eds., *Open Sources: Voices from the Open Source Revolution*, ch. Introduction, pp. 1–17. O'Reilly and Associates, Inc., 1999.

[36] J. Feller and B. Fitzgerald, *Understanding Open Source Software Development*, ch. Overview of Open Source Software, pp. 9–25. Addison-Wesley, 2002.

[37] Open Source Inititative, "Open Source Definition (Version 1.9." http://www.opensource.org/docs/definition.html.

[38] E. S. Raymond, *The Cathedral and the Bazaar:Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly and Associates, Inc., 1999.

[39] S. Weber, *The Success of Open Source*, ch. What Is Open Source and How Does It Work?, pp. 54–93. Harvard University Press, 2004.

[40] Open Source Inititative, "The Approved Licenses." http://www.opensource.org/licenses/.

[41] C. DiBona, S. Ockman, and M. Stone, eds., *Open Sources: Voices from the Open Source Revolution*, ch. The Open Source Definition, pp. 171–188. O'Reilly and Associates, Inc., 1999.

[42] Open Source Inititative, "The BSD License." http://www.opensource.org/licenses/bsd-license.html.

[43] Brigham Young University, "JHDL Getting Started." http://www.jhdl.org/documentation/starter.html.

Vita

Alexandra Vanessa Poetter was born on December 7, 1974 in Freiburg, Germany. She immigrated to the United States along with her parents and sister in 1983, where they settled and bought a home in North Prairie, Wisconsin. After graduating from Mukwonago High School in 1993, Alex attended the University of Wisconsin-Madison and graduated with a B.S. in Electrical Engineering/Computer Engineering Option in December, 1998.

Following graduation, Alex moved to Fort Collins, Colorado to start a full-time job at LSI Logic as a Test Engineer. There she met Kevin Paar, her future husband, and decided to relocate to Blacksburg, Virginia with him in 2001. She applied to Virginia Tech to pursue a Master's Degree in Computer Engineering. Alex worked as a Graduate Teaching Assistant for a few semesters, and then was given the opportunity to join the Configurable Computing Lab as a Research Assistant. While attending graduate school, Alex married Kevin in August of 2002.