A Device-Level FPGA Simulator

Jesse Everett Hunter III

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

Peter Athanas, Chair
Cameron Patterson
Joseph Tront

June 10, 2004
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

keywords: FPGA, Device Simulator, JBits, JHDL, JHDLBits, Xilinx, Virtex-II, VTsim

Copyright © 2004, Jesse Everett Hunter III. All Rights Reserved.

A Device-Level FPGA Simulator

Jesse Everett Hunter III

Abstract

In the realm of FPGAs, many tool vendors offer behaviorally-based simulators aimed at easing the complexity of large FPGA designs. At times, a behaviorally-modeled design does not work in hardware as expected or intended. VTsim, a Virtex-II device simulator, was designed to resolve this and many other design problems by providing a window into the FPGA fabric via a virtual device. VTsim is an event-driven device simulator modeled at the CLB level with multiple clock domain support. Utilizing JBits3 and ADB, VTsim enables simulation and examination of all resources within an FPGA via a virtual device. The only input required by VTsim is a bitstream, which can be generated from any tool suite. The simulator is part of the JHDLBits open-source project, and was designed for rapid response, low memory usage, and ease of interaction.

To my loving wife, Karen, who supported me during this grand endeavor. You cease to amaze me with your brilliance, compassion, and happiness. Thank you for being you.

Acknowledgements

I want to thank my advisor, Dr. Athanas, his guidance throughout my graduate career at Virginia Tech, outstanding Configurable Computing course, and for always being willing to go the extra mile to help me out on this great endeavor. Without your help, VTsim would have never gotten off the ground.

Thanks to Dr. Patterson, who always had an open door and was willing to let me pick his brains about JBits and FPGA architectures; even if I didn't get it the first or even the second time.

Thanks to Dr. Tront for your outstanding courses; I learned more than I could possibly ever imagine. Thank you for the time spent analyzing simulator models and the productive brainstorming sessions.

Thanks to the excellent faculty and staff at Virginia Tech who helped me reach this point in my life through interesting and exciting coursework.

Thanks to my wife, Karen, who was always willing to listen to my incessant simulator ramblings, and help proofread my thesis no matter what time of day, or in some cases night, it was.

Thanks to my parents, Jesse and Cary, my sister Alicia, mother-in-law Jean, sister-in-laws

Michele and Laura, wife Karen, and the rest of my family. I am who I am because of you - Thanks.

Thanks to Alex Poetter for being a great friend and for keeping me sane during the late night, last minute CCM Lab simulator and thesis writing sessions.

Thanks to Neil Steiner for ADB, the countless whiteboard brainstorming sessions, and broadening my knowledge of Java; even if it always included a plug for C++.

Thanks to all the people who had the unfortunate pleasure of sitting around me throughout my time in the CCM Lab: Tony, Stephen, Tom, Alex, Neil, Dong Kwan, and the new guys Sid and James. How you ever put up with my tapping, annoying music, and constant questions, I will never know.

Thanks to the rest of the CCM Lab crew for welcoming me into the lab, the always fun lunch outings, and frequent Ultimate Frisbee breaks.

Thanks to all the CEL friends I have made throughout my time at Virginia Tech, especially this semester: Ken, the other Ken, Dan, Brian, the other Brian, Ed, Tim, Joe, Ricky, Vinit, Chaky, Keith, Bob, and all the former TAs. You guys are the finest group of TAs I have ever known.

Thanks to all the current and former members of the JHDLBits team.

And thanks to everyone that I forgot to include here; there are so many people who have helped me get to this point in my life, and for that, I am forever in your debt.

Table of Contents

Table of Contents vi

List of Figures x

List of Tables xiii

Glossary xiv

1 Introduction **1**

1.1 Overview 1

1.2 Objectives 3

1.3 Organization 4

2 Background **6**

2.1 Overview 6

2.2 General FPGA Information 6

2.3 Xilinx FPGAs 9

2.3.1 Virtex-II CLB 10

2.3.2 Virtex-II IOB 13

2.3.3 Virtex-II Clock Tiles 14

2.4 Related Work: JHDL, JBits, JHDLBits and ADB 15

2.4.1 JHDL 16

2.4.2 JBits and ADB Background 17

2.4.3 JHDLBits 18

2.5 Related Work: Simulators 19

3 JHDLBits and JBits Enhancements **22**

3.1 Overview 22

3.2 Enhancements 23

4 Simulator Design **26**

4.1 Overview 26

4.2 Approaches 26

4.3 Design Organization 31

4.4 Implementation 33

4.5 Tile Organization 33

4.5.1 CLB and Slice Models 34
4.5.2 Slice Design & Testing 35
4.5.3 Slice Interconnect Scheme 37
4.5.4 Utilizing JBits for CLB Configuration 38
4.5.5 CLB Connectivity 40
4.5.6 IOB Tiles 42
4.5.7 CLKT and CLKB Tiles 42
4.6 Event Queue Models 43

5 Simulator Usage **46**

5.1 Overview 46
5.2 VTsim Key Classes and Methods 47
5.3 JHDLBits Design Flow 50
5.4 JBits Design Flow 51
5.5 Other Design Flows 53

6 Results **56**

6.1 Overview 56
6.2 Input Test Files 57

6.3 Event Queue Comparisons 59

6.4 Detailed Memory Analysis 62

6.5 Simple Design Results 65

6.5.1 ADB Trace Time 66

6.5.2 Virtual Device Creation Time 67

6.5.3 SimNet Creation Time 68

6.5.4 Stabilization Time 69

6.5.5 Simulation Execution Time 70

6.6 Small and Large Design Comparisons 71

6.7 VirtexDS Comparisons 74

6.8 VirtexDS vs. VTsim 75

7 Conclusions 79

7.1 Summary 79

Bibliography 81

List of Figures

2.1 Four Major FPGA Design Areas 7

2.2 Internal FPGA fabric 8

2.3 Virtex-II FPGA TileMap 10

2.4 Functional LE layout 11

2.5 I/O banks for wire-bond (left) and flip-chip (right) packages 14

2.6 Global clock buffer configurations 15

2.7 JHDL constituants 16

2.8 The JHDLBits Design Flow 18

3.1 Bitstream Verification Process 25

4.1 Sample Infinite Loop 28

4.2 AND to Flip-Flop Circuit 36

4.3 FPGA Editor: AND to Flip-Flop Circuit 37

4.4 Internal CLB connections 38

4.5 CLB configuration code snippet 39

5.1 The six VTDS initialization steps 48

5.2 Simple oscillator circuit 51

5.3 HelloVTsim example code 52

5.4 HelloTile example code 54

6.1 Ten-Bit Counter Design 57

6.2 Stress Test Design 58

6.3 VTsim: 10-Bit Counter Memory Profile 63

6.4 VTsim: Stress Test Memory Profile 64

6.5 ADB Trace Time for Virtex-II Devices - 10-Bit Counter Design 66

6.6 Virtual Device Creation Time for Virtex-II Devices - 10-Bit Counter Design 67

6.7 SimNet Creation Time for Virtex-II Devices - 10-Bit Counter Design 68

6.8 Stabilization Time for Virtex-II Device - 10-Bit Counter Design 69

6.9 300,000 Cycle Execution Time for Virtex-II Devices - 10-Bit Counter Design 70

6.10 Cycles-per-Second Analysis of Virtex-II Devices - 10-Bit Counter Design . . 71

6.11 Comparison Between Small and Large Designs 72

6.12 Comparison Between Small and Large Designs 73

6.13 300,000 Cycle Execution Time for Virtex Devices - 10-Bit Counter Design . . 74

6.14 Cycles-per-Second Analysis of Virtex Devices - 10-Bit Counter Design 75

6.15 300,000 Cycle Execution Time Comparison - Virtex vs. Virtex-II Devices . . 76

6.16 Virtex and Virtex-II: 1,000 Cycle Execution Time for the Stress Test 77

6.17 Virtex and Virtex-II: Cycles-per-Second Comparison for the Stress Test . . . 78

List of Tables

2.1 Sizes and Types of SelectRAM 12

4.1 Granularity Comparisons 29

6.1 Event Comparisons for 1,000 Clock Cycles 60

6.2 VTsim memory usage 62

Glossary

A glossary and brief explanation of common terms used throughout this thesis is presented below. The list is organized alphabetically.

API: An Application Programming Interface (API) is a complete program that performs specific low-level functions directly for the user.

ASIC: An Application Specific Integrated Circuit (ASIC) is a custom IC developed for a specific target application and cannot be reprogrammed.

CLB: A Configurable Logic Block (CLB) is the most prevalent tile type in an FPGA. Each CLB contains four slices and two tri-state buffers.

DCI: Digitally Controlled Impedance (DCI) provides controlled impedance termination.

DCM: A Digital Clock Manager (DCM) provides several clock management features ranging from clock de-skew, frequency synthesis, to phase shifting.

DDR: Dual Data Rate (DDR) is a standard specifying that data is transmitted on both the rising edge and falling edge of a clock pulse.

Fast Carry Look-Ahead Adder: Each CLB contains dedicated fast carry look-ahead adder chains provided to implement large high speed adders.

FPGA: A Field Programmable Gate Array (FPGA) is a general purpose reprogrammable chip that can be programmed to carry out a specific hardware function.

GUI: A Graphical User Interface (GUI) provides a graphical program for user interaction.

IOB: An Input/Output Block (IOB) provides access to and from the internal FPGA fabric. IOBs can be configured for many different input/output standards commonly found in digital electronics.

LE: A Logical Element (LE) is half of a slice. Each logical element contains one function generator, one memory element, a single-bit adder chain, and other reconfigurable logic.

LUT: A Look Up Table (LUT) is a common term for the function generator within a slice. Specifically, LUT is a function generator mode that can implement any 4-input function.

PLD: A Programmable Logic Device (PLD) is a device that can be programmed to perform a wide variety of logical operations and covers a broad variety of devices including FPGAs.

RAM: Random Access Memory (RAM) is used for fast, short-term storage because it is volatile.

ROM: Read Only Memory (ROM) is a type of memory that is static and cannot be written.

RTL: Register Transfer Level (RTL) is a model describing the gate-level design between two memory elements (registers).

RTR: RunTime Reconfiguration (RTR) is a technique where configurations are loaded onto the hardware as necessary instead of complete configuration during system startup.

SelectRAM: SelectRAM, distributed or block, provides both single and dual port configurations for random access memory (RAM).

SEU: A Single Event Upset (SEU) is a fault usually caused by radiation.

Slice: A slice is the internal element to the CLB. There are four slices within a Virtex-II CLB. Each slice contains two function generators, two memory elements, dedicated fast carry look-ahead adder chain logic, and other reconfigurable logic.

Tile: An element within the two-dimensional FPGA fabric. Two types of tiles are CLBs and IOBs.

XHWIF: The Xilinx HardWare InterFace (XHWIF) is the standard Xilinx interface for FPGA based hardware.

Chapter 1

Introduction

1.1 Overview

Modern Field-Programmable Gate Array (FPGA) tool suites attempt to manage the complexity of large designs by providing a single integrated design environment. Many tool suites include behavioral simulators, typically based on an architecture-independent model, allowing users to verify the functionality of a design. Examples of such tool suites are the Xilinx ISE tools [1] with optional ModelSim simulator [2] and Altium's Nexar2004 [3]. These flows assume that a functionally verified design will work in hardware as intended and expected. If the design does not function correctly in hardware, several possible problems may be the cause: the FPGA may be faulty; the implementation tools may have inferred, placed, and/or routed the logic differently than intended; or a flaw in the design was missed during testing. This thesis presents a new Java-based device simulator, VTsim, developed to resolve these problems by providing a window into the FPGA fabric via a virtual device.

VTsim provides Virtex-II FPGA designers with a new simulation paradigm: bitstream sim-

ulation. The only required input for VTsim to function is a valid bitstream, allowing the simulator to be independent of the design process. VTsim is an event-driven simulator that provides rapid response, is memory efficient, and supports multiple clock domains. At the time of writing, the simulator provides approximately 90% device coverage and models the majority of logic commonly used in FPGA designs. Through use of several optimization techniques, VTsim provides up to a 9,000% performance increase over prior work. A flexible, structured API has been developed to ensure ease of interaction and to allow for future incorporation of additional components.

VTsim is integrated into the JHDLBits [4] design suite available at SourceForge.net [5], allowing simulation in either the JHDL [6] to JBits [7] flow, or as a standalone simulation tool. By using VTsim, a designer can access and modify all resource values within the virtual FPGA at any time, view the state of flip-flops and lookup tables, and examine or change values on a routed wire. To perform these functions, VTsim relies on two additional tools: JBits and ADB [8]. JBits is an API that provides access to every configurable resource in a Xilinx FPGA and is used by VTsim for bitstream configuration information and bitstream bit-manipulation. ADB (Alternate wire DataBase) is a tool that supports routing and tracing services, provides complete device coverage, is memory efficient, and supports Virtex, Virtex-E, Virtex-II, and Virtex-II Pro FPGAs [9]. Information from ADB is used to configure the internal connections of the virtual device.

A device simulator is useful in reconfigurable designs where logic blocks are inserted and removed based on certain system states. In designs that implement partial reconfiguration, the placement of certain logic blocks is typically fixed, allowing the designer to reconfigure a small portion of the chip without affecting the overall design. The majority of simulators currently available do not have adequate support for reconfigurable designs. Because VTsim works at the bitstream level, both partial and complete reconfiguration may be simulated.

With the inclusion of FPGAs in mission-critical space applications, such as Xilinx FPGAs in the NASA/Jet Propulsion Laboratory (JPL) Mars exploration mission [10], the analysis and simulation of Single Event Upsets (SEUs) is an important emerging topic [11]. To simulate and analyze the results of a single event upset, a simulator must have complete knowledge and control of all FPGA configuration information. The majority of mainstream simulators do not rely on the low level configuration information used inside of a physical FPGA, and without the ability to alter the configuration information, it is virtually impossible to simulate SEUs in the FPGA configuration. This is the type of problem where VTsim excels. Because VTsim provides access to all configuration resources within the FPGA, easy simulation and analysis of single event upsets is possible.

1.2 Objectives

The primary goal of VTsim was to create a Java-based virtual FPGA that accurately models and simulates all resources within a Virtex-II FPGA. VTsim was designed with the following criteria in mind:

- Support for all Virtex-II devices
- 100% resource coverage
- Allow viewing and modification of individual FPGA resources
- Support multiple clock domains
- Provide rapid response
- Low memory usage
- Flexible and versatile interface

- Ease of interaction
- Structured, reusable design
- Initially without timing support, but can be extended appropriately

To support all Virtex-II FPGAs, the simulator had to be flexible and easily expandable to accommodate both current and new devices. A secondary goal was to design the simulator framework such that a large portion of the simulator could be retargeted for future FPGA architectures other than the Virtex-II family. It should also be noted that since the Virtex-II Pro tile structure is the same as the Virtex-II (except for the addition of an embedded PowerPC), the simulator could easily be extended to support Virtex-II Pro without PowerPC support. Because JBits3 does not have support for Virtex-II Pro devices, the simulator could not initially be developed for the Pro series. If future releases of JBits support Virtex-II Pro FPGAs, VTsim could be extended with relative ease to handle Virtex-II Pro devices.

At the time of writing, the simulator does not provide 100% resource coverage, but does model the majority of commonly used logic in designs. As will be discussed in detail throughout the thesis, VTsim has achieved all of the major goals with the exception of 100% device coverage.

1.3 Organization

This thesis is organized into seven chapters with sections that describe several key areas relating to VTsim. Chapter 1, Introduction, introduces VTsim by describing the general functionality, goals, advantages, and limitations. Chapter 2, Background, provides information on configurable devices, FPGAs, JHDL, JBits, JHDLBits, and prior work. Chapter 3, JHDLBits and JBits Enhancements, discusses the collective enhancements made to JBits and

JHDLBits. Chapter 4, Simulator Design, discusses the design of the simulator, including approaches, design organization, and implementation. Chapter 5, Simulator Usage, describes the steps necessary to utilize VTsim in several different design flows. Chapter 6, Results, explains the results of VTsim, compares and contrasts simulation with other simulators. Chapter 7, Conclusions, summarizes the thesis and results, and discusses future work.

Chapter 2

Background

2.1 Overview

Before VTsim is discussed, background discussion on FPGAs in general, Xilinx FPGAs in particular, configurable devices, and prior and current work relating to VTsim is presented.

2.2 General FPGA Information

FPGAs were first introduced in the mid-1980s to replace multi-chip glue logic circuits with a single reconfigurable solution [12]. FPGAs have far outgrown their sole use as a replacement for simple glue logic circuits [13]. Presently, FPGA applications include signal and image processing, graphic accelerators, military target correlation/recognition, cryptography, reconfigurable computing, and off-chip coprocessors. FPGAs are utilized in four major design areas: rapid prototyping, emulation, pre-production, and full-production [14]. Figure 2.1 illustrates the breakdown of the four major FPGA design areas by market share percentage

[14].

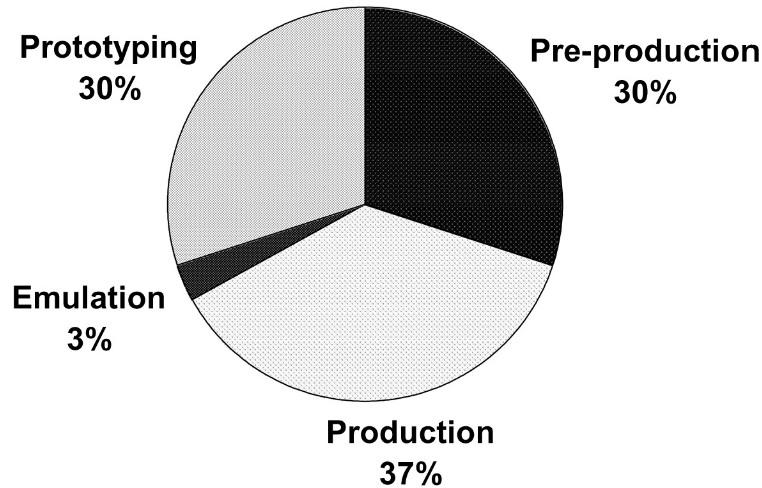

Figure 2.1: Four Major FPGA Design Areas

FPGAs are the direct result of the convergence of two distinct technologies: Programmable Logic Devices (PLDs) and Application Specific Integrated Circuits (ASICs) [15]. A simple PLD consists of arrays of AND and OR gates that can be used to create basic circuit designs. ASICs are custom-made chips generally used in high volume applications because non-recurring engineering costs (NREs) are much higher than in an FPGA design cycle. FPGAs are sized from thousands of gates to tens-of-million gates and are available in a variety of sizes with different packaging, internal logic blocks, and process technologies.

Internal FPGA architectures are commonly constructed using a symmetric tile structure containing a network of switchboxes, logic blocks, wire channels, and input-output blocks [16]. Figure 2.2 illustrates a tile matrix containing switchboxes (SB), wire channels, and logic blocks. A switchbox is a location in the FPGA fabric that provides a method to connect internal wires together. The switchbox allows horizontal wire segments to switch to vertical wire segments and vice versa. The switchbox also allows horizontal wire segments to connect

to other horizontal wire segments as well as connecting vertical wires to other vertical wires.

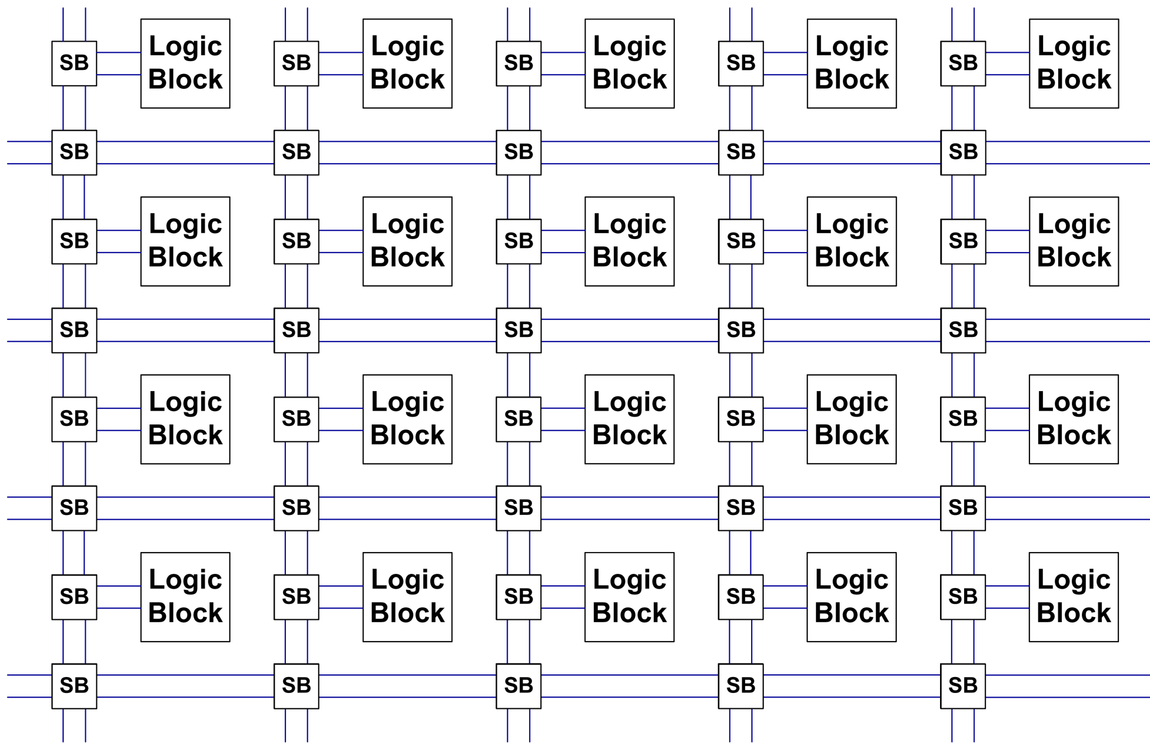

Figure 2.2: Internal FPGA fabric

The size and contents within a logic block vary greatly depending on the manufacture and target market. For example, FPGAs targeted towards cost-effective solutions typically contain simpler logic blocks than an FPGA targeted for high-performance applications. Although the contents within logic blocks can vary for different architectures, there are two basic building blocks found in a logic block: memory elements and function generators. Memory elements provide designers with the ability to temporarily store information until desired conditions are met. Function generators can be configured to produce any function up to the number of inputs into the function generator. Depending on the architecture, some function generators can operate in different modes such as random access memory (RAM), read only memory (ROM), or more complex modes like shift registers. FPGAs are configured

through a bitstream that is loaded into the device. A bitstream is a file created by the FPGA manufacturer that configures the switchboxes, logic blocks, and other internal FPGA logic.

FPGAs have redefined the boundaries of digital electronics allowing designers to build systems piecewise. Multiple designers can rapidly test and verify the functionality of each individual piece of a system to ensure proper functionality prior to merging the entire system together. With increasing interest in reconfigurable computing, FPGAs are recognized as the most viable, cost effective solution. Whether a design is statically or dynamically reconfigurable, FPGAs provide flexibility, rapid programmability, and a short time to market design cycle.

2.3 Xilinx FPGAs

Xilinx, a prominent leader in the FPGA market, was founded in 1984 and shipped its first commercial FPGA in 1985 [12]. Xilinx currently markets the Virtex-II family FPGAs geared toward high-density, high-performance designs [17]. Virtex-II FPGAs are, at the time of this writing, the most advanced FPGAs for programmable logic and offer the widest selection of density choices in the industry with eleven devices ranging from forty thousand to eight million system gates. [18].

The Virtex-II FPGA has dedicated 18-bit x 18-bit block multipliers, fast look-ahead carry logic adder chains, and up to 93,184 internal registers/latches [17]. Virtex-II devices are divided into a matrix of symmetric tiles as described in Section 2.2. The six major tile types in the Virtex-I are: Input/Output Blocks (IOBs), Configurable Logic Blocks (CLBs), hardware multipliers, 18Kbit block SelectRAM, and Digital Clock Modules (DCMs). Figure 2.3 taken from [17], shows the tile layout (TileMap) of the Virtex-II FPGA.

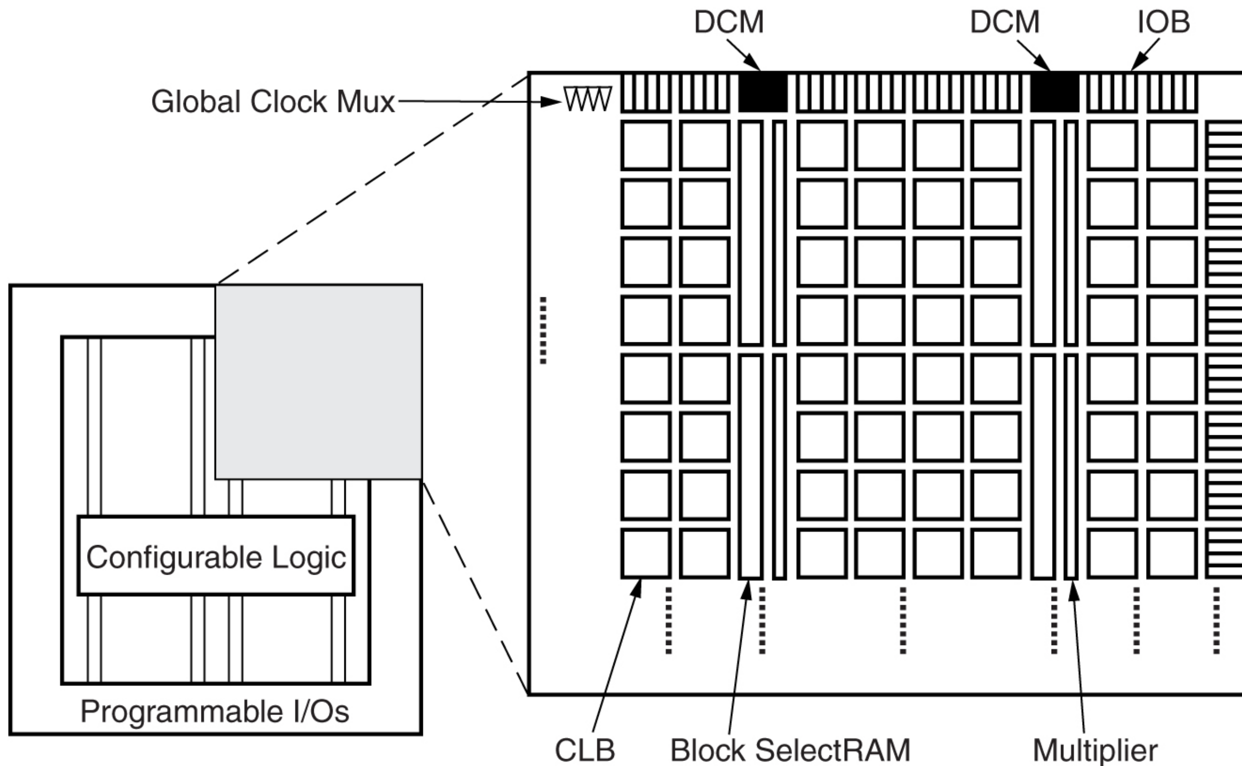

Figure 2.3: Virtex-II FPGA TileMap

2.3.1 Virtex-II CLB

The prominent configurable element in the Virtex-II FPGA is the CLB. CLBs occupy the majority of tiles in the device. Each CLB consists of four slices and two tri-state buffers. Each slice, which is divided into two similar logical elements (LE), contains the following components:

- Two function generators `(F and G)`
- Two memory elements (configurable for flip flop or latch mode)
- Shift logic

- Fast-carry look-ahead adder chain logic
- Horizontal cascade chain (`OR gate`)

Each LE contains one function generator, one memory element, and portions of the shift logic, carry chains, and OR chains. Figure 2.4, shows a functional overview of the LE layout.

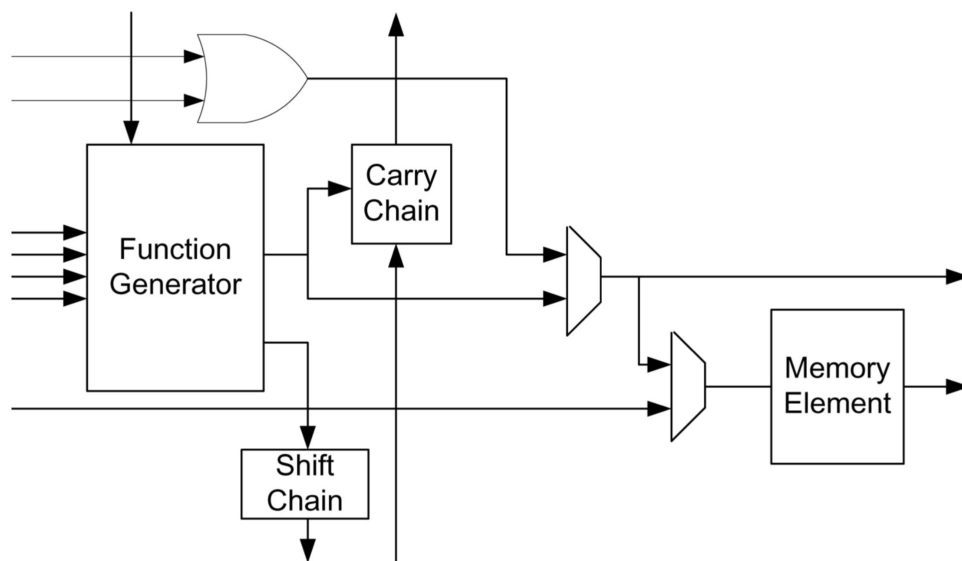

Figure 2.4: Functional LE layout

Each function generators can be configured four different ways: 4-input lookup table (LUT), 16-bit shift registers, 16-bit distributed SelectRAM, or as a 16-bit ROM (Read Only Memory). In the 4-input LUT mode, the function generator can implement any combinatory logic function up to 4-inputs. Multiple function generators can be cascaded or used in parallel to create functions of any arbitrary input size. The 16-bit shift register mode can be used independently of other function generators or cascaded together to form up to a 128-bit shift register in a single CLB. Multiple CLBs can also be cascaded together to form even longer shift registers.

There are two modes of operation for a function generator operating as SelectRAM: Single-port SelectRAM or dual-port SelectRAM. A single-port SelectRAM has only one address port, whereas dual-port SelectRAM has one port for synchronous writes and asynchronous reads. A second port is available for dedicated asynchronous reads. The dual-port configuration allows simultaneous reads and writes to the same SelectRAM. Each CLB can be configured in seven different SelectRAM configurations as shown in Table 2.1. The function generator ROM mode is very similar to the single-port SelectRAM mode. A single LUT can implement a 16x1 ROM or multiple LUTs can be cascaded together to form a ROM of arbitrary length.

Table 2.1: Sizes and Types of SelectRAM

Type of SelectRAM	**RAM Size**
Single-Port	`16 x 8 bit`
Single-Port	`32 x 4 bit`
Single-Port	`64 x 2 bit`
Single-Port	`128 x 1 bit`
Dual-Port	`16 x 4 bit`
Dual-Port	`32 x 2 bit`
Dual-Port	`64 x 1 bit`

The two memory elements within a slice can be configured as either an edge-triggered D-type flip-flop or level-sensitive latch. There are six different modes of operation for each memory element:

- Asynchronous set and reset (preset and clear)
- Asynchronous reset (clear)

- Asynchronous set (preset)
- Synchronous reset
- Synchronous set
- No set or reset

2.3.2 Virtex-II IOB

Input/Output Blocks (IOBs) are tiles in FPGAs that provide an access point to and from the internal FPGA fabric. IOBs are located around the perimeter of the FPGA fabric, see Figure 2.3. IOBs are commonly used for connecting external clocks, input/output data lines, and as test probes for debugging purposes. Each IOB within a Virtex-II FPGA has access to four external pads. Two pads can be used together to form a differential pair, or independently as either single-ended pads or digitally controlled impedance (DCI). The Virtex-II FPGA provides several different I/O standards: twenty-five single-ended I/O modes, eight differential signal modes, and twenty-six DCI modes.

Internally, an IOB contains six storage elements and several multiplexers to provide maximum input/output configurations. The memory elements within IOBs have the same functionality as memory elements in a CLB. There are three separate paths within the IOB to provide a path for input, output and the ability to put the output pin in tri-state mode. Also, combining two input or output flip-flops enable use of Dual Data Rate (DDR) registers. IOBs are divided into seven different banks, as shown in Figure 2.5; modified from [17]. The left image in Figure 2.5 depicts the top view for wire-bond packages; the right image the top view for flip-chip packages. There are several rules for combining different I/O standards within an IOB bank. Please refer to the Virtex-II datasheet available from Xilinx [17] for further information on operating modes, and IOB configurations.

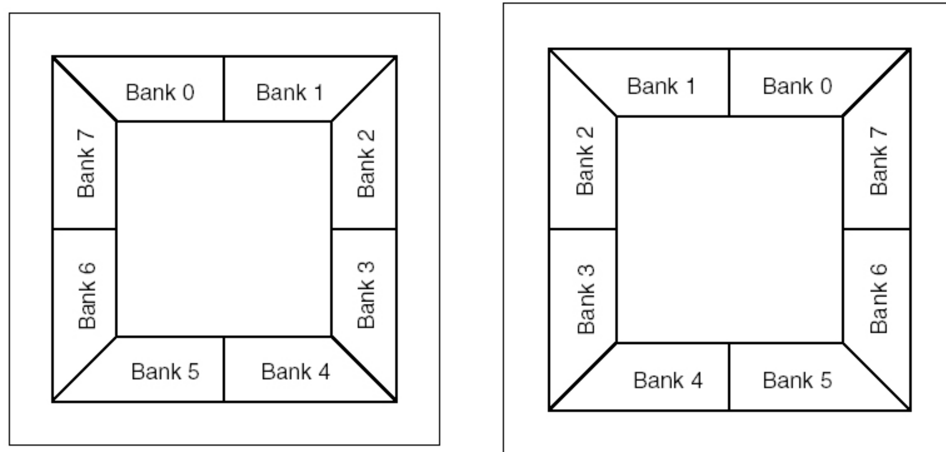

Figure 2.5: I/O banks for wire-bond (left) and flip-chip (right) packages

2.3.3 Virtex-II Clock Tiles

Virtex-II FPGAs contain two separate global clock buffer tiles: CLKT and CLKB. CLKT is located in the middle of the top row of the FPGA, see Figure 2.3, and contains eight global clock multiplexer buffers; the CLKB tile is located in the middle of the bottom row. The clock tiles are located in the middle of both the top and bottom rows to provide a low-skew clock distribution throughout the device. Only eight of the sixteen global clocks can be used in each of the four quadrants (top-left, top-right, lower-left, and lower-right) throughout the device. The global clocks can be used in conjunction with the DCMs or directly driven from the clock input pads.

Each global clock multiplexer buffer can be configured as a BUFG (global buffer), a BUFGCE (global buffer with clock enable), or as a BUFGMUX (clock selection multiplexer) as shown in Figure 2.6, modified from [17]. The simplest and most common configuration for a global clock is as a simple buffer (BUFG). A gated-clock can be implemented using the BUFGCE configuration. BUFGMUX mode allows switching between two unrelated asynchronous or

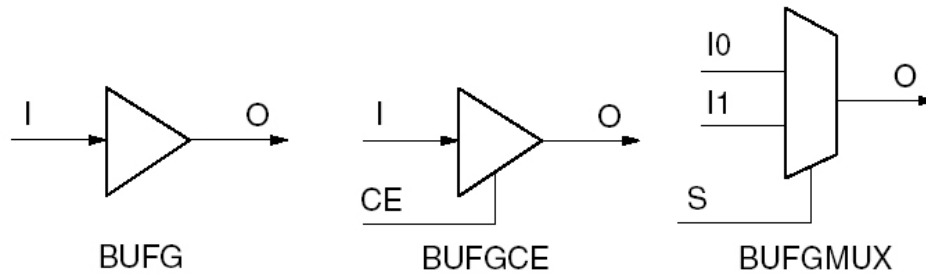

Figure 2.6: Global clock buffer configurations

synchronous clocks and ensures that the high or low time when switching clocks is never shorter than the shortest high or low time [17].

2.4 Related Work: JHDL, JBits, JHDLBits and ADB

This section introduces the design tools related to VTsim. Some of these tools are required to actually use VTsim, while others either enhance, or are enhanced by VTsim. As stated in the intro, VTsim is part of the JHDLBits open-source project geared to provide enhanced control of resource manipulation, placement, and routing. JHDLBits is a tool for converting high-level JHDL designs to bitstreams using JBits for bitstream interaction and ADB for routing. Figure 2.7 shows the JHDLBits constituents and the relationship to the JHDLBits project.

VTsim relies on JBits to handle all bitstream manipulation. A rudimentary understanding of JBits is necessary to understand how configuration information is processed and the bitstream manipulated. VTsim depends on ADB to provide routing and device information such as device tile layout and CLB locations. The routing information provided by ADB is used to create a netlist of all internal connections. The following three sections outline

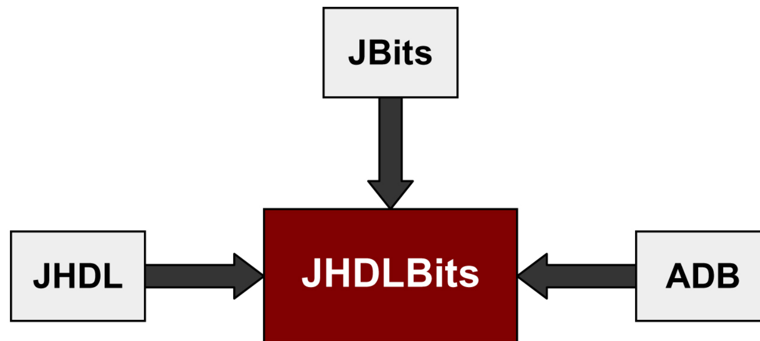

Figure 2.7: JHDL constituants

JHDL, JBits and ADB, and JHDLBits.

2.4.1 JHDL

Researchers at Brigham Young University have developed a Java-based structural Hardware Description Language (JHDL) FPGA design suite. Java was selected because it is easy to use, is object-oriented, has built-in documentation capabilities, is portable, and has a rich set of Graphical User Interface (GUI) APIs integral to the language [4]. JHDL consists of a single API that allows designers to create both static and dynamic circuit designs. [6].

The JHDL simulator has the capacity to function in either simulation or hardware mode. In simulation mode, all circuit values are calculated behaviorally and are device and architecture independent. In hardware mode, the simulator extracts the memory element values from an active FPGA, such as flip-flops, from the physical hardware and propagates the values through all of the non-memory elements, such as gates and adders. However, this does not provide a complete model of the hardware because the non-memory element values are still modeled behaviorally. Through the use of JHDLBits, VTsim has been integrated into JHDL to act as the simulator in place of actual hardware. This allows designers to emulate

hardware simulation by using the device simulator.

2.4.2 JBits and ADB Background

JBits is a Java-based API providing access to every configurable resource in the Virtex family FPGAs. Device resources may be programmed and probed at run-time, even if the FPGA is active in a working system. The JBits3 SDK [19] provides support for Virtex-II FPGAs, unlike the Virtex-based JBits2.8 release. JBits allows users to manipulate FPGA resources through two methods: `getTileBits` and `setTileBits`. The `getTileBits` is passed the tile coordinates and resource name and returns the associated configuration bits. The `setTileBits` essentially does the opposite. In the `setTileBits` method, JBits updates the configuration bits from the user defined tile coordinates, resource name, and new configuration bits.

JBits3 is a complete API for examining and modifying device configuration, but it does not include a device simulator or router. The previous JBits2.8 release included a device simulator, VirtexDS [20], and a router, JRoute - a run-time reconfigurable router [21]. The absence of a device simulator in JBits3 hinders design verification and the development of run-time reconfigurable systems. VirtexDS will be discussed further in Section 2.5.

Although JBits3 does not include a router, a router interface is provided allowing users to plug-in a separate router. One example of a plug-in router is the Alternate Wire Database (ADB) [8] supporting Virtex, Virtex-E, Virtex-II, and Virtex-II Pro FPGAs. ADB does not perform timing-driven routing; however, this simplification allows ADB to route nets quickly. Unlike JRoute in JBits2.8, ADB provides complete device coverage and is quite memory-efficient.

2.4.3 JHDLBits

JHDLBits is an open-source endeavor striving to merge the low-level control of JBits with the high-level design abstractions of JHDL. The JHDLBits project consists of a collection of tightly integrated components providing an end-to-end pathway for creating, manipulation, and debugging FPGA bitstreams [4]. Through use of ADB and JBits3, JHDLBits provides a quick path from design files to bitstream. Figure 2.8 from [22] illustrates the JHDLBits design flow.

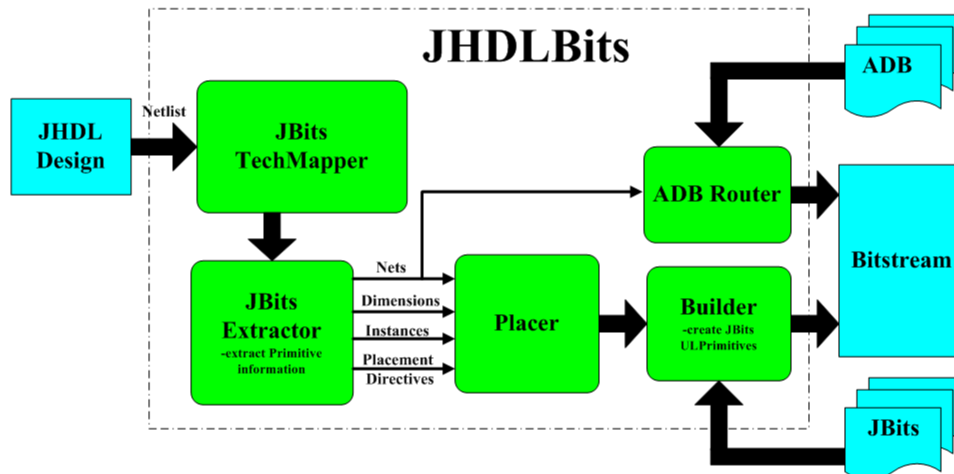

Figure 2.8: The JHDLBits Design Flow

The first step in the JHDLBits design flow is to create a working design in JHDL. The next step is to create a top-level test-bench file to act as an interface to JHDLBits. During execution of the test-bench file, the JHDLBits extracts all primitive and net information from the JHDL design and converts the nets and primitives into JBits `Nets` and primitives. Upon net and primitive conversion, JHDLBits creates the output bitstream and can either exit or instantiate VTsim for further design testing.

As discussed earlier, the JHDL simulator allows users to either behaviorally simulate their

design completely in software or to extract the memory elements from the hardware and behaviorally simulate the remaining elements. JHDLBits incorporates VTsim into the design flow by extending the JHDL simulator to add an additional simulation model. In this model, the JDHL simulator interacts with the device simulator instead of the physical hardware, allowing the user to achieve the same functionality of hardware without requiring the presence of actual FPGA hardware. Utilizing VTsim in the JHDL simulator has the same inherent problems as physical hardware: the non-memory elements are still calculated behaviorally. If simulation differs from the expected results, a designer is unable to probe values inside of the actual FPGA other than memory elements. This type of design problem is where VTsim excels. Future releases of JHDLBits will provide a new simulation model allowing designers to select VTsim as the simulator instead of the standard JHDL simulator. Choosing VTsim will provide a means to examine and manipulate all internal resources from within the standard JHDL framework.

2.5 Related Work: Simulators

Several types of simulators are currently available for FPGA design testing and verification. Most simulators require circuit designs to be modeled using VHDL (VHSIC Hardware Description Language), Verilog, or SystemC. Common types include: behavioral, functional, and RTL (Register Transfer Level) simulators. Behavioral simulators are the fastest type of simulators with speed improvements ranging from 10 to 100 times greater than RTL simulation [23]. Behavioral simulators are architecturally independent and therefore cannot take advantage of dedicated hardware circuitry like high-speed adders during simulation. Behavioral simulators provide a general picture of the overall functionality of a design. Behavioral simulators do not provide any timing related information. Functional simulators are typically architecture dependent and provide basic functional verification of the circuit design.

Because many functional simulators are architecture dependent, timing information can be included in the simulation model.

RTL models describe the transfer of data between registers. Data manipulation such as arithmetic operations (addition, subtraction, multiplication, division, etc.), logical operations (and, or, nand, not, etc.), and shift-type operations (right, left, circular, etc.) requires code written following specific guidelines [24]. For example, to properly design an RTL model in VHDL, a designer must use specific methods from the `STRUC_RTL.*` library. Utilizing these libraries allows synthesis tools, like Synplicity's Synplify Pro [25], to convert the RTL design into a gate-level circuit for simulation.

A new type of FPGA simulator that is design platform independent is the bitstream simulator. A bitstream simulator only requires an input bitstream for simulation. For a bitstream simulator to function correctly, a specific technique is required to extract the required information from the bitstream. The bitstream generation process is proprietary and highly-confidential, so vendors typically refrain for giving users access to the bitstream information. However, with the release of JBits from Xilinx, bitstream interaction is now available for Xilinx Virtex family FPGAs.

The first generation bitstream simulator, VirtexDS, provided support for the original Virtex family FPGAs. One advantage of a bitstream simulator is that the bitstream contains all of the information that is used inside the actual FPGA, such as routing information and logic design. A bitstream simulator functions precisely like a physical FPGA, and if designed properly, can incorporate timing information to provide high-level details of the circuit. Possibly the greatest advantage of a bitstream simulator is that it allows simulation of run-time reconfigurable (RTR) designs. Mainstream simulators are designed primarily for static designs that do not change over time. Bitstream simulators provide the necessary means to verify the functionality of reconfigurable systems. VTsim is a second generation

bitstream simulator for Xilinx Virtex-II FPGAs, developed to provide designers the ability to accurately simulate both static and dynamic reconfigurable systems.

While no direct comparisons can be made between VirtexDS and VTsim since the two operate on different architectures (VirtexDS supports Virtex FPGAs and VTsim supports Virtex-II FPGAs), there are some similarities between the device simulators. Both simulators are event-based, require only an input bitstream for simulation, support all devices within their designated family and are two-state simulators.

An event based simulator operates on changes that occur in the circuit. In both VirtexDS and VTsim, events are triggered when the value on a routed signal changes states. A two-state simulator is a simplified simulation model that only operates on two values: one and zero. Simplifying the simulation model to only two-states helps reduce the overall simulator design complexity and improves simulation execution times [26].

Two features not available in VirtexDS are present in VTsim: support for input stimuli files and the ability to analyze circuits containing multiple clocks. Unlike VirtexDS, VTsim currently does not have support for the Xilinx Hardware Interface (XHWIF) [27], and it does not simulate asynchronous circuits. While VirtexDS provides limited timing information, VTsim currently does not support timing information, although VTsim can be extended to support timing.

This chapter presented background information relating to VTsim, focusing on the tools VTsim is dependent upon and introduced some common simulator terminology. The next chapter discusses the JHDLBits and JBits enhancements that provide a pathway from JHDL to JBits, culminating in simulation.

Chapter 3

JHDLBits and JBits Enhancements

3.1 Overview

Although JBits3 is a fully functional API, it could be more user-friendly if five important components were not missing: a device simulator, a primitive library, a circuit interconnection structure, a placer, and a router. The exclusion of a device simulator greatly hinders design development and verification of FPGA designs that utilize JBits, especially Run-Time Reconfigurable (RTR) designs. The omission of a primitive library reduces circuit construction to such a low level that it is difficult to model any circuit containing more than a few gates. The lack of an interconnect structure denies designers a simple means to connect the circuit logic, and expand on modules already created. Without a placer and router, users cannot map their designs to the FPGA logic. Design verification and testing of JBits designs is difficult without a device simulator, primitive library, circuit interconnect structure, placer, and router.

While the main focus of this thesis is the description, development and implementation of a

Virtex-II device simulator (VTsim), a secondary goal is to create a JBits primitive library and interconnect structure to make JBits more complete and user friendly. All developments and extensions to JBits expressed in this thesis are part of the open-source JHDLBits project.

3.2 Enhancements

An early step in the JHDLBits project was to create a primitive library, which is a collection of basic building blocks commonly used by a circuit designer. Examples of primitives are NAND, NOR or any simple logic gate, flip-flops with an enable (FDE), flip-flops with a clear and an enable (FDCE), high-speed adders, and others. The construction of a primitive library was also necessary to allow development of JHDLBits. To simplify the JHDLBits conversion from JHDL to JBits, each JBits primitive was designed to match a corresponding JHDL primitive. When a JHDL primitive is found in the JHDLBits extraction process, JHDLBits maps the primitive directly to a JBits primitive, a simplification greatly improving conversion speed and overall memory usage.

Apart from the JHDLBits project, a primitive library was also necessary to aid JBits designers. In general, JBits users most commonly design at the primitive level or higher. The exclusion of a primitive library diminished interest in the latest JBits release. Without the inclusion of a primitive library, design complexity increases tremendously. VTsim played an integral role in the primitive verification process by providing quick feedback on the newly designed primitives without the risk of damaging expensive hardware.

For primitive development, connection, and testing, development of an interconnection infrastructure was required. The infrastructure added several features currently missing in JBits3. A bridging object, the `Bitstream` class, was created to allow abstract access to both JBits and router objects and facilitated the creation of primitives without dependencies on

architecture-specific classes [4]. A `Net` class was created to allow the connection of primitives by maintaining a list of the source and sink pins that form each net. The `Net` and `Bitstream` class information could then be passed to the placer and router.

After the JHDLBits primitive library and interconnect structure were developed, it was necessary to design a placer to interface with JBits. JHDLBits currently includes a simple placer, which evaluates the size of each primitive and assigns it to a specific location in the FPGA. In the simple placer model, each component is placed adjacent to the previous component. A more complex, intelligent, hierarchically-dependent placer is currently under development. Even though the placer is simplistic, it is suitable for designs not highly dependent on timing or routing resources. During the JHDLBits testing phase, a designs with very complex routing failed due to poor placement; however, these designs utilized nearly 100% of the FPGA resources.

JBits3 did not include a router, but the release included a router interface designed to allow users to create and plug-in a separate router. One router designed to work with JBits3 is the Alternate Wire Database (ADB). ADB supports Xilinx Virtex, Virtex-E, Virtex-II, and Virtex-II Pro FPGAs. Unlike JRoute, its predecessor, ADB provides complete device coverage and is more compact in size compared to other mainstream routers [8]. One limitation of ADB is that it does not create routes based on timing information; however, this simplification allows ADB to route very quickly. ADB is included in the JHDLBits open-source design suite.

To safeguard the physical FPGAs, VTsim was used extensively to verify the functionality of JHDLBits generated bitstreams. For example, a circuit modeled in JHDL was run through the JHDLBits extraction process to create a bitstream. The same JHDL design was also run through the mainstream tools to produce a second bitstream. Each bitstream was separately loaded into VTsim and carefully analyzed as shown in Figure 3.1. If the functionality of the

two bitstreams did not match, modifications to the JBits primitives were required. When the functionality of both matched, the JBits primitive was deemed ready to be tested on physical hardware. Thus, VTsim was developed in tandem with the JBits primitives and enhancements to provide verification of both the newly developed primitives and the device simulator.

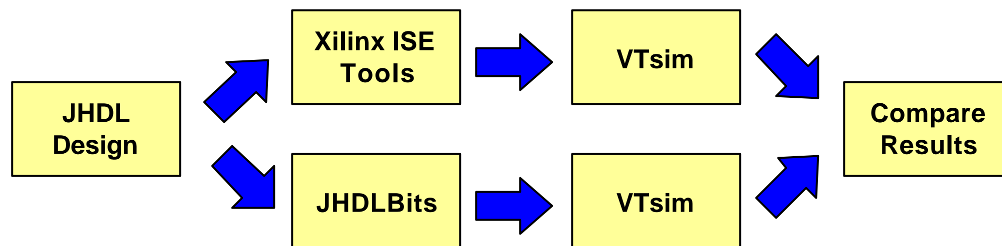

Figure 3.1: Bitstream Verification Process

With the design of these key components, sample JBits bitstreams could be generated using the enhancements found in the JHDLBits tree structure. However, without a bitstream simulator, it was not possible to verify the functionality of the entire process. The next logical step in the JHDLBits project was the creation of a device simulator.

This chapter conveyed the JBits enhancements added to the JHDLBits framework including a primitive library, an interconnect structure, and several bridging classes aimed at simplifying the JHDLBits design process. The next chapter presents a detailed examination of the internal simulator design.

Chapter 4

Simulator Design

4.1 Overview

To achieve the goals set forth in Section 1.2, careful planning and revisions were necessary. The first section in this chapter describes the various approaches that were tried before a final simulator model was chosen. A design outline is then examined, representing the process followed throughout the course of the project. Next, a detailed description of the step by step simulator design process is presented focusing on the design of the major components including a discussion on a variety of event-queue models that were evaluated.

4.2 Approaches

The initial design chosen for the device simulator was a non-event driven model. In the non-event driven model, the execution order of the simulator was defined and fixed at run-time by the creation of a data structure containing the execution order and circuit logic

information obtained from JBits3 and ADB. During each clock cycle, the simulator executed the entire data structure regardless of what values changed. Because the simulator executed the entire data structure for all clock cycles in the same manner, clock cycles that did not have any logic values change still required the same amount of time as a cycle where all logic values changed. Hence, the non-event driven model always executed assuming a worst case. One advantage of the non-event driven simulator is that event processing overhead was not required during each clock cycle because execution of each clock cycle is executed identically.

An important aspect of the simulator design was choosing the appropriate granularity level. Granularity level for the simulator is defined as the element that is simulated such as LE (half of a slice), slice, or tile. If the selected granularity level is too low, the simulator design is awkward and difficult. Conversely, if the selected granularity level is too high, many low-level resource details of the FPGA are missing. The goal was to identify a granularity level that maintained the fine-grain resource details of the FPGA while facilitating the development of a simplistic, highly memory efficient, fast simulator. For the non-event driven simulator model, LE granularity was the highest permitted without requiring additional checking for the possibility of infinite loops.

Raising the abstraction level to either slice or CLB granularity would introduce the possibility of an infinite execution loop. Figure 4.1 shows an example of such an infinite loop. The output of slice0, G LE is connected to the input of slice0, F LE. If slice granularity was used in conjunction with a non-event simulation model, the simulator would recognize that slice0, LE0 is connected to slice0, LE1 and slice0, LE1 is connected back to slice0, LE0. This pattern would cause the simulator to be caught in an infinite execution loop of slice0. Additional restrictions could be added to the execution model to check for infinite loops and specify a resolution function if an infinite execution loop was found. Resolution checking would add an additional layer of complexity to the simulator and reduce execution speed. Rather than investigating complex methods of resolving the loop issue, a decision was made

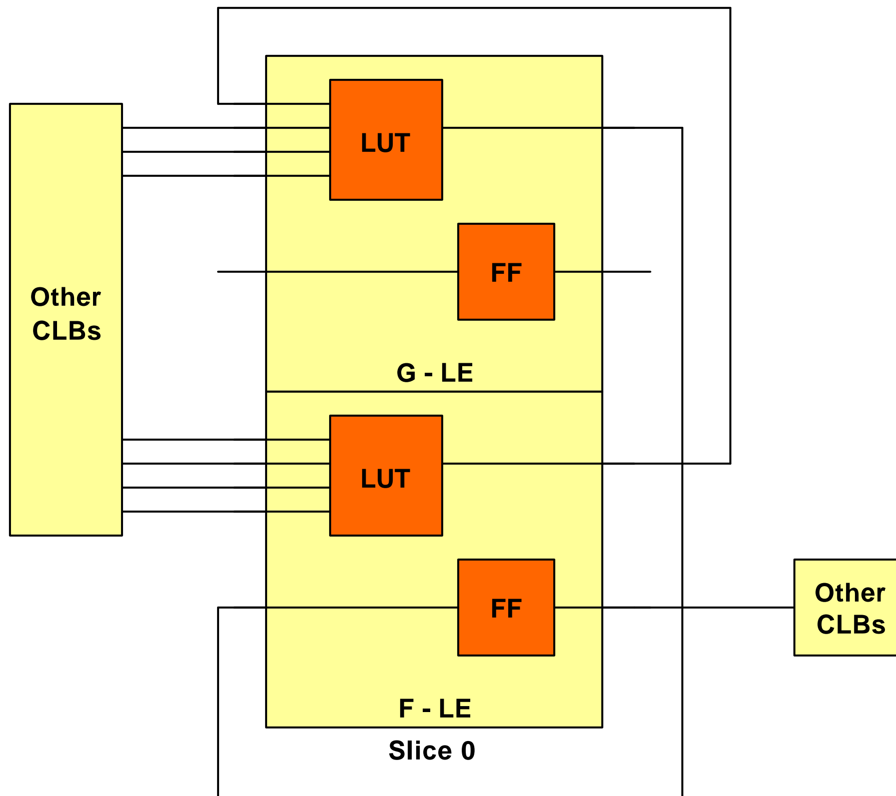

Figure 4.1: Sample Infinite Loop

to abandon the non-event driven simulator model and pursue other options.

Next, the event-driven simulator model discussed in Section 2.5, was examined. The event-driven model seemed the most logical approach for several reasons. First, the event-driven model was completely independent of the level of granularity, allowing the granularity to be selected based on factors such as memory use, execution time, and design complexity. The approach fostered the creation of a more versatile, well-tuned simulator. Unlike the non-event driven model, the order of execution is dynamic which causes variable clock cycle execution times. The event-driven simulation model was expected to have better average clock cycle execution times than the non-event driven model because it is uncommon for

every signal in a circuit to change during a single clock cycle. However, the worst-case execution time for the event-driven model would be much slower than the execution time for the non-event driven model.

In a typical circuit, every wire in the device would not be changing at the same time, so overall execution time for the event-driven simulation model will be much faster than the execution time for the non-event driven simulation model. An event-driven simulator adds a level of complexity to the simulation model because the simulator must keep track of all events and process them in the correct order. The additional layer of complexity increases overall execution time and memory usage. Because normal execution of the event-driven model will be faster and consume less memory, the event-driven model was selected as the basis for the device simulator design.

As discussed earlier, the event driven simulation model allows flexibility in the choice of the granularity level. Table 4.1 shows several simulator variants with different levels of granularity and a comparison of their relative complexity, execution speed, and overall memory usage. Please note that the stars in the table represent that the two table entries can reverse depending on the implemented design.

Table 4.1: Granularity Comparisons

Granularity	**Simulation Speed**	**Memory Usage**	**Design Complexity**
LE	Worst	Worst	Worst
Slice	Average	Best*	Average
CLB (Tile)	Best	Average*	Best

Simulation speed was approximated based on the total number of expected events processed versus the complexity of each event. LE granularity required the most events to be placed

on the queue. Keeping track of and processing these events increased the memory overhead when compared to the complexity of the event. Because LE granularity caused the most events, the size of the event queue was much larger than other granularities and required the most memory overhead.

Slice granularity had slightly better memory usage than CLB granularity primarily because many CLBs were not completely utilized. For the CLB model, the entire CLB must be placed on the queue for execution. Because a CLB contains four slices, the memory overhead associated with the creation and execution of the CLB is greater than the lower-level slice granularity if not all of the slices within a CLB are utilized. In a highly compact design utilizing all slices within a CLB, the memory overhead for CLB granularity would be lower than slice granularity. This shows that simulator memory overhead is directly related to how the implemented design is placed and routed. Because slice granularity was marginally better than CLB granularity in overall memory usage for certain designs, similar results were expected for simulation speed. However, because the internal connectivity infrastructure for the CLB used substantially less memory and provided faster execution times than the processing of four slices using the event queue, CLB granularity was the better performer in terms of overall simulation speed.

As discussed earlier, the Virtex-II FPGA is divided into a matrix of tiles. Therefore, it is natural to partition the virtual device using a similar approach. Selecting a CLB granularity aligns with the tile matrix design because a CLB is a specific tile type. CLB granularity would allow for the design of a uniform tile-structure rather than requiring components be designed at different levels of abstraction. Based on all of the factors discussed in this section, the decision was made to design VTsim as an event-driven device simulator modeled at the CLB granularity level.

4.3 Design Organization

The decision to use Java as the main programming language was made early in the design process because the two software packages VTsim interacted with, JBits and ADB, were both Java-based. Using a common programming language greatly simplified the simulator design. The first step in creating the simulator was to gain a solid understanding of the underlying FPGA structure and partitioning. To accomplish this, several weeks were spent reviewing topics ranging from proper Java coding techniques to data books and whitepapers on Xilinx Virtex-II FPGAs, to careful examination of FPGA Editor. FPGA Editor is a graphical application for displaying and configuring FPGAs [28]. FPGA Editor provides users with a tool to manipulate resources within the FPGA including routing, LUT equations, and individual resources (MUX, FF, carry-chain logic, etc.). Most of the naming conventions used in VTsim originate from names found in either FPGA Editor or ADB. FPGA Editor was an invaluable tool in understanding how the detailed FPGA structure.

Because VTsim is a second-generation bitstream simulator, VirtexDS was carefully studied to determine its strengths and weaknesses. The monitoring of several newsgroups and message boards allowed for a better understanding of designer preferences and what were considered important features for a bitstream simulator. With this knowledge, a structured programming outline and design schedule was developed. The following are design process steps for the construction of VTsim:

- Model a CLB
  - Create a working model for a slice
    - Model all logical elements found within a slice
    - Extract configuration information from the bitstream using JBits

  - Test and verify the slice model
  - Devise a means for slice interconnection
    * Consider a model based on the JHDLBits `Net` class
    * Create a simple design connecting two slices
  - Implement interconnection scheme at the CLB level
    * Consider revising interconnection scheme if not scalable to CLB level
- Create methods to extract information from ADB
  - Pass information acquired in ADB to all CLBs using interconnection scheme
- Model all CLBs within a device
  - Implement the interconnection scheme to connect an array of CLBs
- Create the event queue and clocking method
- Create a bitstream for a simple design using JHDLBits
  - Explore the simulator response and make necessary modifications
- After CLB verification, follow same design flow for other tile types
  - IOBs
  - Block SelectRAM
  - Hardware Multipliers
  - Continue until device is 100% modeled

After the initial literature review and creation of a general design flow, time was spent understanding how JBits and ADB operated in tandem. At this point in the JHDLBits

design, the required enhancements to JBits were present in the JHDLBits design tree. This provided JBits with the necessary components to generate bitstreams for simple designs. Then, ADB could be used to trace the internal routes and generate `TraceNodes` for the simple designs. `TraceNodes` are ADB data structures that contain a tree structure for all wire segments on a specific route [29]. Using the tracer in ADB, the JBits-generated bitstreams could be evaluated in terms of overall routing. Note, however, that without a simulator, the functionality of the bitstream could not be tested. Upon gaining a solid understanding of the interactions between ADB and JBits, the design of VTsim began.

4.4 Implementation

VTsim was designed using a bottom-up approach, with a high-level abstraction backbone to ensure a fine granularity while maintaining a high-level stable framework. Design work started at the lowest level, resources within a slice, and then the abstraction level slowly rose as more advanced features were added. Initial circuit development limited simulation to designs that only utilized CLBs, and did not include support for other tiles such as the hardware multipliers, block SelectRAM and IOBs. The initial goal was to create a simulator that could be useful in the development of JHDLBits. As support for more advanced applications was added to JHDLBits, these more advanced features were also incorporated into VTsim. Eventually, the majority of the FPGA components were covered by VTsim.

4.5 Tile Organization

As the design complexity level rose during each step, it quickly became apparent that a higher-level tile structure was necessary to maintain the locations of all the different tile types.

Therefore, a `Tile` class was developed from which all tile types would extend. Using this added level of design hierarchy, design of the entire virtual device was greatly reduced, leading to the creation of a two-dimensional array of tiles created during simulator initialization. Essentially, the two-dimensional array of tiles is the virtual FPGA.

By using information provided by ADB, the location of all tile types could be defined, and each tile type was then created to build the virtual FPGA. Initially only a limited number of tile types were supported. Undefined locations were not assigned a specific tile type and were instead assigned to the general `Tile` class. As more tiles types were developed, the location of each was extracted from ADB, and the configuration information for each type extracted from JBits. The framework was developed to allow quick and easy integration of new tiles as they were developed. Currently, four tile types have been implemented: CLBs, IOBs, CLKT, and CLKB. CLKT and CLKB are the top and bottom clock tiles that contain the sixteen global clocks; eight per tile. The four tile types represent between 80% and 95% logic coverage depending on device size, see Figure 2.3.

4.5.1 CLB and Slice Models

Following the design flow described in Section 4.3, the first step was to model a CLB. As discussed in Section 2.3, a CLB contains four slices and two tri-state buffers. Instead of behaviorally modeling the inner working of the CLB, great care was taken to replicate every resource within the slice; thus relying on the configuration data extracted from the bitstream using JBits.

For example, FPGA Editor shows a 6-input multiplexer (MUX) named CY0G in the G-half of the slice (the upper half of the slice containing the G function generator). The output of CY0G is determined by looking at the value of the three associated configuration bits. These

three configuration bits act as the select line inputs to the MUX. A variable is defined in the slice model that evaluates the three select lines and chooses the correct output based on these values – essentially modeling the component as a MUX in Java. Unlike CLB execution order, which is dynamic, the execution order within a slice is static and predefined. The entire slice was designed in this manner to allow designers to probe every resource within the virtual FPGA before or after any clock cycle.

4.5.2 Slice Design & Testing

All four slices within a CLB are identical and contain two function generators and other configurable logic. The two prevalent logic blocks inside a slice are function generators and memory elements. Although function generators can operate in many modes, initially only look-up-table (LUT) mode was implemented to reduce the design complexity. This simplification allowed for quick testing and verification. In the same manner, the memory element was initially only modeled as a D-type flip-flop with no set/reset or enable. Although this design methodology restricted the types of designs that could be implemented, simple circuits such as counters and registered combinatorial logic could be tested. These designs acted as a basis for slice verification.

After completion of the slice model, time was spent to verify its functionality. Because no framework to interact with JBits and ADB was in place yet, the design was hand-coded and entered into the slice model instead of simply reading in the configuration information from the bitstream, which also allowed VTsim to be tested without dependencies on either ADB or JBits. This meant that all errors found during simulation could be attributed to VTsim and not on the interactions between other tools. Once the slice model was verified, the next step was to use JBits to acquire the configuration information from the bitstream.

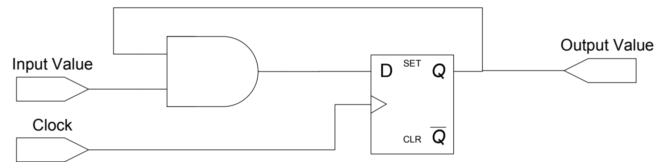

Figure 4.2: AND to Flip-Flop Circuit

Because only a single slice could be implemented at this time, and no external slice interconnect structure existed, only very simple circuits could be simulated. The initial test design, illustrated in Figure 4.2, was a two-input AND gate connected to the D-type flip flop. This circuit allowed verification of the LUT, flip-flop, and all other slice logic required to connect the two together. This test did not verify the entire slice because only a small portion of logic was required for correct operation of the design. The design was made with the aid of FPGA Editor to determine what logic needed to be connected to form the circuit. Figure 4.3 illustrates the design layout in FPGA Editor.

As shown in Figure 4.3, the SOPEXT MUX (multiplexer) was configured for the G input, and the DYMUX was configured for the DY input. During testing, it was noted that the output value from the function generator never propagated through the SOPEXT MUX. This error was caused by a bit-flip in the hand-coded configuration information; therefore some minor code tweaking was necessary. After the test completed successfully, other registered combinatorial logic was tested to ensure no other unexpected problems arose. After all of the bugs were worked out of the simple slice model, the next step was to create a means of interconnection between slices.

Figure 4.3: FPGA Editor: AND to Flip-Flop Circuit

4.5.3 Slice Interconnect Scheme

Once a working model of a slice was created, the next step was to design a means to connect all four slices together inside a CLB. Since the interconnect scheme was to be used multiple times for every CLB in the virtual device, it was important that the process require minimal time and memory. The initial idea was to create a `Net` class that would be transferred back and forth between the slices. The theory was that each net would retain its value and could be assigned to any input or output pin in the slice; however, this method was better suited for more dynamic routing that varied either over time or by designs. Because the routing between all four slices inside of a CLB is more or less static and predefined, a different approach was desired.

After studying FPGA Editor, examining CLB wires using ADB, and reading the Virtex-II

datasheet, it was concluded that approximately half of all connections to or from a slice came to or from another slice inside the same CLB, or were left unconnected. In Figure 4.4 the bold lines represent wires connected to either the CLB switch box or surrounding CLBs. Figure 4.4 shows that half of the wires inside a CLB are internal. Therefore, it was decided that the values be passed as parameters between slices instead of creating a separate class. Passing the values as parameters did not create any additional memory overhead and required no additional modifications to the slice class.

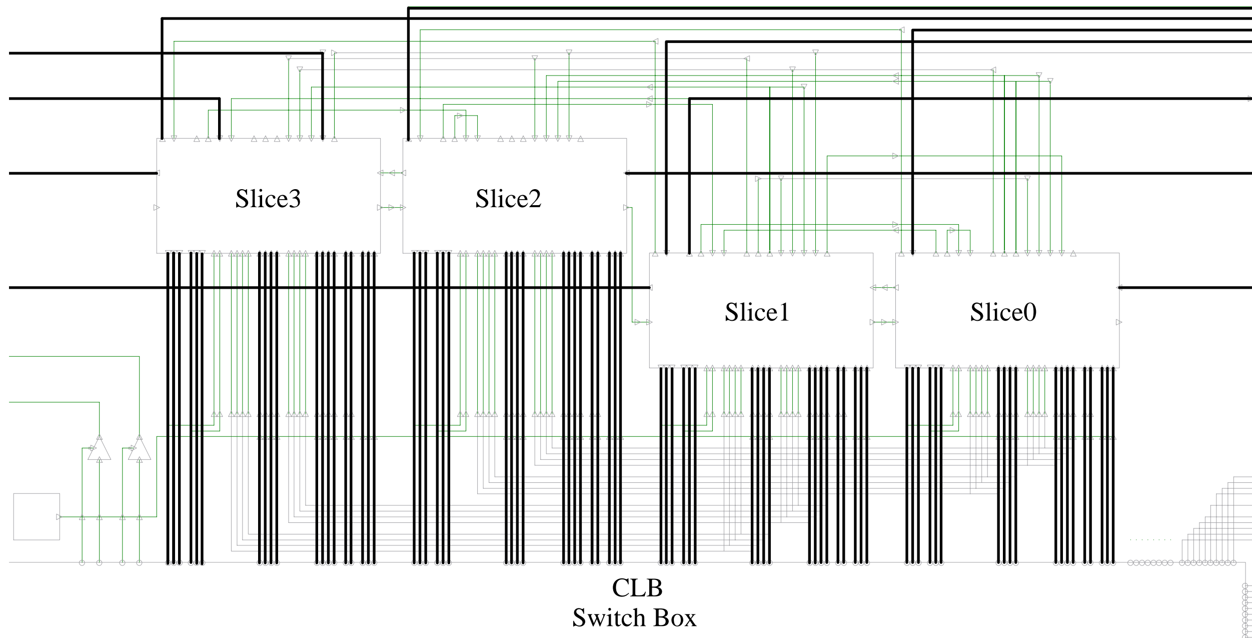

Figure 4.4: Internal CLB connections

4.5.4 Utilizing JBits for CLB Configuration

After determining that no additional classes needed to be created for slice interconnects, the next step was to remove the hand-coded dependencies from the slice model by using JBits to obtain the configuration information from the bitstream. Following the examples found

in the JBits documentation on reading configuration information [19], a simple loop method was derived to extract the configuration information for all four slices from the bitstream and to configure the virtual FPGA. Figure 4.5 illustrates some of the required function calls in the loop to get all the required configuration information.

```
458:   // ==================== LUT config information ====================
459:   // Get the configuration information for resource the lut mode
460:   CLBconfigInfo[i][CLBslice.LUTMODE] = Util.IntArrayToInt(Bitstream.
461:     getTileBits(jbitsTileRow, jbitsTileCol, LUT.MODE[i]));
462:
463:   // Get the configuration information for resource flutconfig
464:   CLBconfigInfo[i][CLBslice.FLUTCONFIG] = Util.IntArrayToInt(Bitstream.
465:     getTileBits(jbitsTileRow, jbitsTileCol, LUT.CONFIG[i][LUT.F]));
466:
467:   // Get the configuration information for resource glutconfig
468:   CLBconfigInfo[i][CLBslice.GLUTCONFIG] = Util.IntArrayToInt(Bitstream.
469:     getTileBits(jbitsTileRow, jbitsTileCol, LUT.CONFIG[i][LUT.G]));
470:
471:   // Get the configuration information for resource flutcontents
472:   CLBconfigInfo[i][CLBslice.FLUTCONTENTS] = Util.IntArrayToInt(Util.
473:     InvertIntArray(Bitstream.getTileBits(jbitsTileRow, jbitsTileCol,
474:     LUT.CONTENTS[i][LUT.F])));
475:
476:   // Get the configuration information for resource glutcontents
477:   CLBconfigInfo[i][CLBslice.GLUTCONTENTS] = Util.IntArrayToInt(Util.
478:     InvertIntArray(Bitstream.getTileBits(jbitsTileRow, jbitsTileCol,
479:     LUT.CONTENTS[i][LUT.G])));
```

Figure 4.5: CLB configuration code snippet

Because only parts of the CLB were implemented, only portions of the bitstream configuration information and loop method could be tested. The next step was to create a bitstream that contained a 2-input AND gate connected to a D-type flip flop. This is the same design that was hand-coded during the initial slice design phase. To simplify the test, the functional CLB had to be placed in a specific location so it could be easily extracted and the design verified. To do this, many of the JBits enhancements found in JHDLBits were utilized.

Because this was the first time the primitives defined in JHDLBits were tested, there were many different places where errors could be introduced. A design was constructed using

JHDLBits that contained two elements: an AND2 primitive and an FDE primitive. An FDE primitive is a D-type flip flop with an enable line. The enable on the flip-flop was not yet implemented in the simulator, so toggling the enable line would have no affect on the simulator flip flop model. Therefore, the enable line on the FDE primitive was connected to logic one for simulation purposes.

After fixing a few errors found in the JHDLBits primitives and associated classes, a working bitstream was generated. Using the loop method described above, all the configuration information was extracted from the bitstream and passed to the slice. It was expected to find a few errors in the verification process since so many different parts were being merged together. The process worked successfully the first try and subsequent tests using different logic also worked flawlessly.

4.5.5 CLB Connectivity

While researching different slice interconnect schemes, two observations were made:

1. If connections between elements were static, a simple connection scheme should be used
2. If connections were dynamic - a more elaborate scheme needed to be developed

Because connections between CLBs are dynamic, and in reconfigurable applications the connections can change at any time, it was necessary to develop a flexible connectivity interface. Initially, the thought was to create an expandable array within each CLB for each output CLB pin. The array would keep track of all sink CLBs and the associated pins. If an output value changed, the CLB would pass the value to every other CLB in the expandable array and propagate the value through the CLBs. The major downside of this process is that

all CLBs would need to have knowledge of the device and how everything was connected, going against the principle that all functionality within a CLB was independent of the overall structure and activity outside the CLB. Because a CLB in a physical device does not have all the connection information, it was decided that this method was not a good approach. Instead, ports would be defined in the CLB that would act as pins. The ports would simply reference an external net to get the current value at execution time. Using ports references reduced the number of method calls during execution. For example, an output port on a CLB drives ten sink ports on different CLBs. If a value changed on the output port, the original model would need to propagate the value through all the CLBs immediately when the value changed. The propagation approach required additional memory overhead and increased simulation time because all CLBs maintained a complete list of all sink and source ports. The new model retains the value in a separate data structure and CLBs can access the value as needed, reducing simulation time and memory overhead. Simply stated, all simulation nets (`SimNets`) retain the value and a list of the source and sink CLBs. Inside the CLB model, `SimNet` references were created for each output port and during simulation, the CLB would query the `SimNet` for its value.

The next step was to determine how to create, define, and assign connections from the bitstream information. This process required the use of ADB. During simulation initialization, the routing information would be extracted from ADB in the form of a `TraceNode`, and for each route a `SimNet` was created. As discussed in Section 4.3, an ADB `TraceNode` contains all wire information for a single route. From the `TraceNode` the source pin is easily found: It is the topmost entry in the `TraceNode`. By recursively traversing the `TraceNode` tree looking for endpoints, all sink pins could be found. Once all required information was extracted from the `TraceNode`, a `SimNet` was passed the information for configuration. Using the `SimNet` structure, CLB independence was maintained, and a low-memory, highly efficient algorithm was developed.

4.5.6 IOB Tiles

As discussed in Section 2.3.2, IOBs provide access points into the FPGA fabric for clock, input, and output signals. Each IOB consists of four cells containing pad logic, referred to as IOB slices. An IOB slice consists of six memory elements, and several control muxes. Designing the IOB slice model was relatively simple because code used in the CLB slice design section could be reused to implement the memory element functionality. The IOB tile uses the same interconnect scheme as the CLB. Input and outputs from IOBs are assigned a `SimNet` that controls the updating of connected nets.

Because the simulator is a two-state simulator within the virtual FPGA, use of differential-pair inputs/outputs and other IOB configurations are not supported; only logic one and logic zero inputs are valid for VTsim. When IOBs are configured for output mode, VTsim supports tri-state logic to the output pad; the output is seen as zero, one, or two, with logic two signifying the output is in tri-state mode. DDR mode is also currently not supported because the DDR clocks must be generated by the DCM at 180 degrees out of phase. Future VTsim releases will include DCM support, which will allow support for DDR mode.

4.5.7 CLKT and CLKB Tiles

The CLKT and CLKB tiles each contain eight global clock buffers used to drive the clock lines throughout the FPGA. The only logic inside the CLKT and CLKB tiles is the actual clock buffers, which can operate in three different modes as described in Section 2.3.3. As with IOB and CLB tiles, the input and output of the clock tiles are connected to `SimNets`. During simulation, the output `SimNets` of the clock buffers is used to trigger the clocked event queue, which is the beginning of an execution cycle. A more detailed description of the queue interaction with the FPGA will be presented in the next section.

4.6 Event Queue Models

The event queue is the heart of the simulator. It is responsible for updating all activity throughout the virtual FPGA. The queue executes all clocked and non-clocked logic, and is vital to the implementation of a flexible and highly efficient simulator model. One fundamental goal for the event queue was to ensure adequate scaling, implying execution time should be completely independent of the device used during simulation. For example, a ten-bit counter design implemented on an XC2V40 device should have the same execution time for an identical design on an XC2V8000 device. After running several tests, it was determined that one-third of the total simulation execution time was spent evaluating and adding events; one-third updating non-clocked events; and one-third updating clocked events. The non-clocked and clocked update methods reside in each specific tile type, and have been optimized prior to testing various event queue models; therefore it was essential to minimize time spent in evaluating and adding events.

The initial design for the event queue was to use a `GrowableIntArray` class from ADB, better known as a stack. A `GrowableIntArray` is a "class to hold a growable array of ints, typically employed as a reusable stack" [29]. When a value on a `SimNet` changed, the `SimNet` called an `addIfNotPresent` method for each sink tile location. The `addIfNotPresent` method would search the stack to determine if the tile was already an entry on the stack. To do this, the method checked every entry on the stack and compared it to the sink tile. If the method completed without finding the entry on the stack, the sink tile would be added to the stack. The major disadvantage of this approach is that the queue could become very large, and because the `addIfNotPresent` is the most frequently called method in the simulator, the stack would be searched several millions of times for long simulations. Because this design was fairly simple, it was the first design implemented and performed reasonably well compared to other design attempts.

To improve upon the initial event queue model, several event queue variants were tested. The first attempt at improving the queue execution speed was to use a fixed-size Boolean array. Each location in the array represents a specific tile coordinate, and the size of the array is determined by the size of the device. A TRUE entry in the array indicated the tile needed to be updated, while a FALSE entry indicated the tile did not need updating. Execution of the queue began at the first entry in the array and continued at the beginning again once execution reached the end of the array. The queue executed in the circular fashion until no TRUE entries were found in the array during one complete cycle. The major downfall of this model is that the size of the array is completely dependent on the size of the device. Small devices perform very well using this model; however, designs using the large devices are nearly twice as slow as the `GrowableIntArray` model. Because of the inability for the Boolean array to scale between small and large devices, another approach was preferred.

Several other models were implemented including a model combining the approach used in the Boolean array with a `GrowableIntArray`. This model provided better scaling between devices, but sacrificed small design performance to achieve only marginally improved large design performance. Another approach was to use a `HashMap` in conjunction with a `GrowableIntArray`. The `HashMap` would store the indices to the `GrowableIntArray` allowing for a better checking method instead of sequentially checking the entire stack. This model was never fully optimized because it quickly became apparent that the use of a single `HashMap` could be used instead of merely complementing the `GrowableIntArray`.

Use of a single `HashMap` would reduce the complexity of the `addIfNotPresent` method, resulting in shorter execution time. The basic principal behind the `HashMap` approach is that a key to the `HashMap` would be the tile location. Therefore, when the `addIfNotPresent` method was called, the queue would not need to check if the value was already entered into the stack, instead the put method could be called. If an entry already existed with the identical key, the entry would be overwritten with the new key and value, which is identical

to the previously stored value. Removing the necessity to search the entire stack resulted in an astounding thirty-five percent speed improvement for large designs. The downside to the `HashMap` model was that for small designs performance was cut by approximately ten percent. However, a thirty-five percent speed improvement for larger designs was deemed more crucial because small designs were still capable of executing approximately seventeen thousand clock cycles per second. Further analysis of execution speed and overall performance will be covered in the results section. Because of the drastic speed improvements achieved using a single `HashMap`, the decision was made to complete optimizations of the event queue using the `HashMap` model.

This chapter discussed the fine-grained simulator design details including the evaluation of different simulation models, explanation of individual tile designs, and enhancements made to the event queue. The next chapter explains different usage models for VTsim and provides example code to help clarify specific points.

Chapter 5

Simulator Usage

5.1 Overview

An important part of the simulator design process was to ensure VTsim could be utilized in the design flows described throughout the thesis. As discussed earlier, the simulator is part of the JHDLBits suite aimed at generating a bitstream from JHDL code. Therefore, it was important to provide several methods of integrating the simulator into this design flow. JBits primitives and a connectivity structure were created for the JHDLBits flow, but were designed so that they could be used in a JBits-only design. Therefore, VTsim needed to provide a way for JBits-only designs to interact with the simulator. A third simulator usage goal was to be able to simulate bitstreams not associated with either JBits or JHDL. A description of the key classes and methods is presented in conjunction with examples of how to use VTsim in each design flow.

5.2 VTsim Key Classes and Methods

Regardless of the chosen design flow, initialization and access to VTsim is performed similarly. When VTsim is coupled with the JHDLBits design flow, JHDLBits invokes and initializes VTsim, making the inclusion of VTsim quite transparent. In the JBits and other design flows, the user is responsible for creating the appropriate VTsim object and assigning the input bitstream and optional .net file. The .net file is generated during the final bitstream creation process in both the JHDLBits and JBits-only design flows and includes source and sink pin information used to map the original design to the bitstream file. The .net file information provides a method for users to easily simulate their designs by accessing or changing values on `SimNets` using the `SimNet` name.

VTsim is a browser class allowing users to interact with the simulator from a command prompt, which is very useful for debugging purposes. The VTsim browser file, which will be further explained in Section 5.5, demonstrates examples of how to create and interact with the virtual device simulator class: `VTDS`. The `VTDS` class can be thought of as the window into the virtual FPGA fabric. The `VTDS` class provides means to access and modify all simulation nets and resources including configuration information. There are four different `VTDS` constructors available to the user depending on the desired functionality and input choices.

VTDS(): Constructs the `VTDS` (Virtex-II device simulator) object from already defined bitstream and router information. This constructor was designed for use with the JHDLBits extraction process. It assumes a predefined output bit file and that ADB has the desired bitstream resident in memory.

VTDS(String bitstream): Constructs the `VTDS` object from a user-defined input bitstream.

VTDS(String bitstream, String netFile): Constructs the `VTDS` object from a user-defined input bitstream and .net file.

VTDS(String bitstream, Boolean infer): Constructs the `VTDS` object from a user-defined bitstream and infers the .net file from the bitstream name (If infer is true). If the infer value is true, the two names are assumed to be the same, but with different extensions (i.e. myDesign.bit and myDesign.net). If the infer value is false, the constructor is the same as `VTDS`(String bitstream).

All `VTDS` constructors follow the six initialization steps shown in Figure 5.1, with the exception that some constructors omit tracing and .net file analysis. The first step in the initialization process is the creation of the ADB and JBits object. During the ADB/JBits creation phase, device information extracted from ADB and JBits is used to configure VTsim. The next two steps, tracing and .net file analysis, are optional depending on the mode of operation. Tracing accounts for approximately one-half to two-thirds of the total simulator initialization process. The next step is to create and configure the virtual device using information acquired from ADB and JBits. ADB is used to extract the total size of the device and specific tile locations for the bitstream. JBits is used to retrieve the bitstream configuration information such as function generator values and memory element modes.

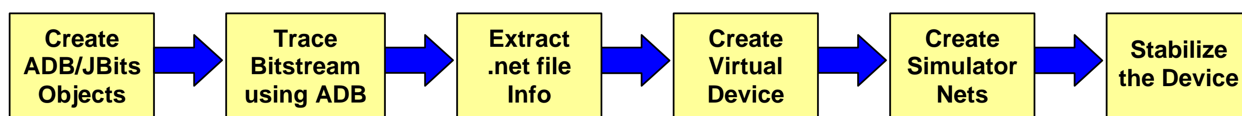

Figure 5.1: The six VTDS initialization steps

The next step is to configure the virtual device `SimNet` connections, by traversing ADB `TraceNodes` for all output pins throughout the device. The `SimNet` creation process is design dependent and can consume as much as one-quarter of the total initialization time

for large, complex designs. The final initialization step is to stabilize the device. This is done by executing the update method for all tiles throughout the device to ensure the internal tile values have been properly initialized to a stable state.

After `VTDS` is configured and initialized, the user has several options. Common simulator interactions include stepping one or more clocks and examining or altering of `SimNet` and resource values. These actions can be performed by using several methods found in the `VTDS` class. Note that all methods throughout the VTsim API are available to the user, however the following are the six most commonly used methods for design verification.

clockUpdate(int numCycles): Clocks the virtual device the number of user specified clock cycles for ALL sixteen global clocks.

clockUpdate(int numCycles, String clkName): Clocks the virtual device the number of user specified clock cycles for the `SimNet` associated with the name clock name.

getSimNetValue(String name): Retrieves the current value on the `SimNet` associated with the name.

getResourceValue(int tileRow, int tileCol, int slice, String name): Retrieves the current value for the resource associated with the name and location information. The name is typically a variable field found in either the `CLBslice` or `IOBslice` classes.

setSimNetValue(String name, int value): Changes the current value on the `SimNet` associated with the name to the user-defined value.

setResourceValue(int tileRow, int tileCol, int slice, String name, int value): Changes the current value for the resource associated with the name and location information to the user-defined value. The name is typically a variable field found in either the `CLBslice` or `IOBslice` classes.

5.3 JHDLBits Design Flow

There are two different ways in which VTsim can be integrated into the JHDLBits design flow. One way is to design a standalone test bench to test the bitstream. A second way is to integrate VTsim into the JHDL simulator through the JHDL hardware interface. The JHDL hardware interface extracts memory element values from the bitstream during after each clock cycle. This mode is completely transparent to the designer because during the JHDLBits extraction process, JHDLBits overrides the required methods to extend the JHDL hardware interface to interact with VTsim. In this mode, the designer can either use the standard JHDL simulator and waveform viewer to evaluate their circuit or a standard JHDL test bench can be designed. Because the JDHL simulator only extracts memory element values from the hardware, the remainder of the circuit is behaviorally modeled. Future work on the simulator is expected to completely override all JHDL simulator functions allowing for true bitstream simulation from within the JHDL framework. However, until these additional features have been implemented, if a designer wants to completely model the circuit using VTsim, a separate VTsim test bench file must be created.

To remove dependencies on the JHDL simulator in the JHDLBits design flow, a standalone test bench utilizing VTsim's API is necessary; however, current limitations in the JHDLBits extraction process prohibit fully verifying this approach. When the necessary changes are made to the JHDLBits process, this approach can then be verified. The same approach described here is used in the JBits-only design flow and because the limitations lie in the actual JHDLBits process and not in the interaction with VTsim, it is expected that once changes are made to JHDLBits, the process will be very similar to the JBits-only flow.

5.4 JBits Design Flow

JBits designs can be simulated using two different techniques. Through use of the VTsim browser, any bitstream can be simulated and evaluated interactively from the command prompt. A thorough description of the VTsim browser is explained in Section 5.5. The second technique for simulation of JBits created bitstreams is to write a test-bench file. To utilize VTsim in the JBits design flow, the JHDLBits `Net` class must be used to produce a .net file; otherwise default `SimNet` names are used making it a little more difficult for `VTDS` to access values using the `SimNet` name field.

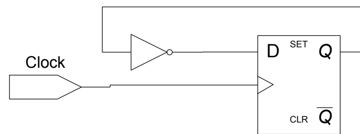

Figure 5.2: Simple oscillator circuit

A test-bench file can either be incorporated into the JBits design or written as a secondary file. Because the test-bench is a standard Java file, the user has access to all of Java's features. Incorporating VTsim into an existing JBits design file will reduce the simulator initialization time by a factor of two because the ADB routing object is already resident in memory. An example of an inclusive test-bench file is the HelloVTsim example in Figure 5.3. The HelloVTsim example illustrates the proper techniques to create the `VTDS` object and access the simulation information using several `VTDS` methods described in Section 5.2. The design implemented in the HelloVTsim example is a simple oscillator circuit (flip-flop to inverter to flip-flop), as shown in Figure 5.2.

The HelloVTsim file starts by creating the `Bitstream` object from a null bitstream. The next step in the JBits design is to create the JHDLBits `Nets` that will be associated with

```
 1: // Import the simulator package
 2: import JHDLBits.Virtex2.Simulator.*;
 3: // Import the proper JHDLBits packages
 4: import JHDLBits.Virtex2.ULPrimtives.*;
 5: import edu.vt.JBits.ArchIndependent.Net;
 6: import edu.vt.JBits.Virtex2.Bitstream;
 7:
 8: public class HelloVTsim {
 9:   public static void main(String args[]) {
10:     // Create the Bitstream class from a null bitstream
11:     Bitstream bits = new Bitstream("null2v40.bit");
12:     // Create the SimNets to be used throughout the circuit
13:     Net clk = new Net("clk");
14:     Net vccNet = new Net("vccNet");
15:     Net invOut = new Net("invOut");
16:     Net FFout = new Net("FFout");
17:
18:     // Connect the clk SimNet to global clock 0
19:     clock CLOCK = new clock(clk, clock.GCLK0);
20:     // Create the VCC, inverter, and FF primtives with placement
21:     vcc VCC = new vcc(vccNet,0,0,0,0);
22:     inv INV = new inv(FFout,invOut,0,0,0,1);
23:     fdc FF = new fdc(invOut, vccNet, FFout,0,0,1,0);
24:
25:     // Route the Nets; write the nets; then write the bitstream
26:     Net.route();
27:     NetWriter netwriter = new NetWriter("HelloVTsim.net");
28:     netwriter.write();
29:     bits.writeFull("HelloVTsim.bit");
30:
31:     // Create the device simulator object and get the initial values
32:     VTDS vtds = new VTDS();
33:     int FFoutValue = vtds.getSimNetValue("FFout");
34:     int invOutValue = vtds.getSimNetValue("invOut");
35:     // Check if the initial values are correct: FFout=0 and invOut=1
36:     if((FFoutValue == 0) & (invOut == 1)) {
37:       System.out.println("Initial FFout and invOut values are correct.");}
38:     else{System.out.println("Simulation failed on first clock cycle.");}
39:     // Clock the device one time
40:     vtds.clock(1, "clk");
41:     // Check to see if the values are correct: FFout = 1 and invOut = 0
42:     if((FFoutValue == 1) & (invOut == 0)) {
43:       System.out.println("Values after one clock cycle are correct.");}
44:     // Clock the device 101 times; values should equal initial values
45:     vtds.clock(101, "clk");
46:     // Check to see if the values are correct: FFout = 0 and invOut = 1
47:     if((FFoutValue == 0) & (invOut == 1)) {
48:       System.out.println("Simulation completed successfully!");}
49:     else{System.out.println("Simulation failed on 101 clocks.");}
50: } }
```

Figure 5.3: HelloVTsim example code

the clock, vcc, inverter, and flip-flop primitives. The names assigned in the `Net` constructors are the same used when accessing `SimNet` values during simulation. After `Net` creation, the clock, vcc, inv, and fdc JBits primitives are instantiated. The last four entries in the constructor for all primitives are the tile row, tile column, slice, and LE. At this point in the file, the oscillator circuit has been fully created. The next step is to route the `Nets` using ADB, and then use JBits to create the output bitstream.

After the bitstream generation process, the simulator test-bench is created. For the oscillator design, three simulation steps were selected: verification of the initial values, values after one clock cycle, and values after an additional one hundred and one clock cycles. The simulation process begins by creating the `VTDS` object as shown on Line 32. After the `VTDS` object initializes and stabilizes the system using the `Bitstream` object created at the start of the code, the next step is to verify the initial output value for the flip-flop, FFout, and inverter output value, invOut. FFout and invOut are the names of the output nets created during the design phase. After comparing the two output values to the expected values, an output message is displayed indicating that simulation has either passed or failed. The program continues by clocking the device one time and rechecking the values; again outputting a verification message. The last step is to recheck the output value after one hundred one clock cycles, and displays the final simulation pass of fail message. The same techniques shown in the HelloVTsim file can be applied to more complex circuits using multiple clocks.

5.5 Other Design Flows

As discussed earlier, VTsim only requires a bitstream for input; any arbitrary bitstream can be simulated regardless of the tools used to create the bitstream. A bitstream generated by other tools will not have the optional .net file to use for simulation, which makes probing and

modifying `SimNets` difficult because there is no relationship between the `SimNets` created and the design file. When `VTDS` is used without a .net file, the simulator assigns the name of the `SimNet` to the source pin name value. For example, if a `SimNet` is connected to the X output of tile Row 5, tile Column 9, Slice 3, the associated `SimNet` name would be defined as: "Tile[5][9].X3". A user could also access the same value using the `getResourceValue` method from the given coordinates. Therefore, if the location of a specific source pin to be probed is known, a designer could access it using either of these two formats. Figure 5.4 illustrates a simple example of how to construct the `VTDS` object and perform this type of simulation.

```
 1: // Import the simulator package
 2: import JHDLBits.Virtex2.Simulator.*;
 3:
 4: public class HelloTile {
 5:   public static void main(String args[]) {
 6:     // Create the device simulator object
 7:     VTDS vtds = new VTDS("HelloTile.bit");
 8:
 9:     // Get value on output pin X at location: row=5, col=9, slice=3
10:     // Therefore the name is: Tile[5][9].X3
11:     int value0 = vtds.getSimNetValue("Tile[5][9].X3");
12:
13:     // Another way to access the same location is as follows
14:     int value1 = vtds.getResourceValue(5, 9, 3, "CLBslice.X");
15:
16:     if(value0 == value1) {
17:       System.out.println("Value0 = Value1.");
18:   }
19:     else{System.out.println("The two values do not match!");}
20: }
```

Figure 5.4: HelloTile example code

If placement information is not known, it may be difficult to determine specific primitive locations; although all methods are still available for use. Future versions of VTsim are expected to include a UCF parser designed to associate input and output pin definitions in the UCF file to VTsim IOB input/output `SimNet` names. The use of a UCF parser will allow full simulation of arbitrary bitstreams using pin information. For example, a user could set

values on input pins, pulse the clock and observe the values on the output pins. While this technique is available in the current version of VTsim, the user must know the location of the IOB associated with the desired output pin; the UCF parser will automate the process allowing test-bench designs similar to Figure 5.4.

Another feature of the simulator is the command line browser VTsim. VTsim is an interactive approach to the simulation and verification process. VTsim provides all of the get functions found in `VTDS`, and also includes a few extra features such as viewing flip-flop values and all of the `SimNet` names. Currently, VTsim does not support the set methods found in `VTDS`; however this could be extended rather easily if deemed useful.

The simulation models in this chapter present users with several different options to simulation and allows the user to choose which simulation model best suits their needs. Because VTsim is in the JHDLBits open source project, it is expected that users will make changes and improvements to the simulator allowing for better and possibly more convenient methods to interact with the simulator. The next chapter evaluates the performance of VTsim.

Chapter 6

Results

6.1 Overview

This chapter evaluates the performance of VTsim for all eleven Virtex-II devices using two separate tests to analyze simulation times and memory usage. The goal of the two tests is to provide an accurate depiction of the overall performance of VTsim for both simple and complex designs. The two tests have been designed to analyze the five distinct steps in the VTsim simulation process: 1) ADB trace, 2) Virtual device creation, 3) `SimNet` creation, 4) Device stabilization, and 5) Execution.

For testing purposes, the simple design consisted of one ten-bit counter. An adder circuit containing flip-flops and a large number of XOR gates served to represent a complex design. The ten-bit counter design was used to provide verification of the scalability of VTsim and to illustrate the initialization and execution times of a simple design for different-sized devices. The large stress-test design, which will be explained in the next section, forced the simulator queue to resolve a large number of events during each clock cycle.

Both test designs were created using the JHDL tool-suite. Initially, JHDLBits was used to create the Virtex-II bitstreams for testing in VTsim. However, because JHDLBits cannot generate Virtex device bitstreams, the Xilinx ISE tools were used to create both Virtex and Virtex-II bitstreams to ensure consistency during testing. As will be discussed throughout this chapter, VTsim went through several optimization steps during the results collection, aimed at improving simulator performance.

6.2 Input Test Files

The ten-bit counter design, shown in Figure 6.1, is composed of a ten-bit adder cell and a 10-bit register. The counter design was accomplished by tying one of the ten-bit inputs to ground, the other ten-bit input to the output of the ten-bit register, and by tying the carry-in high. The adder design was implemented using the high-speed carry chain found in both Virtex and Virtex-II devices. The clock enable for the registers was tied high so that during each clock cycle the counter would increment once.

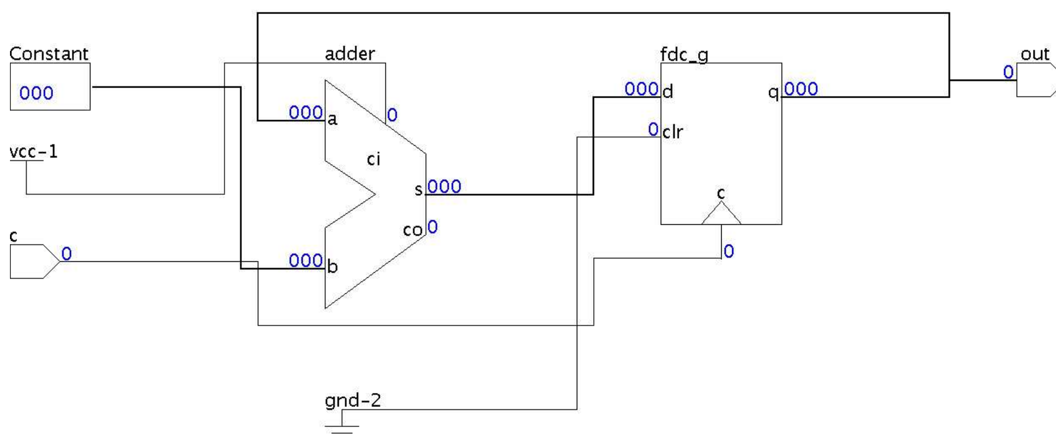

Figure 6.1: Ten-Bit Counter Design

The goal of the large stress-test design was to create a circuit that triggered an extremely large number of events during each clock cycle to stress the VTsim event queue and simulate

a worst-case scenario. The large stress-test design, shown in Figure 6.2, was implemented by instantiating 375 smaller cells. The number 375 was picked because it was the largest number of cells that could be implemented in the Virtex XCV1000, the largest Virtex device. Each small cell contains a nine-bit adder, nine flip-flops, one two-input XOR gate, six four-input XOR gates, one six-input XOR gate, and a five-input NAND gate.

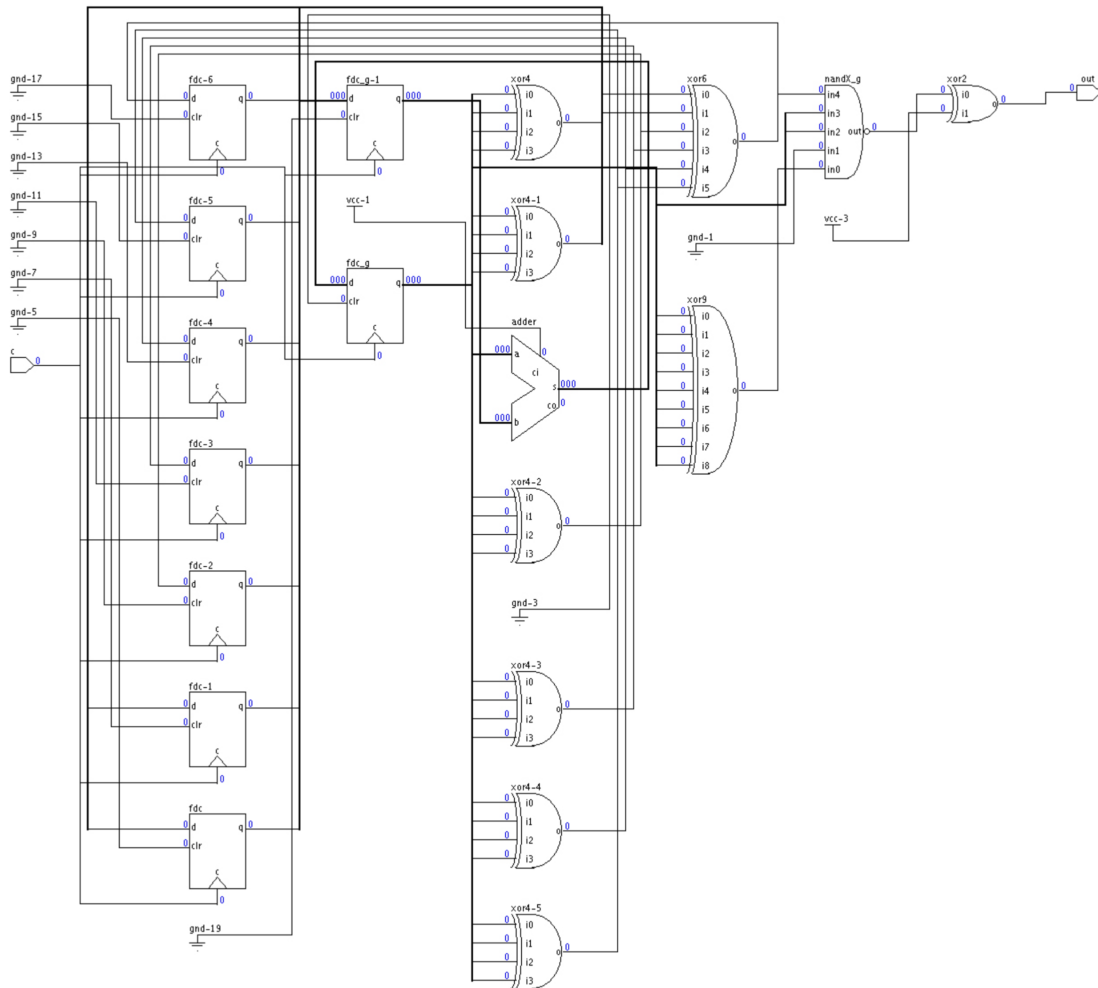

Figure 6.2: Stress Test Design

Time measurements for VTsim were made using the `System.currentTimeMillis()` method prior to calling the test method, and immediately after, with the difference between the two

being the total time. Because VirtexDS was accessed through the BoardScope interface[30], wall clock time was used for calculating simulation times. To minimize the amount of error introduced from this method, simulation runs of greater than one minute were used. Both programs were run from the command prompt using the following Java heap memory options: java -Xms512M -Xmx768M. The reason for choosing a very large initial and maximum heap size, 512 MB and 768 MB respectively, was to ensure that the Java Virtual Machine did not spend excess time constantly resizing memory. All time-based simulation tests were performed under Debian version Sid, Linux kernel version 2.6.5 on a 3.2 GHz Pentium 4 with HyperThread technology. Memory analysis was performed using JProbe [31] running Windows2000 on the same platform.

6.3 Event Queue Comparisons

Using a design that generated a large number of events not only provided a worst-case scenario, but also facilitated better comparisons of simple and complex designs via the event queue. As previously discussed, an event occurs when a `SimNet` value changes, causing all sink tiles are to be added to the event queue. Because a `HashMap` stores events using the tile location as the key, if an entry already exists in the `HashMap`, the location is overwritten with the same value again. During a clock cycle, multiple writes to the same location commonly occur; the total number of writes to the event queue is considered possible events. A processed event is an event that is removed from the event queue either during the clocked or non-clocked update cycle. Typically, there are many more possible events than processed events. A comparison of possible and processed events in simple and complex designs is illustrated in Table 6.1.

As presented in Table 6.1, the total number of possible events per second for a simple and a

Table 6.1: Event Comparisons for 1,000 Clock Cycles

Criteria	**10-Bit Counter**	**Stress Test**
Possible Events	13,838	22,174,630
Processed Events	11,227	11,404,068
Execution Time	13.761s	75.117s
Possible Events/Sec	301,683.37	295,201.2
Processed Events/Sec	244,753.9	151,817.4

complex design are approximately the same, with values within two percent of each other. In a simple design, approximately eighty-one percent of possible events are processed events. Because of the complexity of the large design, and the choice of tile (CLB) granularity, only fifty-one percent of possible events become processed events. Initial thought lead to the conclusion that the granularity level was chosen too high. However, after a deeper study into where the events occurred, it was noted that most the event sink pins were connected to the function generator inputs. Therefore, reducing the granularity level to slice or even LE would not improve upon this problem, thus reinforcing the choice of tile granularity.

After seeing how many possible and processed events occur, it became apparent the importance of a simplified event queue. Several small modifications to the event queue structure provided improvements to execution times and are noted here. One major performance goal was ensure that simulation was design dependent and not device dependent, meaning that a design placed in the smallest and largest device should have the same simulation execution times. Initially, comparisons between small and large devices using the ten-bit counter design yielded almost exponentially increasing simulation times: the small device simulated in eleven seconds and the largest device completed simulation after approximately one minute. This was puzzling because tests indicated that the same number of events was

being processed in both the small and large devices.

After a lengthy analysis, the exponential growth was attributed to the size of the `HashMap`. During initial stabilization of the device, all tiles are placed on the queue for analysis causing the event queue to grow to the total number of tiles. After stabilization was complete, the `HashMap clear()` method was called to ensure no keys were left on the `HashMap`. If the data in Table 6.1 is extrapolated, approximately fourteen possible events occur and eleven events are processed. Because the `HashMap` size is equal to the total number of tiles, the `HashMap remove()` method requires more time to search the map for the correct key. Instead of calling the `clear()` method at the end of stabilization and after each clock cycle, the `HashMap` was recreated. By doing so, the one minute execution time for the largest device decreased to eleven seconds.

The simple recreation of the `HashMap` provided equal execution times for all devices. Recreating the `HashMap` instead of using the `clear()` method did not cause a noticeable change in execution times for the ten-bit counter design for the smallest device, or for the large complex design, meaning that resizing the `HashMap` did not account for much time. Therefore, it is the author's conclusion that Java programmers should not use the `clear()` method for `HashMaps` that vary in size, but instead recreate the `HashMap`.

Simulator optimizations were also made in the event queue interaction with `SimNets`. Whenever an event was processed, the tile (event) obtained the values of all input and output `SimNets` by calling a `getValue()` method on each `SimNet`. The `getValue()` method was a single line that returned the current value on the desired `SimNet`. By bending the rules of object oriented programming and accessing the value directly, `mySimNet.value`, a ten-percent speed improvement of cycle execution times was achieved. Although this was not a highly-desired approach because of the fixed dependencies on the variable value, the performance improvement was significant enough to warrant retaining the change.

6.4 Detailed Memory Analysis

One important aspect of the simulator design was to keep memory usage low, preferably with a base memory usage below 64 or 128 MB. Base memory usage was considered the amount of memory required to configure and initialize an empty design for a device. Table 6.2 expresses the base memory usage for all Virtex-II devices. The white area in the figures indicates the allocated memory heap size and the blue area represents the memory currently in use. Base memory starts off small for the XC2V40 device, but grows 126 MB for the largest Virtex-II device. While all designs used less than the 128 MB desired limit, the growth in memory prior to ADB trace time to cycle execution time for the larger devices is cause for concern.

Table 6.2: VTsim memory usage

Device	**Memory prior to ADB trace (KB)**	**Memory before cycle execution (KB)**
XC2V40	8,980	9,761
XC2V80	9,514	11,145
XC2V250	11,264	13,714
XC2V500	13,377	18,905
XC2V1000	15,768	23,377
XC2V1500	19,062	31,715
XC2V2000	22,551	38,343
XC2V3000	27,955	49,805
XC2V4000	38,220	71,317
XC2V6000	51,058	97,556
XC2V8000	64,557	129,695

One way to reduce the maximum memory usage would be to remove ADB and JBits from

memory when they are no longer needed. Currently both ADB and JBits reside in memory until simulation completion. After VTsim completes construction and configuration of the `SimNets`, neither ADB nor JBits is used, so both could be removed from memory. This could reduce memory usage by as much as 70 MB for the largest devices. Because the maximum base memory usage occurs after the `SimNet` creation phase, the peak memory usage would still remain around 126 MB, but memory usage during cycle execution would be reduced in half. Although this would not improve the base memory usage, it would ensure that the maximum memory usage would occur in initialization and not during cycle executions. Based on this information, Java's maximum heap size must be increased to at least 128 MB, but it is recommended to use a maximum heap size of 256 MB to ensure that memory overflow does not occur.

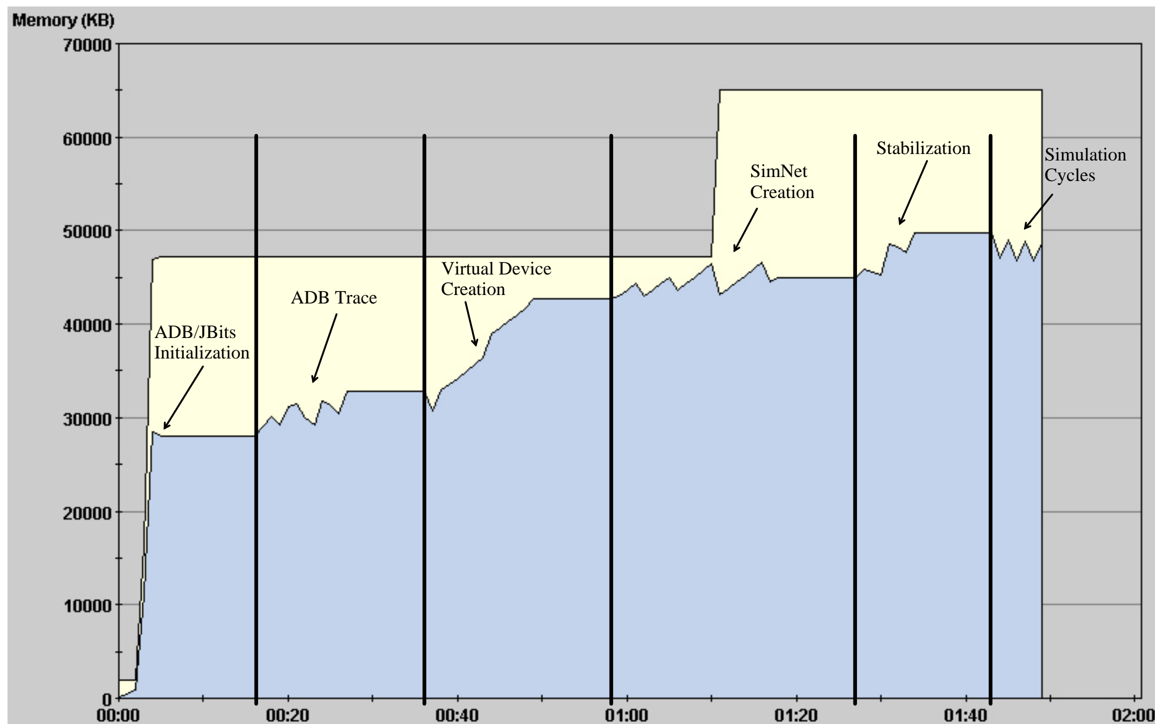

Figure 6.3: VTsim: 10-Bit Counter Memory Profile

Figure 6.3 shows the memory profile for the ten-bit counter implemented on a Virtex-II XC2V3000 device. Figure 6.4 depicts the memory profile for the stress-test design; also implemented on a Virtex-II XC2V3000 device. Pauses, depicted in both graphs by a ten second flat line, were placed in the simulator code to provide a way to decipher the five different simulation steps. The first pause in both figures indicates the amount of memory required to initialize both JBits and ADB, approximately 27 MB. The increase in memory to the next pause is the memory required to perform the ADB trace function. The stress-test design requires slightly more memory for the trace, however considering the complexity of the design, the increased memory is minimal. The next step in the initialization is the creation of the virtual device. The creation of the virtual device is device dependent and uses the same amount of memory for both designs.

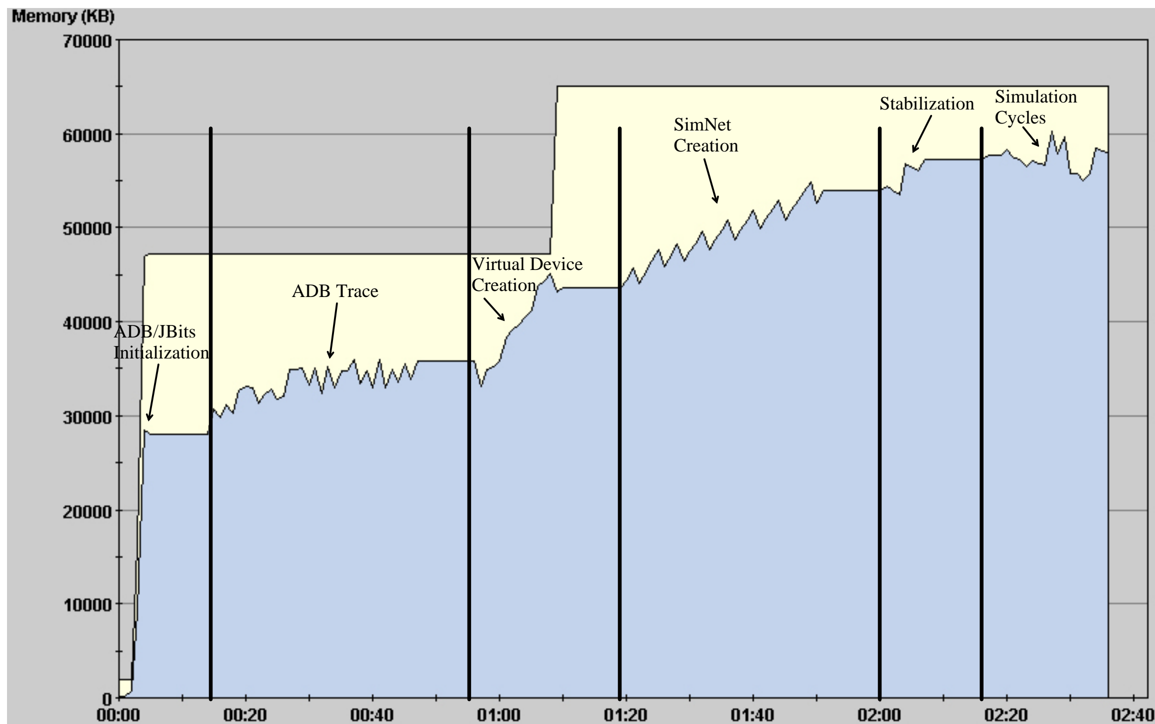

Figure 6.4: VTsim: Stress Test Memory Profile

After the virtual device was created, ADB `TraceNodes` were evaluated to determine the internal connections and create the `SimNets`. The stress-test design requires approximately an additional 9 MB of memory to create and store the connection information. Future work on the simulator will focus on reducing this memory overhead by optimizing the `SimNet` class. After the next pause, the stabilization of both devices requires approximately the same amount of memory. This is because all tiles are placed on the event queue for execution and therefore the maximum size should be the same. The complex design required more memory during the simulation phase primarily because the size of the event queue was much larger than that of the simple design. Memory comparisons of the two designs showed that VTsim is relatively design independent in all areas except `SimNet` creation. It would be worth investigating the possibility of reducing `SimNet` memory usage because there is such a large discrepancy between the two designs.

6.5 Simple Design Results

The ten-bit counter circuit was used because the design is simple, easy to test and verify, and does not trigger many events. The counter design is small, and fits into all Virtex and Virtex-II FPGAs allowing comparisons between different Virtex and Virtex-II devices as well as inter-family comparisons for VTsim and VirtexDS. Because the VirtexDS code is not made public, only clock cycle simulations could be performed. A common term used throughout this section is “normalized time”, which is calculated by taking the time spent in a given method divided by the total number of tiles in the device multiplied by 10,000. This calculation normalizes the specific device time to a standard 10,000 tile device to provide a clearer method of analysis. For example, some the figures described throughout this section appear to have an exponential growth rate. However, because the number of tiles in a device grows in an exponential manner for the larger devices, the overall simulation times

appears to follow the same curve. The normalized figures allow analysis of overall trends. If a normalized curve grows exponentially, it typically means that additional optimizations are needed for that section of code. A linearly or exponentially decaying curve typically means that the method has a large amount of initial overhead and that the overhead is amortized throughout larger designs.

6.5.1 ADB Trace Time

The first step in the VTsim initialization process is the trace function performed by ADB. Figure 6.5 illustrates the time required to perform the trace function for all Virtex-II devices. As expected, the ADB trace time increased as the device size increased. The figure seems to indicate an exponentially growing curve, but the normalized trace time (trace time/tile) remains relatively constant between nineteen milliseconds and twenty-two milliseconds; however, the size of the device grows tremendously making it appear as if the growth was non-linear.

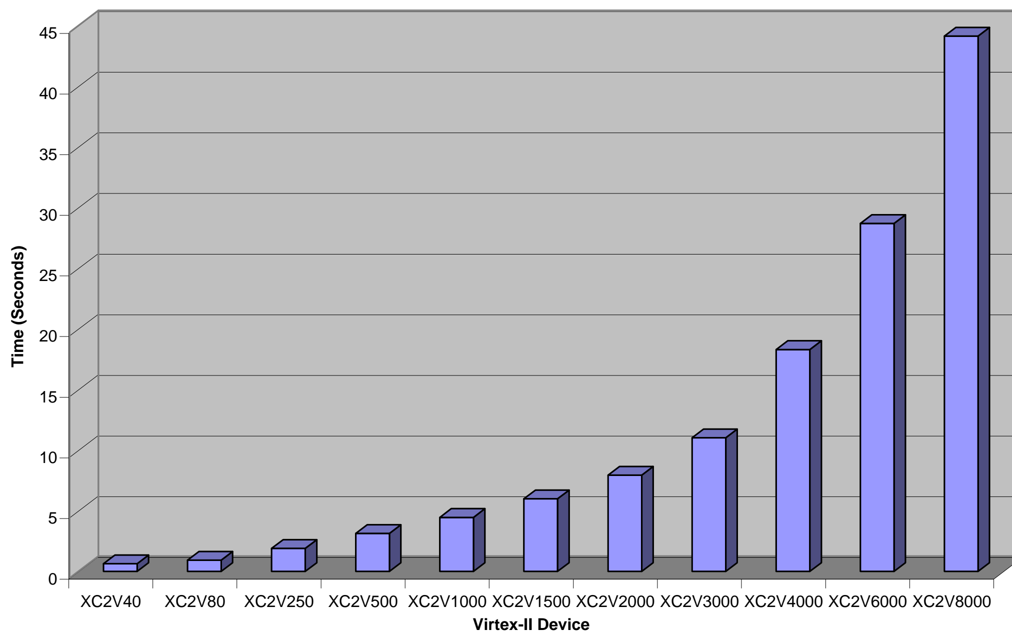

Figure 6.5: ADB Trace Time for Virtex-II Devices - 10-Bit Counter Design

6.5.2 Virtual Device Creation Time

The next step in the simulator initialization process is the creation of the virtual device. The virtual device process creates the proper tile object based on the given location and configures the device from the bitstream using JBits. The virtual device creation phase is a combination of JBits calls to access the bitstream and tile construction. Figure 6.6 presents the virtual device creation time for all Virtex-II devices.

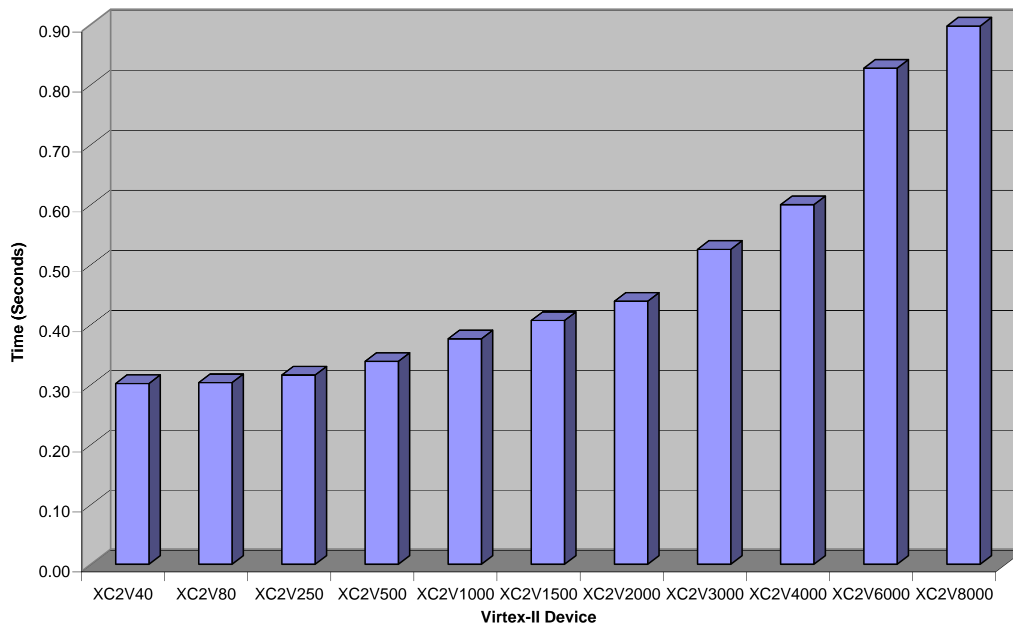

Figure 6.6: Virtual Device Creation Time for Virtex-II Devices - 10-Bit Counter Design

As expected, the virtual device creation time increased with device size, but the normalized virtual device creation time decreased exponentially as the size of the device increased. This indicated that a lot of overhead processing time was spent at the beginning of the process. Because virtual device creation time accounts for less than a second in the largest device, optimizations of the process were not considered important for the initial release.

6.5.3 SimNet Creation Time

Figure 6.7 illustrates the `SimNet` creation process. Similar to both the ADB trace time and device creation time, the `SimNet` creation time increased with each larger Virtex-II device. The `SimNet` creation curve follows a similar curve to the ADB trace time graph with the exception that the `SimNet` creation times are initially lower and end slightly higher. Further analysis of `SimNet` creation times shows a linearly growing normalized `SimNet` creation time. Because the `SimNet` creation time can be in excess of 45 seconds for large devices, this may be an area of concern.

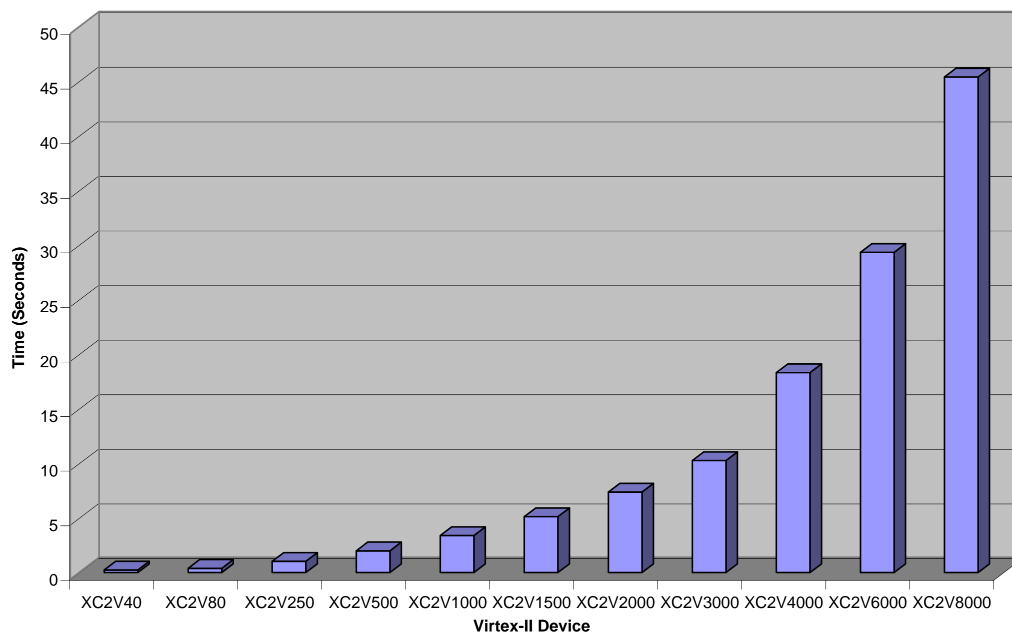

Figure 6.7: SimNet Creation Time for Virtex-II Devices - 10-Bit Counter Design

Additional work in the `SimNet` creation process is necessary. Extending the ADB `TraceRoute` method and incorporating the `SimNet` creation process into the trace function would most likely reduce combined ADB trace time and `SimNet` creation time. Because the `SimNet` creation process traverses ADB `TraceNode` structures in search of pin connections, the `SimNet` creation process is basically re-tracing the FPGA. Therefore, extending the `TraceRoute`

method is a next logical step in the simulator design. Considering the amount of time spent in the `SimNet` creation process, extending the `TraceNode` method could drastically reduce VTsim initialization times.

6.5.4 Stabilization Time

After the virtual device has been completely configured the next step is to stabilize the device. Stabilization is necessary because the values inside each tile have not been updated from their default creation values. The stabilization process propagates all `SimNet` values through the circuit by executing all non-clocked elements throughout the virtual FPGA.

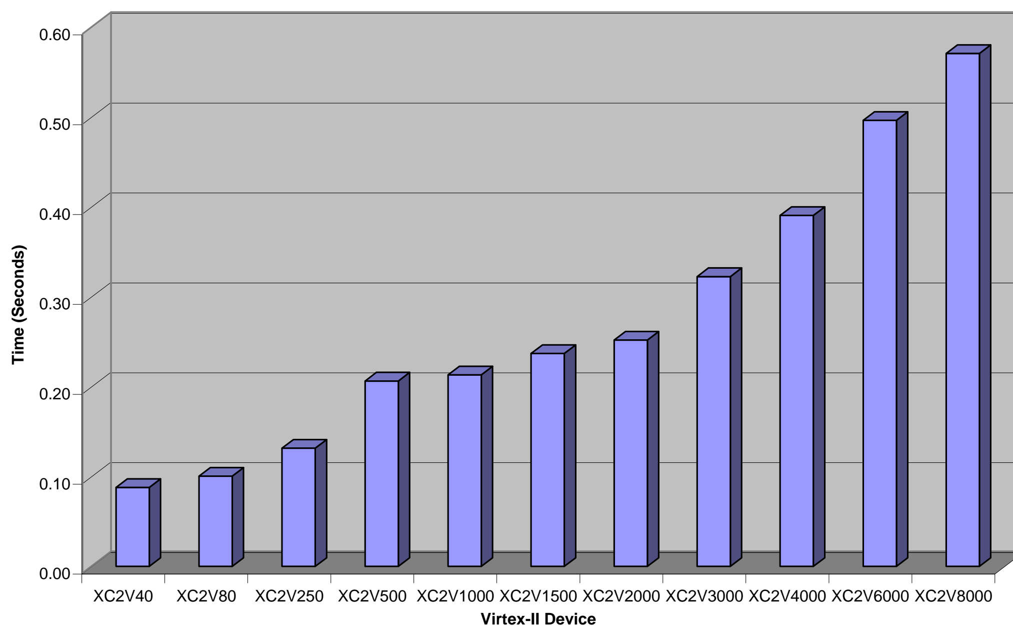

Figure 6.8: Stabilization Time for Virtex-II Device - 10-Bit Counter Design

The VTsim device stabilization time for all Virtex-II devices is illustrated in Figure 6.8. The graph depicts almost linearly increasing stabilization times, all of which are under one second. Analysis of the normalized stabilization time yields an exponentially decreasing curve similar to the virtual device creation time. This can be attributed to initial overhead in the stabilization time process and interaction with the `HashMap`.

6.5.5 Simulation Execution Time

Once the virtual device is stabilized, it is ready for clock execution. As discussed earlier, a clock cycle begins by creating an event on a global clock. This event triggers all sink tiles connected to the clock line to be placed on the event queue. After all tiles on the clocked event queue have been processed, the non-clocked event queue is executed. The procedure of creating a global clock event, executing the clocked event queue, and then executing the non-clocked queue is considered a single simulation cycle. Figure 6.9 illustrates the time required to execute 300,000 clock cycles for the simple ten-bit counter design.

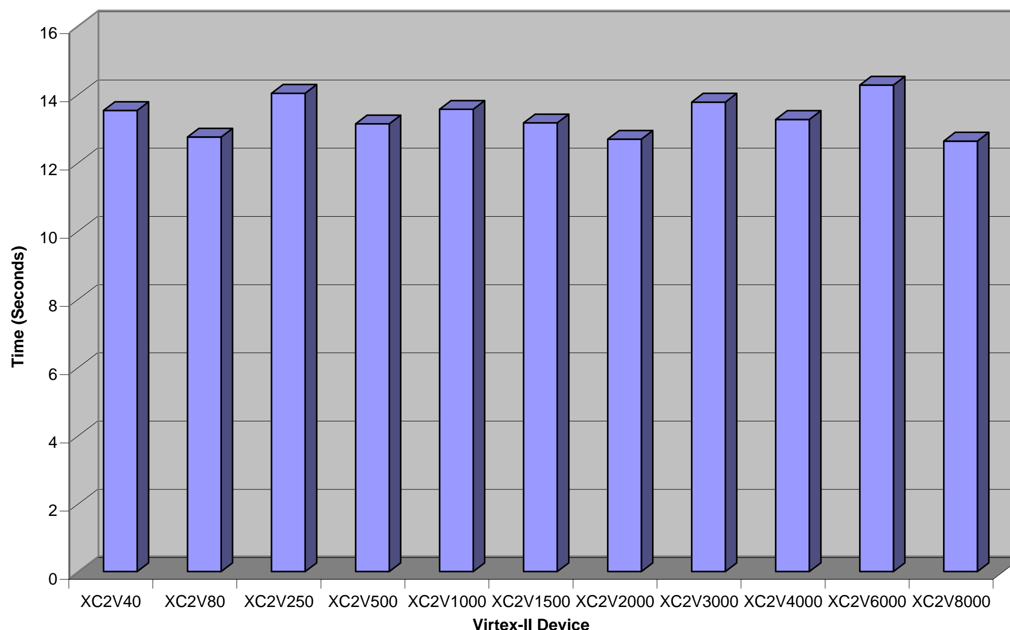

Figure 6.9: 300,000 Cycle Execution Time for Virtex-II Devices - 10-Bit Counter Design

Expanding on the discussion in Section 6.3, VTsim simulation times are constant across all Virtex-II FPGAs because of optimization improvements in the event queue structure. This implies that the design of the event queue and other related methods are device independent. Because of the simplicity of the ten-bit counter design and minimal number of triggered events, the execution speed of the simulator for this design can be considered optimal.

Figure 6.10 shows that the maximum simulation speed of the simulator is approximately 22,500 cycles per second and is constant for all Virtex-II devices.

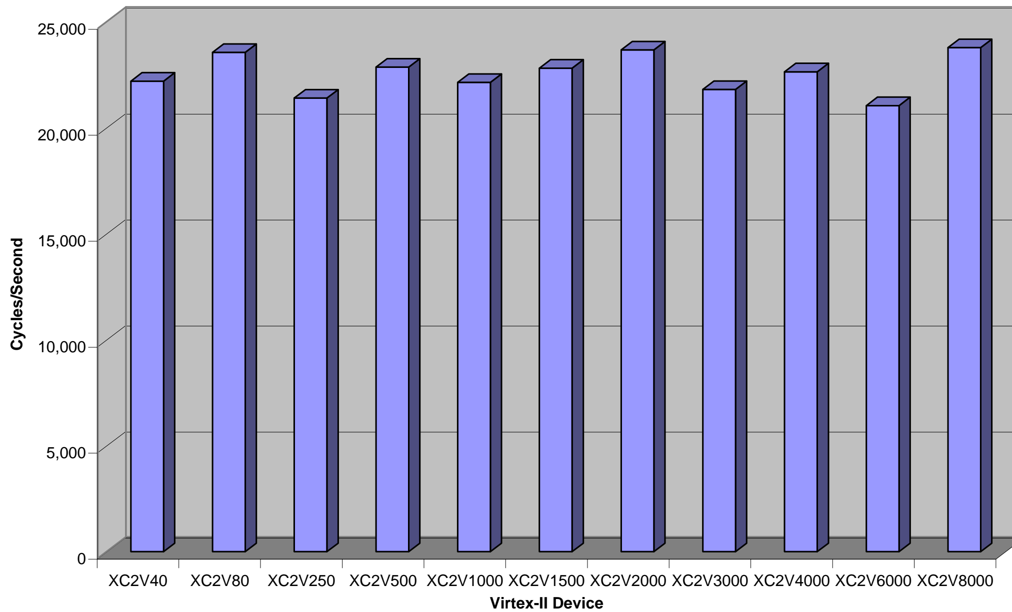

Figure 6.10: Cycles-per-Second Analysis of Virtex-II Devices - 10-Bit Counter Design

6.6 Small and Large Design Comparisons

To provide a more accurate picture of VTsim's performance, simulation results for the small, simple ten-bit counter design were contrasted against the large, complex stress test design. Figure 6.11 portrays the differences in virtual device creation times and stabilization times. The virtual device creation time for both circuits was essentially the same. This was expected because the virtual device creation process performs the same number of JBits and ADB calls regardless of the implemented design.

Surprisingly, the stabilization time for the complex design took approximately the same amount of time to execute as the simple design. It was hypothesized that because the

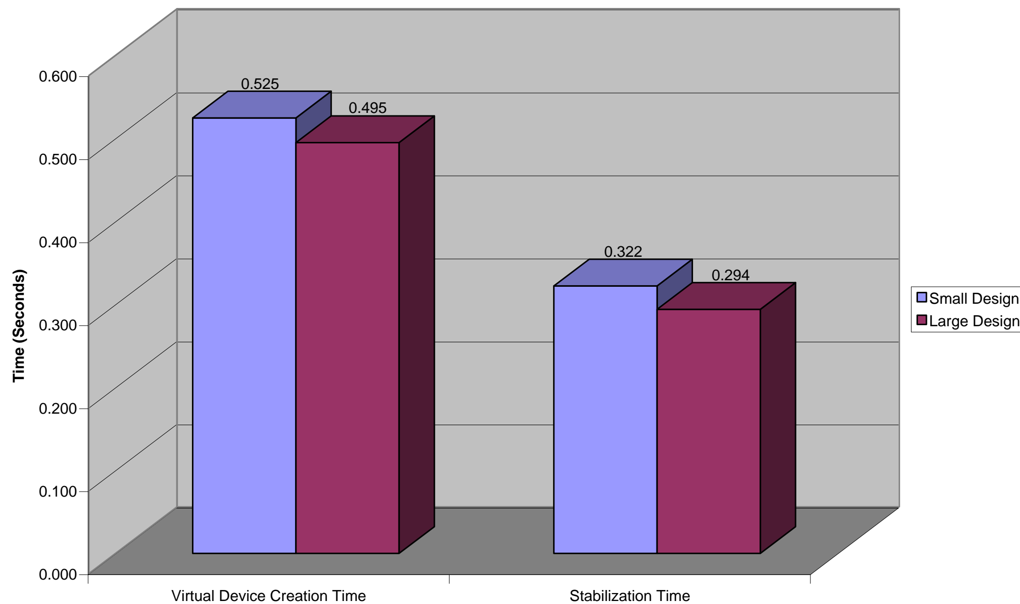

Figure 6.11: Comparison Between Small and Large Designs

complex design had a large number of `SimNets`, a large number of the `SimNet` values would need to change to stabilize the system. The reason that the two times are practically the same is because the `SimNet` default value is zero during initialization, and all the flip-flops were initialized to zero. Therefore, there were no 'one' values to propagate through the circuit to change any of the XOR gates.

Figure 6.12 illustrates the differences in ADB trace times, `SimNet` creation times, and 1,000 cycle execution times for both the simple and complex designs. It was expected that the ADB trace time would remain relatively constant across designs, following the assumption that ADB evaluated all routing configuration bits in the same manner that VTsim analyzes the logic configuration bits. In a discussion with the ADB author, it was clarified that ADB trace times varied between the designs because of the increasing complexity of storing the internal connections into data structures as the size of each routing net increased.

The `SimNet` creation time only increased slightly for the larger design, which was unexpected.

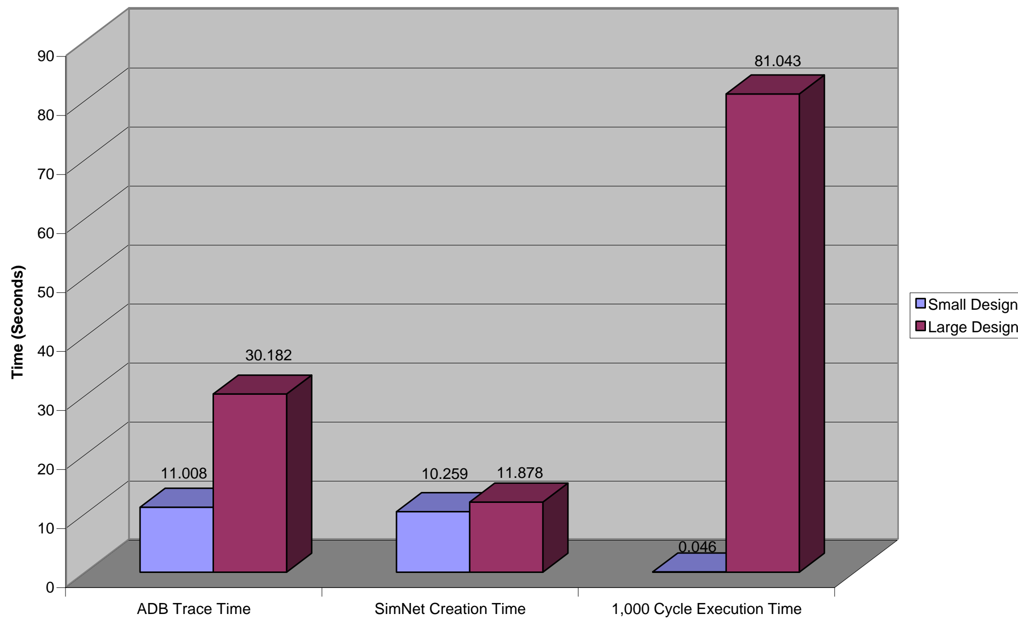

Figure 6.12: Comparison Between Small and Large Designs

It was originally expected that tracing a bitstream with few routes would be much faster than when traversing ADB `TraceNodes` in search of tile input pins. The results have a propensity to indicate that the ADB `TraceNodes` between the two designs were relatively the same length. This is understandable because the complex design was placed efficiently and routes should be relatively short. It would be interesting to test a design that had the output of one flip-flop driving all other flip-flops throughout the device. Such a design would lead to long complex routes that should require a much longer `SimNet` creation time.

The 1,000 cycle comparison between the two designs presented the drastic difference in device complexity. As discussed earlier in Section 6.3, the complex design had over sixteen thousand times more possible events and over 1,000 times more processed events. These events account for the difference in execution times. To summarize the table, the small design executes approximately twenty-two thousand cycles per second and the complex design executes slightly more than twelve cycles per second. The next section will compare both of

these results to similar designs implemented in Virtex devices and simulated using VirtexDS.

6.7 VirtexDS Comparisons

The counter and stress test designs used to test VTsim were also implemented in Virtex FPGAs for the analysis of VirtexDS. The small counter design was used to determine the maximum performance of VirtexDS and also provide conclusions on the scalability of VirtexDS for different devices. The VirtexDS comparisons were made using the same hardware setup as the VTsim tests. Figure 6.13 illustrates the required execution times for the ten-bit counter for different Virtex Devices.

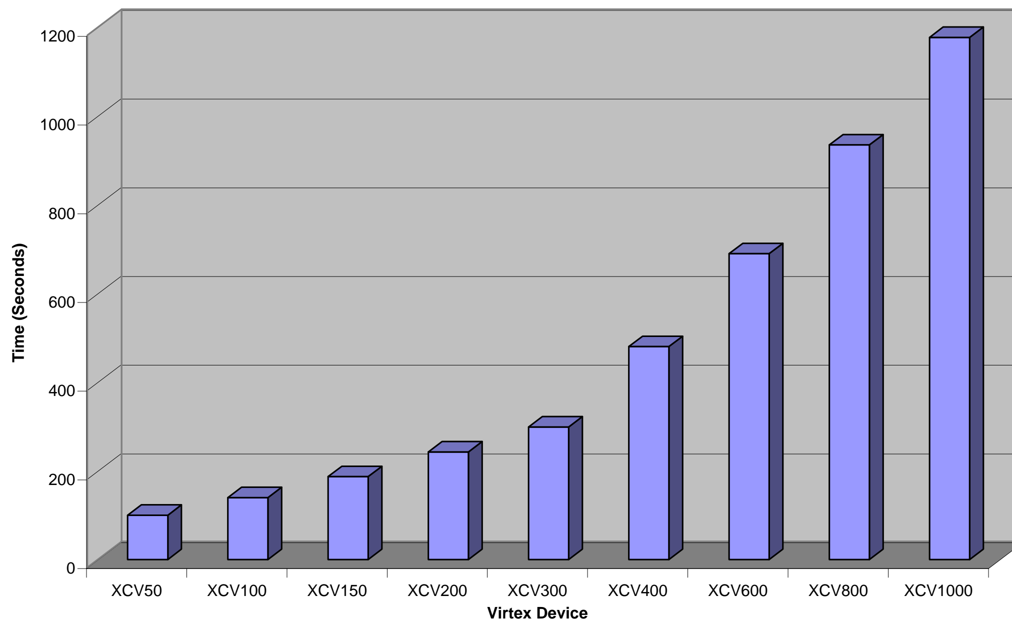

Figure 6.13: 300,000 Cycle Execution Time for Virtex Devices - 10-Bit Counter Design

The graphs in Figure 6.13 suggest that VirtexDS did not scale very well between different sized devices. Execution time for the largest device was nearly twelve times longer than for the smallest device. Figure 6.14 indicated the same pattern seen with in the three hundred

thousand cycle execution times. Simulation of the smallest device, XCV50, yielded approximately three thousand cycles per second in the beginning, and dropping to approximately two hundred fifty cycles per second for the XCV1000.

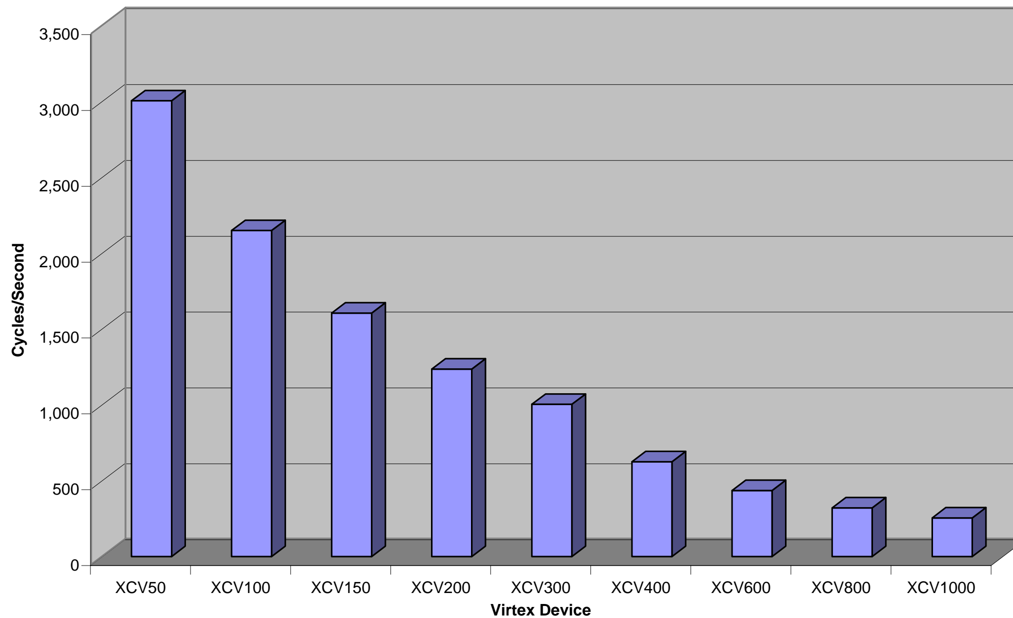

Figure 6.14: Cycles-per-Second Analysis of Virtex Devices - 10-Bit Counter Design

6.8 VirtexDS vs. VTsim

One goal for VTsim was to create a simulator with faster cycle execution times than the predecessor VirtexDS without sacrificing any functionality. The non-optimized version of VTsim performed approximately fifteen times better than VirtexDS for the ten-bit counter design, but almost four times as slow as VirtexDS for the stress-test design. Through the optimization described in Section 6.3, VTsim improved simulation execution time by nearly sixty percent, surpassing the VirtexDS simulator for all designs. Figure 6.15 compares the execution times between Virtex and Virtex-II devices for the simple ten-bit counter design. Note that the Virtex and Virtex-II devices are different sizes, but because VTsim's cycle

execution time does not vary with device size, a comparison can be made between the devices. VTsim is roughly 7.5x faster than VirtexDS for the smallest Virtex device and nearly 90x faster than the largest Virtex device.

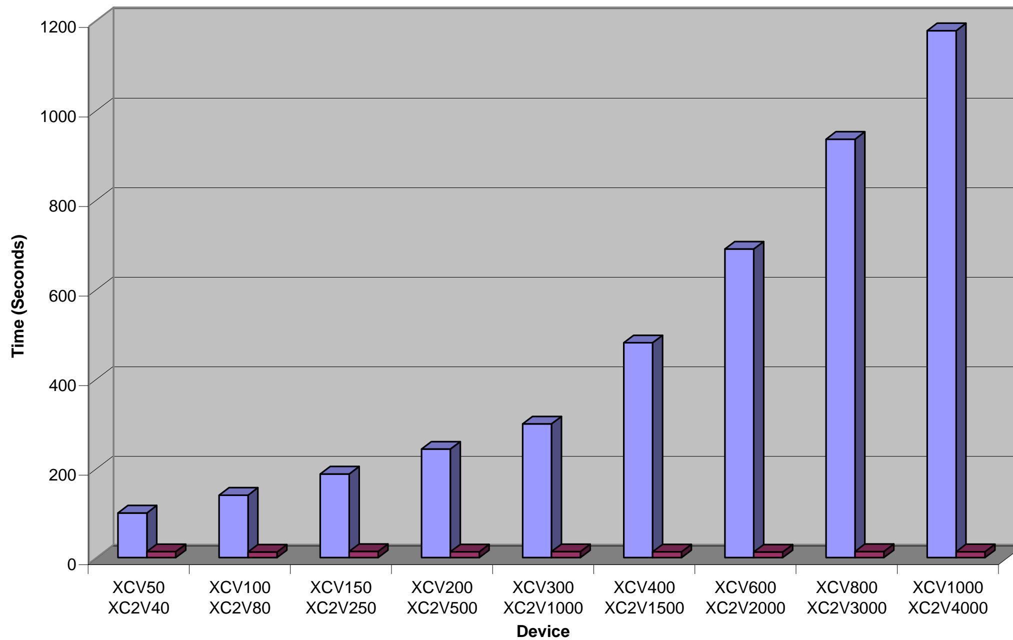

Figure 6.15: 300,000 Cycle Execution Time Comparison - Virtex vs. Virtex-II Devices

Contrary to the ten-bit counter design, which shows best-case simulation performance, the stress-test design conveys worst-case simulation performance. Figure **??** depicts a comparison of execution times for 1,000 clock cycles. VTsim's simulation lead is cut down from 90x faster for simple designs to only 2.5x faster for the complex design. One possibility for this drastic decrease in the performance gap may be attributed to how the event queue chooses the next event. Because the events are stored in a `HashMap` for VTsim, the execution order of events is not defined, which could possibly hinder VTsim's performance for large designs. Although ordering of the queue could possibly lead to faster execution times for certain designs, it is expected that queue ordering may slow other design execution times where order may not be as critical. The overhead required to order the queue may also out-weight the benefits of an ordered queue. Based on these hypotheses and experiments performed in the queue

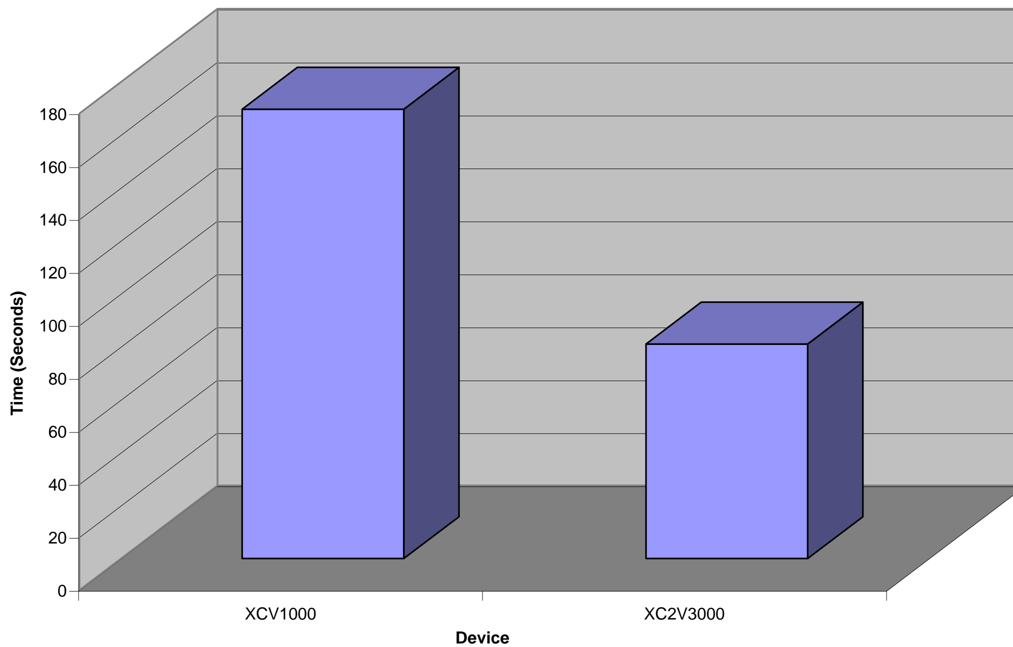

Figure 6.16: Virtex and Virtex-II: 1,000 Cycle Execution Time for the Stress Test

optimization process, the current recommendation is to leave the event queue model in its current state.

Figure 6.17 shows that VTsim executes twice as many cycles per second. Analysis of both the simple and complex designs leads to the conclusion that VTsim is completely device independent, but is more design dependent than VirtexDS. VirtexDS on the other hand is device dependent, but more design independent than VTsim.

In summary, this chapter analyzed and evaluated the performance and memory usage of VTsim for two extreme test cases. In the next chapter, conclusions and future simulator explorations will be presented.

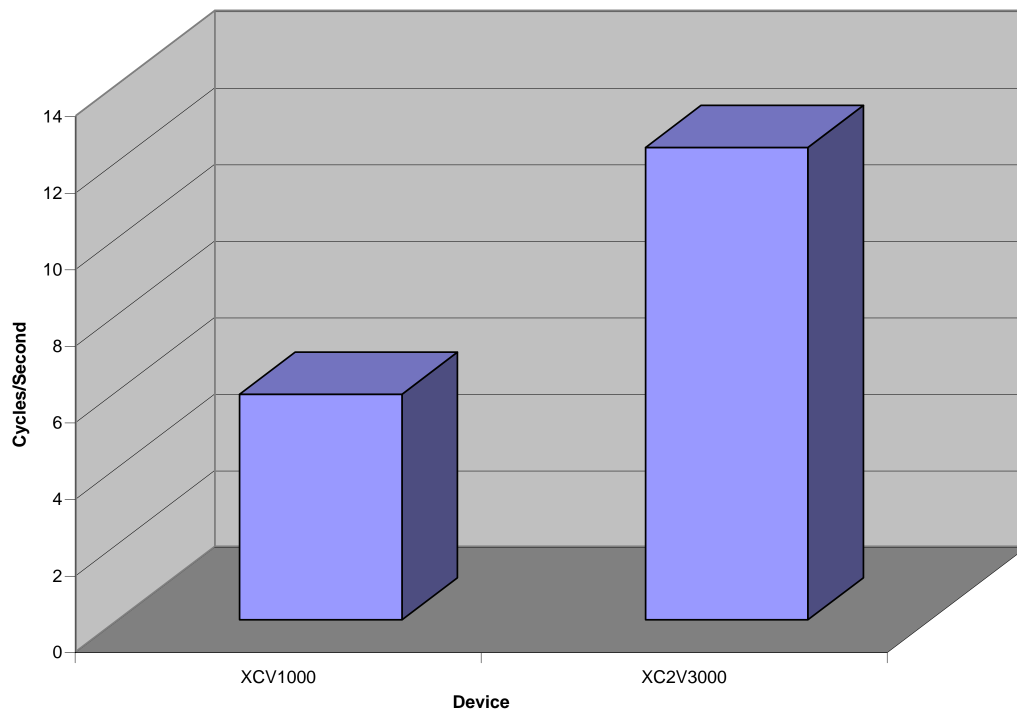

Figure 6.17: Virtex and Virtex-II: Cycles-per-Second Comparison for the Stress Test

Chapter 7

Conclusions

7.1 Summary

This thesis described VTsim, a Virtex-II device simulator with multiple clock domain support. Through both examination and analysis, VTsim demonstrated rapid response, relatively low memory usage, and outperformed its predecessor, VirtexDS by up to 9,000%. The majority of the goals set for VTsim were achieved with the exception of 100% resource coverage. At the time of writing, resource coverage is at approximately 90%, but the simulator includes support for the majority of logic commonly used in FPGA designs. The simulator framework and API were designed to allow future developers to easily integrate new features and tile types. Because VTsim is part of the JHDLBits open source endeavor, user feedback and changes are welcomed and encouraged. The open source model provides the structure necessary for VTsim to be adapted and improved with relative ease.

Although VTsim is a fully functional simulator, some additions would help make it both easier to use and more flexible. The inclusion of an interactive internal FPGA viewer/editor

would expand the audience served by VTsim. The interactive viewer would allow users to examine or change FPGA resources at levels as specific as the individual configuration bits. Including support for other tiles types not currently implemented, especially the hardware multipliers, Block SelectRAM, and DCMs, would allow for simulation of additional designs.

At the time of writing, VTsim was the sole bitstream simulator available for Virtex-II FPGAs. For this reason, it is hoped that the design and implementation of VTsim will serve as a useful tool for the research community.

Bibliography

[1] Xilinx, Inc., “Design Tools Center.” http://www.xilinx.com/products/design_resources/design_tool/grouping/logic_design_prod.htm.

[2] Mentor Graphics, “ModelSim PE.” http://www.model.com/products/pe.asp.

[3] Altium Limited, “Nexar - The Complete Solution to Developing Embedded Systems on FPGAs.” http://www.altium.com/nexar/index.html.

[4] A. Poetter, J. Hunter, P. Athanas, and C. Patterson, “JHDLBits: The Merging of Two Worlds,” in *14th International Workshop on Field-Programmable Logic and Applications, FPL 2004*, (Antwerp, Belgium), August 2004.

[5] Open Source Development Network, Inc. (”OSDN”), “SourceForge.net: The World’s Largest Open Source Software Development Website.” http://www.sourceforge.net/.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Xilinx, Inc., "Xilinx Chips Land On Mars." http://www.xilinx.com/prs_rls/design_win/0412_marsrover.htm.

[11] F. L. Kastensmidt, G. Neuberger, L. Carro, and R. Reis, "Designing and Testing Fault-tolerant Techniques for SRAM-based FPGAs," in *Proceedings of the First Conference on Computing Frontiers*, pp. 419–432, ACM Press, 2004.

[12] Xilinx, Inc., "Corporate Backgrounder." http://www.xilinx.com/company/press/grounder.htm.

[13] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, 2002.

[14] R. Jayaraman, "Physical design for FPGAs," in *Proceedings of the 2001 International Symposium on Physical Design*, pp. 214–221, ACM Press, 2001.

[15] R. Swan, A. Wyatt, R. Cant, and C. Langensiepen, "Re-configurable Computing," *ACM Crossroads Student Magazine - Computer Architecture*, vol. 5.3, Spring 1999.

[16] S. Trimberger, "Effects of FPGA Architecture on FPGA Routing," in *Proceedings of the 32nd ACM/IEEE Conference on Design Automation*, pp. 574–578, ACM Press, 1995.

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

[18] Xilinx, Inc., "Virtex-II Platform FPGA Overview." http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=v2_overview.

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

[20] 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.

[21] 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.

[22] A. Poetter, P. Athanas, and C. Patterson, "JHDLBits," in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'04)*, June 2004.

[23] M. Shoji, F. Hirose, S. Shimogori, S. Kowatari, and H. Nagai, "Acceleration of Behavioral Simulation on Simulation Specific Machines," in *Proceedings of the 1997 European Conference on Design and Test*, p. 373, IEEE Computer Society, 1997.

[24] J. R. Armstrong and F. G. Gray, *VHDL Design Representation and Synthesis*. Prentice Hall PTR, second ed., February 2000.

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

[26] L. Bening, "A Two-State Methodology for RTL Logic Simulation," in *Proceedings of the 36th ACM/IEEE Conference on Design Automation*, pp. 672–677, June 1999.

[27] 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.

[28] Xilinx, Inc., "FPGA Editor Help: Overview." Help Documentation Included With Xilinx ISE 5.2i Tools.

[29] Neil Steiner, "ADB Javadocs." http://www.ccm.ece.vt.edu/jhdlbits/javadoc-ADB/index.html.

[30] D. Levi and S. A. Guccione, "BoardScope: A Debug Tool for Reconfigurable Systems," in *Configurable Computing Technology and its use in High Performance Computing, DSP and Systems Engineering, Proc. SPIE Photonics East* (J. Schewel, ed.), pp. 239–246, November 1998.

[31] Quest Software Inc., "JProbe Suite." http://www.quest.com/jprobe/.

Vita

Jesse Everett Hunter III

Jesse was born in rural Pennsylvania in 1977 to Jesse and Cary Hunter. His hard working parents supported his love for sports and music. Between his junior and senior year of high school, Jesse spent a year in Finland as a Rotary exchange student and was able to visit numerous countries across Europe. To date, Jesse has visited more countries than states. After graduating with a Bachelor of Science in Electrical Engineering, he married Karen Hutchinson. Upon completion of a Master of Science in Computer Engineering, Jesse plans to work in industry for several years before returning to academia to pursue a doctoral degree.