

A Standalone Wire Database for Routing and Tracing in Xilinx Virtex, Virtex-E, and Virtex-II FPGAs

Neil Joseph Steiner

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
Electrical Engineering

Peter Athanas, Chair
Mark Jones
Walling Cyre
Cameron Patterson

August 30, 2002
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

keywords: FPGA, routing, tracing, JBits, Virtex, Virtex-E, Virtex-II, Xilinx, ADB

Copyright © 2002, Neil Joseph Steiner. All Rights Reserved.

A Standalone Wire Database for Routing and Tracing in Xilinx Virtex, Virtex-E, and Virtex-II FPGAs

Neil Joseph Steiner

Abstract

Modern FPGAs contain routing resources easily exceeding millions of wires. While mainstream design flows and place-and-route tools make very good use of these routing resources, they do so at the cost of very significant processing time. A well established alternative scheme is to modify or generate configuration bitstreams directly, resulting in more dynamic designs and shorter processing times. This thesis introduces a complete set of alternate wire databases for Xilinx Virtex, Virtex-E, and Virtex-II FPGAs, suitable for standalone use or as an addition to the JBits API. The databases can be used to route or trace through any device in these families, and can generate the necessary bitstream configurations with the help of JBits or an independent bitstream interface.

As an aspiring designer, I must acknowledge my own designer, who saw fit to make me, to give me a wide range of interests and abilities, to open doors and provide opportunities to pursue those interests, and to know him, partially in this life, and fully in the next. There are those who choose to believe against all odds in chance and random chaotic processes, but I choose instead to believe in a creator, who has compellingly demonstrated his qualifications through the universe around us, and who decided that the most important thing in life was to be in relationship with us, even if it cost him everything.

Acknowledgements

Thanks to Dr. Peter Athanas, my academic advisor, for taking a chance on me, and giving me the freedom to work as I thought best. Dr. Athanas is the reason why I found out about Virginia Tech, and the reason why I didn't bother applying anywhere else.

Thanks to Dr. Mark Jones, for co-managing the Configurable Computing Lab, and nonchalantly dropping profoundly insightful comments on those of us lucky enough to be within earshot.

Thanks to Dr. Walling Cyre, for his impressive editing assistance, even though the paper in question never was accepted, and for those C++ assignments which provided much needed therapy against the Java onslaught.

Thanks to Dr. Cameron Patterson of Xilinx, for lending his considerable expertise, and for looking out for my professional advancement without any solicitation on my part.

Thanks to my committee members collectively, for trudging through this inordinately long thesis. I won't fault you if you gloss over some of it.

An enormous debt of gratitude goes to Jeff Lindholm of Xilinx for the inside knowledge and expertise which was simply not available anywhere else. Jeff remained gracious and helpful throughout my incessant barrage of questions, and without his assistance this work could not have been adequately completed. The countless occurrences of Jeff's name in the build code comments should be proof enough of his contribution.

Thanks to Scott McMillan of Xilinx for designing the wire database interface of JBits 3 in a manner that supported both our efforts. Perhaps without knowing it, Scott sparked my decision to separate the family and device data, which resulted in much more compact and efficient databases.

Thanks to Eric Keller of Xilinx who was always among the first to respond to my questions. Eric goes out of his way to help people, and I secretly don't think the guy knows how not to be cheerful.

This research was funded in part under DARPA contract DABT63-99-3-0004.

Thanks to the Xilinx and VT Loki teams.

Thanks to my CCM Lab friends; best of luck to all of you.

Thanks to Java for reminding me how much I prefer C++.

Thanks to Greg H. Dow, whom I have never had the pleasure to meet, for allowing mere mortals to gaze at his world-class code. In my book, Greg has no equals.

Thanks to Dr. Ed Green of the Virginia Tech Math Department for taking the time to discuss Gröbner Bases with me.

Thanks to Brad Cathey, principal of Highgate Cross & Cathey, for taking a chance on me.

Thanks to Marty Bell, principal of The DesignSoft Company, for the many great years of coding and friendship.

Thanks to Chuck Treu of Raytheon Systems Company, for his friendship and recommendation, including any fabrications which may have helped guarantee my acceptance at Virginia Tech.

Thanks to Theresa Tagliavia and Dr. Bill Wharton, my academic advisors at IIT and Wheaton College, for seeing me through a most tumultuous and lengthy undergraduate career, and adding further fabrications to my Virginia Tech application. You must have made me sound like a real catch.

Thanks to Dr. Ron Carter of the University of Texas at Arlington, for giving me the right advice even though it meant losing me as a prospective student.

Thanks to Cindy Hopkins and Donna Medley for inviting me to the department's first graduate recruiting weekend, and to Dr. Ferrari, under whose tenure the tradition began. Of the many wonderful experiences that I have had in my life, graduate study in the ECE department at Virginia Tech ranks near the top.

Thanks to Dr. Tront, Dr. Brown, and Dr. Shaw, for your VLSI, Electromagnetic Waves, and Applied Math classes. It's always inspiring to learn from people who excel in their field. And though the reasons for the distinction elude me, to Dr. Tront and Dr. Shaw, I must say that it was a pleasure studying under you, and to Dr. Brown, I must say that it was an honor.

Thanks to my parents, Joe and Marge, and to my family, Bern and Camaleta, Norine and Andrew, Jordan, Jackie, and Blaise.

Thanks to my many church families in Chicago, Miami, Dallas, and Blacksburg.

And thanks to my good friends scattered around the U.S. and the world, particularly Eric and "the girls," Carla, Johannes and Véronique, my BFA friends, Wrong House friends, Raytheon CCA ICT friends, Garage friends, and so many more.

Table of Contents

Table of Contents	vi
List of Figures	xv
List of Tables	xix
1 Introduction	1
1.1 Overview	1
1.2 Objectives	2
1.3 Organization	3
2 Background	4
2.1 Overview	4
2.2 Configurable Devices	4
2.3 Xilinx FPGA Families	5
2.4 Related Work: JBits	6
2.5 Prior Work: WireDB	7
2.6 Glossary	8
3 Source Data	12
3.1 Overview	12

- 3.2 BFD Files 12
 - 3.2.1 BFD Dialects 14
 - 3.2.2 File Structure 14
 - 3.2.3 Tile Declaration 14
 - 3.2.4 Sites Declaration 14
 - 3.2.5 Nodes Declaration 16
 - 3.2.6 Bits Declaration 17
 - 3.2.7 Equation Declarations 18
 - 3.2.8 Arc Ambiguities 20
- 3.3 BXD Files 21
 - 3.3.1 File Structure 21
 - 3.3.2 Tile Declaration 22
 - 3.3.3 Wire Anchoring 23
- 3.4 Tilemap Files 24
 - 3.4.1 File Structure 24
 - 3.4.2 Grid Coordinates 24
- 3.5 XDL Files 25
 - 3.5.1 File Structure 26
 - 3.5.2 Tile Declaration 27
 - 3.5.3 Site Declaration 28
 - 3.5.4 Wire Declaration 29
 - 3.5.5 Pip Declaration 30
- 3.6 .xdlrules Files 30
 - 3.6.1 File Structure 31

3.6.2	Tile Type Declaration	31
3.7	Family Specific Perl Code	32
3.7.1	Virtex	33
3.7.2	Virtex-E	34
3.7.3	Virtex-II	35
4	Database Design	36
4.1	Overview	36
4.2	Objectives	37
4.3	Prior Work	38
4.3.1	JRoute	38
4.3.2	WireDB	38
4.4	Device Sizes	39
4.5	Java Data Types	40
4.6	Data Type Selection	41
4.6.1	Object Overhead Issues	41
4.6.2	Tile Index	43
4.6.3	Wire Index	44
4.6.4	Tilewire	44
4.7	Tiles	44
4.7.1	TileInfo Class	44
4.7.2	WireInfo Class	45
4.7.3	Tiles Class	46
4.8	Remote Resources	48

- 4.8.1 RemoteNode Class 48
- 4.8.2 RemoteArc Class 48
- 4.9 Segments 49
 - 4.9.1 Compacted Segments 50
 - 4.9.2 Compacted Segment Example 52
 - 4.9.3 Segment Lookup Tables 53
 - 4.9.4 Segment Lookup Table Example 54
 - 4.9.5 Segments Class 55
 - 4.9.6 Segment Optimization on Disk 56
- 4.10 Resource Usage 57
 - 4.10.1 WireUsage Class 58
 - 4.10.2 ArcUsage Class 59
 - 4.10.3 GroupUsage Class 59
- 4.11 Database 60
 - 4.11.1 Equations Class 60
 - 4.11.2 BitServer Interface 61
 - 4.11.3 Router Class 62
 - 4.11.4 Tracer Class 62
 - 4.11.5 DB Class 62
 - 4.11.6 ADB Package 62
 - 4.11.7 Database Files 63
- 4.12 JBits Support 65
 - 4.12.1 ADB.JBits Package 66
 - 4.12.2 ADB.[family] Packages 67

5 Database Build	68
5.1 Overview	68
5.2 Preprocessing	70
5.2.1 BXD Preprocessing	70
5.2.2 XDL Preprocessing	70
5.3 Family-Specific Data	73
5.3.1 XDL Processing	73
5.3.2 Preliminary BFD Processing	74
5.3.3 Processing Tile Type Declarations	74
5.3.4 Processing Configuration Bits	75
5.3.5 Processing Logic Sites	75
5.3.6 Remote Node Discussion	75
5.3.7 Remote Node Examples	76
5.3.8 Processing Nodes	79
5.3.9 Preprocessing Equations	80
5.3.10 Remote Arc Discussion	81
5.3.11 Remote Arc Examples	82
5.3.12 Identifying Local and Remote Arcs	85
5.3.13 Reconciling Partially Remote Arcs	86
5.3.14 Searching for Node Synonyms	87
5.3.15 Rewriting XDL Arcs With Local Indexes	88
5.3.16 Arc Classes Discussion	88
5.3.17 Reconciling BFD and XDL Arcs	88
5.3.18 Grouping Configuration Bits	91

- 5.3.19 Reconciling Remote Default Arcs 92
- 5.3.20 Processing Equations 93
- 5.3.21 Writing Java Templates 93
- 5.3.22 Compiling Java Code 93
- 5.3.23 Writing Tile Type Data 94
- 5.3.24 Writing Compiled Code 94
- 5.3.25 Database Compression 95
- 5.4 Bitstream Equations 95
 - 5.4.1 Equation Group Example 96
 - 5.4.2 Equation Conditioning 99
 - 5.4.3 Java Code Generation 100
 - 5.4.4 Failed Equation Inverting Attempts 100
 - 5.4.5 Equation Inverting Example 102
 - 5.4.6 Equation Inverting 104
- 5.5 Device-Specific Data 105
 - 5.5.1 Tilemap Processing 105
 - 5.5.2 Parsing BXD File 106
 - 5.5.3 Adjusting Segments 106
 - 5.5.4 Sorting and Compacting Segments 106
 - 5.5.5 Reconciling Remote Nodes 107
 - 5.5.6 Writing Remote Nodes 109
 - 5.5.7 Writing Extra Tile Segments 109
 - 5.5.8 Writing Compacted Segments 109
 - 5.5.9 Reconciling Remote Arcs 109

5.5.10	Writing Remote Arcs	110
5.5.11	Database Compression	110
5.6	Build Times	111
6	Database Usage	113
6.1	Overview	113
6.2	ADB Initialization	115
6.3	JBits Considerations	116
6.4	ADB as a JBits Router	117
6.4.1	JBits RouterInterface Interface	117
6.4.2	ADB.JBits.Router Class	118
6.4.3	Routing Example	118
6.5	ADB as a JBits Wire Database	120
6.5.1	JBits Lookup Class	121
6.5.2	JBits Wire Class	121
6.5.3	ADB Lookup classes	123
6.5.4	ADB Wire Classes	123
6.5.5	Wire Database Example	125
6.6	ADB as a Standalone Wire Database	125
6.6.1	HelloWorld Example	126
6.6.2	HelloSinks Example	126
6.6.3	DB Class	131
6.6.4	ExtendedWireInfo Class	133
6.6.5	TileInfo Class	135

6.6.6	WireInfo Class	135
6.6.7	Growable Array Classes	135
6.6.8	ADB.JBits.DB Class	136
6.6.9	JBitsConverter Class	137
6.7	ADB as an Interactive Browser	137
6.8	ADB Router	139
6.8.1	Related Work	139
6.8.2	The GRAPHSEARCH Algorithm	140
6.8.3	ADB Implementation of GRAPHSEARCH	141
7	Conclusion	143
7.1	Summary	143
7.2	Results	144
7.3	Lessons Learned	147
7.4	Future Work	148
7.4.1	Timing Data	148
7.4.2	Virtex-E Bitstream Support	148
7.4.3	Support for Larger Devices	148
7.4.4	Router Enhancements	149
7.4.5	Other Families and Architectures	149
7.4.6	Other Manufacturers	150
7.5	Conclusion	150
A	Virtex and Virtex-E Routing Heuristics	151
B	Virtex-II Routing Heuristics	152

Bibliography

List of Figures

2.1	Segments crossing tile boundaries	9
2.2	Logic site: Virtex CENTER Slice 0	10
2.3	Virtex CENTER tile	11
3.1	Source data roadmap	13
3.2	BFD tile type declaration	15
3.3	BFD tile declaration	15
3.4	BFD child tile declaration	15
3.5	BFD sites declaration	16
3.6	BFD local nodes declaration	17
3.7	BFD site-referenced remote nodes declaration	17
3.8	BFD relative-offset remote nodes declaration	18
3.9	BFD configuration bit declaration	18
3.10	Simple BFD equation	18
3.11	BFD multiplexer	20
3.12	BXD header declaration	22
3.13	BXD tile declaration	23
3.14	BXD segment extraction	24

3.15	Tilemap declaration	25
3.16	XDL file structure	27
3.17	XDL tile declaration	28
3.18	XDL site declaration	28
3.19	XDL wire declarations	29
3.20	XDL pip declarations	30
3.21	.xdlrules file structure	32
4.1	Database components	37
4.2	java.lang.Integer test code	42
4.3	Tilewire bit usage	44
4.4	TileInfo class	45
4.5	WireInfo class	45
4.6	Tiles class	47
4.7	RemoteNode class	48
4.8	RemoteArc class	49
4.9	Virtex Single segment	51
4.10	Segment lookup entry bit usage	54
4.11	Segments class	55
4.12	WireUsage class	58
4.13	ArcUsage class	59
4.14	GroupUsage class	60
4.15	BitServer interface	61
4.16	DB class	63

4.17	ADB Package	64
5.1	Build process roadmap	69
5.2	Standard BXD	71
5.3	Preprocessed BXD	71
5.4	Family minsets	72
5.5	Virtex CENTER tile type fragment	77
5.6	Virtex BRAM_TOP tile type fragment	78
5.7	Virtex CLKT tile type fragment	79
5.8	Regular BFD equations	80
5.9	Preprocessed BFD equations	81
5.10	Virtex BRAM_TOP tile type fragment	82
5.11	Virtex CLKT tile type fragment	83
5.12	Sample BFD equation group (raw form)	96
5.13	Sample BFD equation group (internal representation)	96
5.14	Sample BFD equation group (Java implementation)	97
5.15	Sample inverted equation group (raw form)	97
5.16	Sample inverted equation group (enhanced form)	97
5.17	Sample inverted equation group (Java implementation)	98
6.1	ADB as a JBits router	114
6.2	ADB as a JBits wire database	114
6.3	ADB as a standalone database	114
6.4	com.xilinx.JBits.ArchIndependent.RouterInterface interface	118
6.5	Basic routing example	119

6.6	com.xilinx.JBits.ArchIndependent.Lookup abstract class	121
6.7	com.xilinx.JBits.ArchIndependent.Wire abstract class	122
6.8	ADB HelloWorld program	127
6.9	HelloWorld program output	128
6.10	ADB HelloSinks program	129
6.11	HelloSinks program output	130

List of Tables

4.1	Virtex dimensions	39
4.2	Virtex-E dimensions	39
4.3	Virtex-II dimensions	39
4.4	Java primitive data types	40
4.5	java.lang.Integer overhead	42
4.6	Virtex segment compaction	52
4.7	Virtex-E segment compaction	52
4.8	Virtex-II segment compaction	52
5.1	Family database compression	95
5.2	Equation group compression	99
5.3	Virtex database compression	111
5.4	Virtex-E database compression	111
5.5	Virtex-II database compression	111
5.6	Database build times	112
6.1	Virtex database initialization	116
6.2	Virtex-E database initialization	116
6.3	Virtex-II database initialization	116

7.1	Database comparison	145
7.2	Virtex raw database performance	146
7.3	Virtex-E raw database performance	146
7.4	Virtex-II raw database performance	146
A.1	Virtex and Virtex-E resource costs	151
B.1	Virtex-II resource costs	152

Chapter 1

Introduction

1.1 Overview

Modern Field-Programmable Gate Arrays (FPGAs) have matured beyond their origins as small configurable chips, suitable only for implementing glue logic, to truly large devices, capable of implementing entire System-on-Chip (SOC) designs. And though earlier FPGAs were frequently used as stepping-stones, to prototype designs before implementing them in Application-Specific Integrated Circuits (ASICs), a growing number of designs include FPGAs in the final product.

The many benefits of FPGAs, including cost-effectiveness, design flexibility, reusability, and rapid development and debug cycles, are well known. But while development cycles are indeed fast compared to traditional ASIC development, the more unique features of FPGAs are forcing new standards of measure. The potential for dynamic and/or partial reconfigurability requires cycle times on the order of minutes or seconds, rather than days or months, and this in turn requires supporting tools.

One such tool is JBits, a system which operates directly on Xilinx configuration bitstreams, and bypasses the regular tool chain entirely. Naturally there is a tradeoff between the sophisticated and comprehensive mainstream tools, and the faster but less elaborate JBits. Part of that tradeoff is to recognize and support the grid of highly regular Configurable Logic Block tiles, but to ignore or only provide reduced support for the rest of the device. One of the biggest casualties of that approach is the device wiring and connectivity, particularly around the edges of the device and around any of the less common tiles. Nevertheless, JBits provides low-level access and control of devices, which is very appealing to some [1], and substantial work has been done to expand its capabilities.

This thesis presents databases for Xilinx Virtex, Virtex-E, and Virtex-II families, which permit comprehensive support for all wiring and connections in the entire device. The database files and

associated API presented here are collectively known as the *Alternate Wire Database* or simply as *ADB*. The term *alternate* is used in recognition of the fact that JBits has its own internal wire database. By contrast ADB is an external tool which can interface with JBits to provide wiring information and routing or tracing services. ADB can also operate in standalone mode, providing information and services to client tools, such as browsers, routers, tracers, and placement tools. ADB should be helpful to JBits users who require exhaustive wiring support, or to researchers who might not normally have access to wiring data for real-world commercial FPGAs.

In addition to the databases and the supporting API, this work also includes scripts to build the database files. The build process is non-trivial, and adding support for a new family generally requires considerable familiarity with the family, as well as some assistance from Xilinx. ADB currently supports the Virtex, Virtex-E, and Virtex-II families, but has the potential to support Spartan-II, Spartan-IIE, and Virtex-II Pro without too much effort. With a little more effort and some additional modifications, ADB could support XC4000, Spartan, and Spartan-XL. And finally, ADB has the potential to support future families as they are introduced.

1.2 Objectives

ADB has a number of design objectives, inherited and expanded from prior work [2]:

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

Each of these objectives serves an important purpose, beginning with the wiring coverage. The latest version of JBits has considerably better wiring coverage than previous versions did, and much has been done to automate the extraction of wiring information, but the problem is still a difficult one, and the JBits wiring coverage is not complete. In this matter, JBits reasonably chooses to sacrifice some coverage in exchange for simplicity and compactness. ADB aims for exhaustive coverage from the beginning, and does not compromise that goal.

The runtime and initialization performance are obviously significant in any system which aims to encourage, rather than discourage, dynamic reconfiguration. And the data overhead in memory and on disk can determine whether the system runs efficiently, or causes excessive virtual memory swapping, or fails to run at all. Furthermore, even if a tool can fit into memory, it does little good

if there is no space left for its clients. Although this may sound theoretical, the quantities of data involved for the larger devices make it a serious consideration, and related work shows that 1 GB of physical RAM was sometimes insufficient to build databases even for *medium-sized* devices. The memory overhead becomes an even greater consideration in embedded systems which may need to perform their own reconfiguration [3], particularly when an FPGA is reconfiguring itself [4].

Finally, while ADB provides exhaustive wiring support, it does not support logic configuration, or reusable cores, or any number of other JBits functions, and is therefore incomplete by itself. Some users may in fact choose to use ADB apart from JBits, but most will probably require both. Furthermore, ADB by itself can neither read nor write configuration bitstreams, so it generally depends on JBits for access, except in cases where a user is simply perusing a device or developing abstract routing or placement tools.

1.3 Organization

This thesis is organized into four major chapters, with additional background and conclusion chapters:

Chapter 2 - Background: *“What are we talking about anyway?”*

Reviews FPGAs, JBits, and prior work, and provides a glossary of terms.

Chapter 3 - Source Data: *“What do we have to work with?”*

Discusses all of the raw data from which the databases are built.

Chapter 4 - Database Design: *“Where are we trying to go with it?”*

Discusses the design of the database, including device modeling, data structures, and coding decisions.

Chapter 5 - Database Build: *“How do we get from here to there?”*

Discusses the steps necessary to convert the raw source data into completed databases.

Chapter 6 - Database Usage: *“How do we use this thing?”*

Discusses interfaces and considerations for using ADB with JBits or in standalone mode.

Chapter 7 - Conclusion: *“But does it work?”*

Summarizes results and discusses future work.

Chapter 2

Background

2.1 Overview

This chapter addresses background topics which pertain to the thesis, including configurable semiconductor devices in general, and Xilinx FPGAs in particular. It also addresses related work, including JBits, JRoute, and WireDB. The chapter concludes with a glossary of terms used in this thesis.

2.2 Configurable Devices

Unlike conventional semiconductor devices, configurable devices can be customized in some manner, and therefore optimized for a particular problem or situation. In some cases, the usefulness of such devices stems from the possibility of dynamically changing part or all of the device configuration while it is in operation, and there are examples of such devices outperforming even specially fabricated Application-Specific Integrated Circuits (ASICs) [5].

Commercially-available devices which can be configured after they have been fabricated, are generically known as Programmable Logic Devices (PLDs) or Field-Programmable Gate Arrays (FPGAs). These devices are often mentioned in contrast to ASICs, which cannot be configured after they have been fabricated. But the distinctions are becoming increasingly blurred, with the introduction of ASICs and CPUs which include blocks of configurable logic [6, 7], or FPGAs which include hard CPU cores [8]. The one common thread seems to be a recognition of the many benefits that configurable logic can provide.

Configurable devices can be divided into SRAM-based and non-SRAM-based categories [9, 10, 11]. SRAM-based devices are volatile and must be configured whenever power is applied, but that allows

them to change configurations. Non-SRAM-based devices are non-volatile and do not need to be configured each time power is applied, and they can in fact only be configured one time. There are also various time, cost, size, and reliability tradeoffs between these categories.

SRAM-based PLDs and FPGAs are available from a variety of manufacturers, including Actel, Altera, Atmel, Lattice, and Xilinx. SRAM-based configurable devices generally support configuration through serial bitstreams, generated by the manufacturer's software tools. Such tools are usually very robust and sophisticated, but they are also usually too slow to facilitate dynamic or runtime reconfiguration.

2.3 Xilinx FPGA Families

Xilinx introduced the first ever FPGA in 1985, aiming “to combine the logic density and versatility of gate arrays with the time-to-market advantages and off-the-shelf availability of user-programmable standard parts” [12]. The first device was called the XC2064 and featured an 8×8 array of gates. By 1994 the product line had expanded to include the XC2000, XC3000, and XC4000 families, and Xilinx had revenues in excess of \$320 million [12]. By 1996 Xilinx had added the XC5200 and XC6200 families and had posted 1995 revenues exceeding \$500 million [13]. Xilinx continues to be a recognized leader in the areas of FPGAs and configurable devices, with the introduction of still newer devices:

Virtex: Introduced in late 1998, based on a 5-layer-metal $0.22 \mu\text{m}$ CMOS process running at 2.5 V, and ranging from 1,728 to 27,648 logic cells [14]. This very successful family has generated over \$1 billion in revenue in record time [15].

Virtex-E: Builds on the Virtex architecture with a 6-layer-metal $0.18 \mu\text{m}$ process running at 1.8 V. Device sizes extend up to 73,008 logic cells, and up to 851,968 bits of block RAM [14].

Virtex-EM: Further extends the Virtex-E block RAM capacity to 1,146,880 bits [16]. *In most internal respects the Virtex-EM family is indistinguishable from the Virtex-E family. Both ADB and the Xilinx Foundation tools include Virtex-EM devices with the Virtex-E family, and any future mentions of Virtex-E in this thesis should be understood to include Virtex-EM devices.*

Virtex-II: Next generation architecture, aimed at communications and digital signal processing applications, based on an 8-layer-metal $0.15 \mu\text{m}$ process running at 1.5 V, with up to 104,832 logic cells, and up to 3,096,576 bits of block RAM [17, 18].

Virtex-II Pro: Further extends the Virtex-II family by including up to four PowerPC 405 processors on a device. The family is based on a 9-layer copper $0.13 \mu\text{m}$ process running at 1.5 V [19].

XC4000: Older FPGA generation, including XC4000, XC4000A, XC4000E, XC4000EX, XC4000H, XC4000L, XC4000XL, XC4000XLA, and XC4000XLV families, with up to 20,102 logic cells, running as low as 2.5 V [20].

Spartan: Low-cost high-volume version of XC4000, with streamlined feature set and up to 1,862 logic cells, running at 5 V [21].

Spartan-XL: Similar to Spartan, with enhanced feature set, running at 3.3 V [21].

Spartan-II: Low-cost high-volume version of Virtex, with streamlined feature set, and up to 5,292 logic cells and 57,344 bits of block RAM, running at 2.5 V [22].

Spartan-III: Low-cost high-volume version of Virtex-E, with streamlined feature set, and up to 6,912 logic cells and 65,536 bits of block RAM, running at 1.8 V [23].

2.4 Related Work: JBits

JBits, which began its life as the Xilinx Bitstream Interface (XBI), is a Java API which provides an interface to Xilinx configuration bitstreams [24]. In many ways JBits sought to provide the same type of bitstream access that was available for the XC6200 family through JERC6K [25]. The original version of JBits targeted the XC4000 architecture, and modeled devices as two-dimensional arrays of Configurable Logic Blocks (CLBs), also known as CENTER tiles. Subsequent versions targeted Virtex, and added support for Input Output Block (IOB) tiles and Block RAM (BRAM) tiles. The upcoming JBits 3 targets both Virtex and Virtex-II, and does much to automate the generation of internal classes and constants.

Recent versions of JBits provide a rich set of development and debugging tools, and also support Run-Time Parameterizable (RTP) cores, sometimes known as Core Templates. The user can control the placement and the configuration of RTP cores, and can write their own cores, including some fairly elaborate ones [26, 27]. JBits is the subject of many conference papers, and the interested reader is encouraged to peruse these papers.

One of the most important features of JBits, from the perspective of ADB, is its support for reading and writing full or partial configuration bitstreams [28]. Although parts of the Virtex bitstream format are documented in Xilinx Application Note 151 [29], the information provided is primarily intended for reading or manipulating BRAM and Look-Up Table (LUT) contents, and is drastically insufficient for general logic and routing configuration. ADB therefore relies on JBits to properly read, parse, and write bitstreams. This is true whether ADB is used in conjunction with JBits or in standalone mode, with the notable exception of Virtex-E which JBits does not support.

JRoute is the router included with JBits [30]. At its inception, it only included support for CENTER tiles, but was later expanded to support IOB and BRAM tiles.¹ The version of JRoute included

¹Although the CENTER, IOB, and BRAM tiles only represent a small fraction of the *tile types*, they account for the vast majority of the *tiles* in a device.

with JBits 3 is based on the more comprehensive internal wire database, which may allow JRoute to provide support for some of the special purpose tiles. JRoute currently includes an auto-router and a template router, and while the latter requires more guidance from the user, it can provide very good results for parallel or regular structures.

JBits also provides two key interfaces which facilitate routing collaboration with ADB. Although JBits includes JRoute to provide routing services for users, it also exposes part of the routing interface, which allows users to select the ADB router in place of JRoute. JBits 3 also includes an internal wire database and exposes its interface, which allows users to select ADB in place of the internal wire database. The interfacing between ADB and JBits is based on a preliminary version of JBits 3, but the interaction between the two products is expected to be finalized before JBits 3 is released.

2.5 Prior Work: WireDB

The present work was preceded by WireDB, written by Athanas. WireDB was developed and used internally within the Configurable Computing Lab (CCM Lab) at Virginia Tech, with the intention of providing comprehensive wiring databases for all major Xilinx FPGA families and interfacing with JBits.

By all initial appearances, WireDB was mature enough to be used as the basis for other tools within the CCM Lab. Athanas wrote a connectivity browser tool *CBrowser*, to demonstrate the use of the database and to provide some debugging support. Goetz began work on an EDIF netlist. And Steiner developed a routing interface between WireDB and JBits.

A number of important lessons were learned from the use of WireDB. Firstly, it became apparent that certain Java Virtual Machines (JVMs) were *excessively* slow in deserializing Java classes.² Secondly, the serialized data stored on disk was very large even in its compressed state. And thirdly, the source data did not provide sufficient information about the directions in which wires could connect to one another.

While the performance and size issues were bothersome, the directionality problem was insurmountable because it made the entire database unusable. Correspondence with Xilinx confirmed that the source data was incomplete for the purposes in which WireDB needed to use it. This factor more than any other is what prompted work on ADB. Although initial work focused on ways to augment the source data with arc direction information, that effort quickly expanded to include alternate representations of the data which might decrease size and increase performance. While ADB retains the objectives of WireDB, it does not share any of its architecture or code base.

²The Sun 1.2.2 JVM takes over four times longer to initialize WireDB databases than the Sun 1.3.0 JVM does.

2.6 Glossary

Abstract Tile Type A tile type which is never instantiated in any family device. Abstract tile types usually serve as parents for other tile types. Converse of *concrete tile type*.

Arc A connection between two nodes. An arc connecting two nodes is the mathematical analogue of a switch connecting two wires. *This term is borrowed from graph theory.* May be *bidirectional* or *directed* (unidirectional). See also *pip* and *directed arc*.

Bit A configuration bit stored in an SRAM memory.

Concrete Tile Type A tile type which is instantiated in at least one family device. May also be called *non-abstract tile type*. Converse of *abstract tile type*.

Device An individual member of an FPGA family. *Examples: XCV50, XCV50E, XC2V40.*

Directed Arc A unidirectional connection between two nodes. A directed arc from *nodeA* to *nodeB* means that *nodeA* drives *nodeB* but does not mean that *nodeB* drives *nodeA*. Note that a regular arc can be decomposed into two directed arcs pointing in opposite directions.

Fake Arc An arc that always exists and requires no configuration bits. This is not the same as a *segment*. Wires on a segment form a single electrical node. Wires connected by fake arcs are instead connected by unidirectional buffers.

Family A collection of devices sharing a common architecture. *Examples: Virtex, Virtex-E, Virtex-II.*

Minset A subset of devices which include every concrete tile type in a family. While a minset is guaranteed to be a *small* necessary subset of devices, it is not guaranteed to be the *smallest* possible subset. *Example: the XCV50E and XCV405E devices form a minset for the Virtex-E family because they include every tile type that occurs in the entire family.*

Node A wire. *Node* and *wire* are used synonymously throughout this thesis. *This term is borrowed from graph theory.*

Package The protective container enclosing a die. The term is only used here to acknowledge that a device may be available in more than one package. The package determines physical dimensions and thermal characteristics, but also determines which IO pads are connected to external leads. *Examples: CS144, BG352, FG680.*

PIP A Programmable Interconnect Point; commonly written *pip*. A pip is a programmable connection between wires, controlled by one or more configuration bits. When a connection is made through a pip, an arc is said to exist from one node to the other. This thesis generally relaxes the terminology and uses the term *arc* even when *pip* is more correct.

Routethrough A connection between two wires achieved by routing the signal through a logic site. *i.e. the connection is achieved with logic resources instead of regular routing resources.*

Segment A single electrical node that may span multiple tiles. A segment consists of the set of physically connected wires in the various tiles that it spans. Figure 2.1 shows examples of segments which cross tile boundaries. Each segment is comprised of wires which exist entirely within tile boundaries. The segment itself does not have a name, but is instead identified by the collective names of its wires.

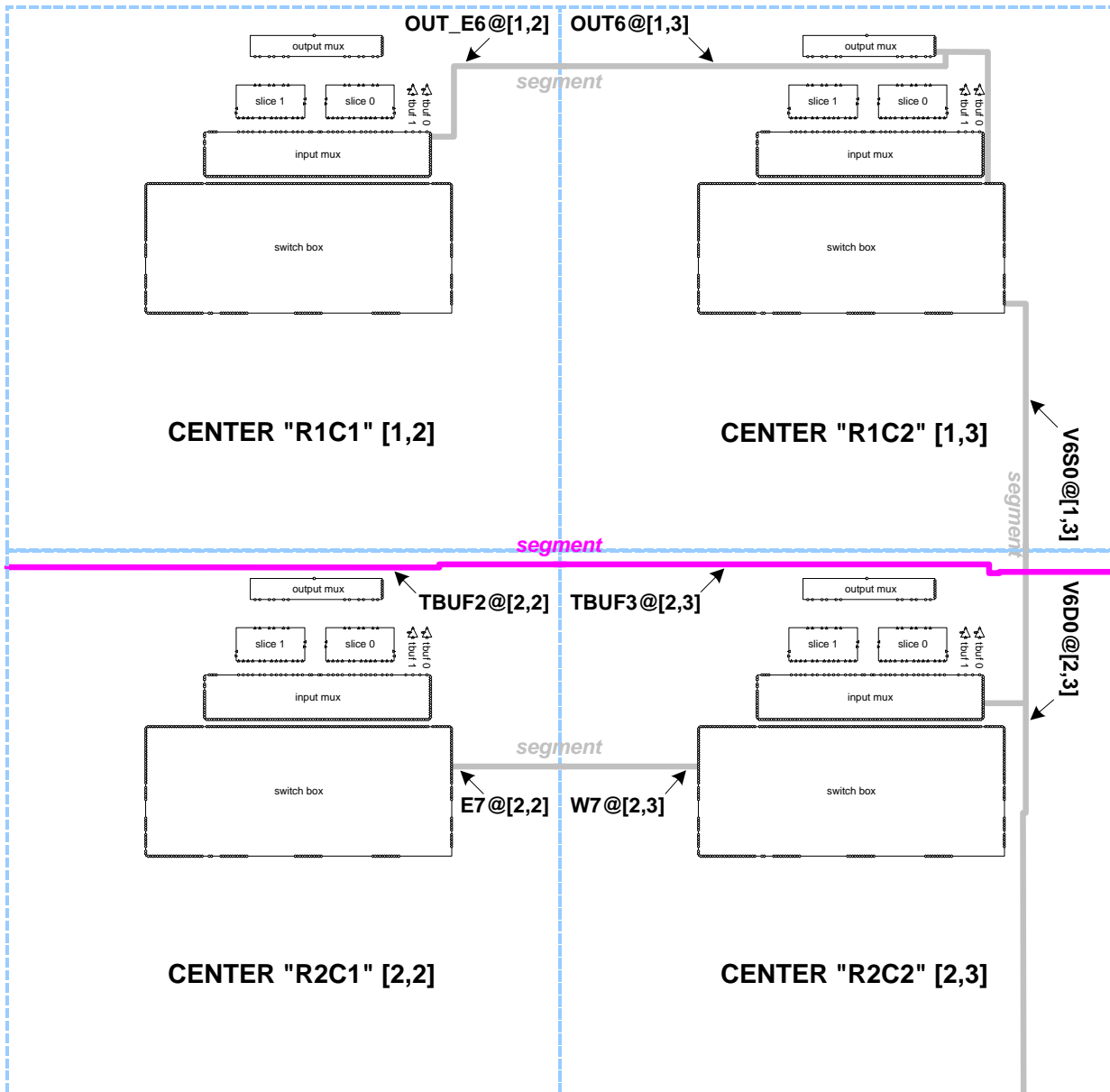


Figure 2.1: Segments crossing tile boundaries

Sink A receiver. This may be a site input or the final node of a directed arc. See also *source*.

Site A resource containing logic or buffers. Figure 2.2 shows the external view of a site.

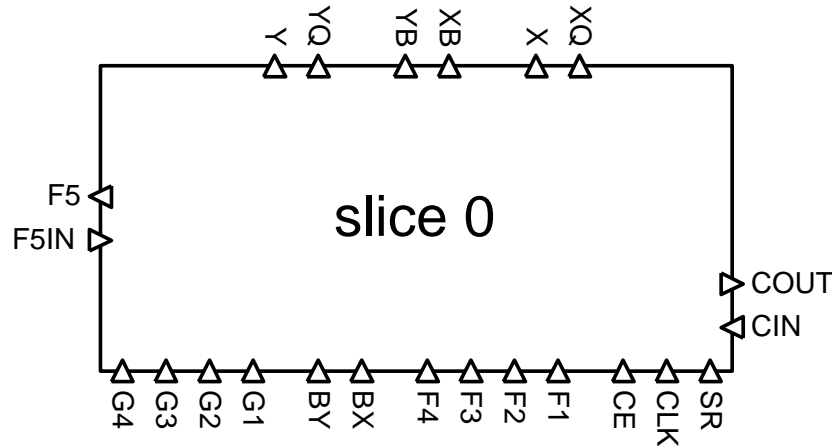


Figure 2.2: Logic site: Virtex CENTER Slice 0

Source A driver. This may be a site output or the initial node of a directed arc. See also *sink*.

Tile An element within the two-dimensional FPGA grid, typically containing logic and routing resources. Tiles are identified by grid coordinates and type. Figure 2.3 shows a Virtex CENTER tile.

Tile Type The classification of a tile or set of tiles according to function or resources. A tile type may be abstract, meaning that it is never instantiated within a family. By contrast, *tiles* are always concrete, since they by definition make up the device grid. *Examples: CENTER, LEFT, BRAM, CLKT.*

Tilewire A compound data type consisting of a wire index and a tile index. A tilewire uniquely identifies a wire in an actual device.

Wire An electrical node, or a portion thereof, contained entirely inside a single tile. *Node* and *wire* are used synonymously throughout this thesis.

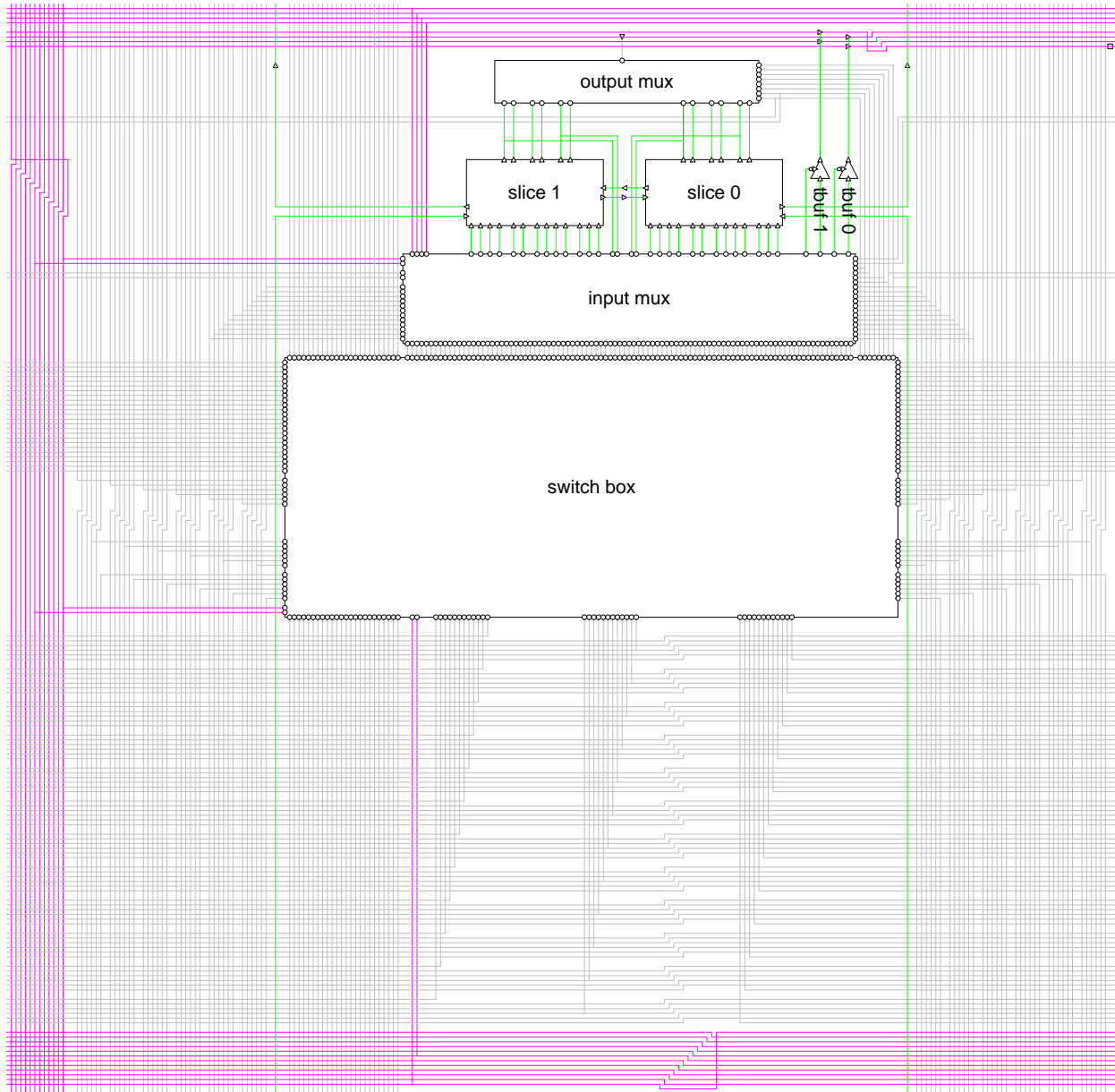


Figure 2.3: Virtex CENTER tile

Chapter 3

Source Data

3.1 Overview

The wire databases presented in this thesis are derived and generated almost exclusively from existing Xilinx data files. These data files collectively constitute the source data.

Because some of these files contain internal and proprietary Xilinx information, the examples included here deliberately present only fragments of the code. These examples will hopefully communicate the important aspects of the data without compromising Xilinx’s intellectual property.

The final wire databases need to understand the tiles contained in each device, the wire and arc resources contained in each type of tile, how wire resources connect between neighboring tiles, and how to establish or infer connections with the bitstream configuration data.

The tile map and intertile connections are derived from BXD files for each device. The wire resources, arc types, and configuration data are derived from BFD and XDL files for each device family. Figure 3.1 shows the main source files and dependencies. The contents and importance of each of these files will be discussed in the following sections.

3.2 BFD Files

BFD (BitFile Description) files are internal and proprietary Xilinx files which map configuration information to bitstream contents. The configuration information consists of all the programmable resources in a device family, including routing resources, logic configuration, and block RAM resources. This configuration information is broken down according to tile type, and is mapped to configuration bits through sets of equations. These equations are expressed in BFD syntax and are commonly referred to as BFD equations.

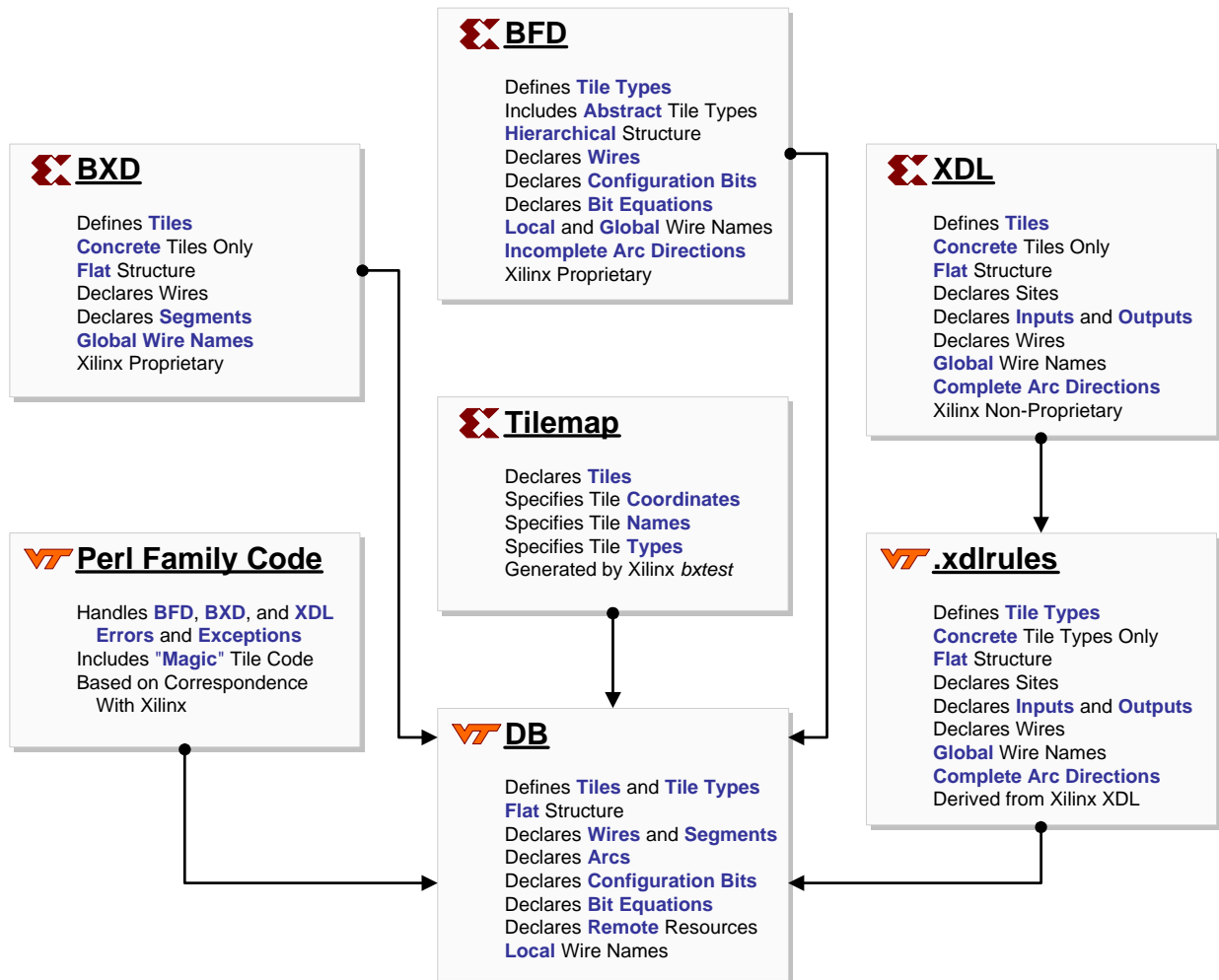


Figure 3.1: Source data roadmap

Each FPGA family typically has its own BFD file, able to accommodate the entire range of devices in the family. This means that a BFD file must define resources and configuration bits for all tile types which occur in the family, even though not all tile types will necessarily be present in all devices in the family. In addition the BFD file may define abstract tile types which never occur in any device in the family, but which are used hierarchically as parents of other tile types. The use of abstract tile types in BFD files increases the maintainability of the files, decreases their sizes, and allows them to more accurately match the device models.

BFD files are normally included with Xilinx software tools in a binary format suitable only for tool use. The BFD files used here are original text files from which the binary versions are generated.

3.2.1 BFD Dialects

BFD files exhibit significant dialect differences between XC4000¹ class and Virtex² class architectures. The work presented here supports the Virtex architectures, but would need to be expanded in order to support XC4000 architectures.

In addition to the major differences mentioned above, usage and style differences also exist between most members of the Virtex architectures. Some of these are differences in BFD style, while others are differences in BXD or XDL data dependencies.

3.2.2 File Structure

The BFD file for a device family consists of tile type definitions for each tile type that may occur within the family. In many cases, additional tile types are defined to serve as parents of other tile types. The general form of a tile type definition is shown in Figure 3.2. The ellipses in the code fragment serve as placeholders for the content of the declarations. The major sections that make up the tile type declaration are the *Sites*, *Nodes*, *ram*, and equation declarations. Except for the *Sites* section, these sections may occur multiple times, but each is optional. In particular, a tile type that inherits from another tile type only needs to add or override the sections that differ.

3.2.3 Tile Declaration

Figure 3.3 shows a sample *Tile* declaration. The tile type is named GCLKT and includes 18 rows by 1 column of configuration bits (indicated by `rows=18 cols=1`). The grid rows and grid columns (indicated by `grows=2 gcols=0`) are not of interest here.

Figure 3.4 shows a sample child tile declaration. The tile type LBRAM inherits all the attributes of parent tile type BRAM (indicated by "LBRAM:BRAM"). It is assumed that the parent tile type BRAM has already been declared. Child tile type LBRAM may redeclare sections to override parent tile type BRAM or may declare new sections as needed.

3.2.4 Sites Declaration

A *site* represents a logic block within the device. Typical examples of sites include *slices*, *IOBs*, *DLLs*, and *BRAMs*. It is common to think of IOBs or BRAMs as tile types, but in the BFD representation, they are in fact logic sites that occur within similarly named tile types.

¹XC4000, XC4000E, XC4000EX, XC4000L, XC4000XL, XC4000XLA, XC4000XV, Spartan, and Spartan-XL

²Virtex, Virtex-E, Virtex-EM, Virtex-II, Virtex-II Pro, Spartan-II, and Spartan-IIe

```

1 # tile type declaration
2 Tile ... {
3     # sites declaration
4     Sites {
5         :
6     }
7     # nodes declaration
8     Nodes ... {
9         :
10    }
11    # configuration bits declaration
12    ram ... {
13        :
14    }
15    # equations declaration
16    :
17 }

```

Figure 3.2: BFD tile type declaration

```

1 # tile type GCLKT
2 Tile "GCLKT" rows=18 cols=1 grows=2 gcols=0
3 {
4     :
5 }

```

Figure 3.3: BFD tile declaration

```

1 # tile type LBRAM derived from parent tile type BRAM
2 Tile "LBRAM:BRAM"
3 {
4     :
5 }

```

Figure 3.4: BFD child tile declaration

The *Sites* declaration section is used to declare sites that occur within the tile, or to reference sites that occur elsewhere in the device. This section is optional, and reasonably so, since some tile types contain only routing resources and no logic blocks.

Figure 3.5 shows a sample *Sites* declaration. A site may be declared to exist within the current tile type, or may merely be declared to exist somewhere within the device.

```

1 Sites
2 {
3     # sites that occur within this tile type
4     DLLFB1 xoff=-254 yoff=293,
5     DLLFB0 xoff=424 yoff=293,
6     # sites referenced by name
7     DLL2S_SITE sitename=DLL2S,
8     DLL3S_SITE sitename=DLL3S,
9     DLL2P_SITE sitename=DLL2P,
10    DLL3P_SITE sitename=DLL3P,
11    GCLK2      sitename=GCLKPAD2,
12    GCLK3      sitename=GCLKPAD3,
13    GCLKBUF2   sitename=GCLKBUF2,
14    GCLKBUF3   sitename=GCLKBUF3
15 }
```

Figure 3.5: BFD sites declaration

Lines 4 and 5 declare two sites that occur within the current tile type. These sites are declared by name and are given offsets within the tile.

Lines 7 through 14 show sites that are referenced globally within the device. These sites are given local names which in turn map to global names. Global site names uniquely identify a site within a device. In line 7 the name `DLL2S_SITE` is used as a local identifier to refer to the global site `DLL2S`.

3.2.5 Nodes Declaration

A *node* represents a wire within the device, as explained in Section 2.6. Nodes may either be local or remote. Local nodes exist within the current tile type, while remote nodes exist in other tiles.³

Figure 3.6 shows a local node declaration, identified by the single keyword `Nodes`. Line 4 shows a node with local name `GCLK0` and global name `CLKC_GCLK0`. The local name is used mainly as an abbreviation in the current tile type. The global name allows this node to be cross-referenced with the BXD and XDL data.

Figure 3.7 shows a remote node declaration with nodes referenced to globally unique site `DLL3`. The declaration is identified by the keywords `Nodes Site`. Line 4 shows a node with local name `DLL_CLKPAD0` and a remote global name `CLKT_CLKPAD0`. The global name `CLKT_CLKPAD0` used in the current tile type matches the global name used in the tile that contains site `DLL3`.

³A remote node may exist in another tile of the same type, or in a tile of an entirely different type.

```

1 Nodes
2 {
3     # local nodes
4     GCLK0 = CLKC_GCLK0,
5     GCLK1 = CLKC_GCLK1,
6     GCLK2 = CLKC_GCLK2,
7     GCLK3 = CLKC_GCLK3,
8     HGCLK0 = CLKC_HGCLK0,
9     HGCLK1 = CLKC_HGCLK1,
10    HGCLK2 = CLKC_HGCLK2,
11    HGCLK3 = CLKC_HGCLK3,
12    VGCLK0 = CLKC_VGCLK0,
13    VGCLK1 = CLKC_VGCLK1,
14    VGCLK2 = CLKC_VGCLK2,
15    VGCLK3 = CLKC_VGCLK3
16 }

```

Figure 3.6: BFD local nodes declaration

```

1 Nodes Site DLL3
2 {
3     # nodes referenced from site DLL3
4     DLL_CLKPAD0 = CLKT_CLKPAD0,
5     DLL_CLKPAD1 = CLKT_CLKPAD1,
6     CLKT_CLK2XL = CLKT_CLK2XL,
7     CLKT_CLK2XR = CLKT_CLK2XR,
8     CLKT_GCLKBUF1_IN = CLKT_GCLKBUF3_IN,
9     CLKT_GCLKBUF0_IN = CLKT_GCLKBUF2_IN
10 }

```

Figure 3.7: BFD site-referenced remote nodes declaration

Figure 3.8 shows a remote node declaration with nodes referenced to a relative offset tile one column to the right (indicated by `row=0 col=1`). The declaration is identified by the keywords `Nodes Tile`. Line 4 shows a node with local name `TBUF_R1` and a remote global name `TBUF3`. The global name `TBUF3` used in the current tile type matches the global name used in the tile one column to the right.

3.2.6 Bits Declaration

Configuration bits represent SRAM resources which control the logic or routing in a tile type. In general each of the configuration bits specified in the tile declaration is either named or declared to be unused. These configuration bit names are subsequently referenced by the BFD equations.

```

1 Nodes Tile row=0 col=1
2 {
3     # nodes referenced from the tile 1 column to the right
4     TBUF3_R1 = TBUF3,
5     TBUF3_STUB_R1 = TBUF_STUB3
6 }

```

Figure 3.8: BFD relative-offset remote nodes declaration

Figure 3.9 shows the configuration bits declaration for configuration row 2. The declaration is identified by the sequence `ram row=2`. Lines 4 through 6 show eight consecutive configuration bit names which will be numbered $([2, 0], [2, 1], \dots, [2, 7])$. Additional configuration bits declarations would be necessary to declare the remaining bits for the current tile type.

```

1 # configuration bits declaration for row 2
2 ram row=2
3 {
4     I251.I200.I14to1_8_, I251.I200.I14to1_9_, I251.I200.I14to1_10_,
5     I251.I200.I14to1_11_, I251.I200.I14to1_12_, I251.I200.I14to1_13_,
6     I251.I200.I14to1_14_, I251.I200.I14to1_15_
7 }

```

Figure 3.9: BFD configuration bit declaration

It is worth noting that configuration bits may be enumerated by row or by column. In certain cases tile types enumerate some bits by row and other bits by column. As long as each bit is declared exactly one time, this is entirely acceptable.

3.2.7 Equation Declarations

The BFD equations determine how the configuration bits are programmed. In its most basic form a BFD equation consists of an assignment, where the left-hand side of the equation is a bit name, and the right-hand side is an expression that must be evaluated. Figure 3.10 shows a simple BFD equation.

```

1 # equation defining named bit Iimux.Iclk_0_.I202
2 Iimux.Iclk_0_.I202 = !( arcval (GCLK2, CLK0) );

```

Figure 3.10: Simple BFD equation

Right-hand side expressions can consist of the following:

- Logical operators `||` (inclusive-OR), `&&` (AND), and `!` (NOT)
- Boolean values `TRUE` and `FALSE`
- BFD functions:
 - `arcinv(node1,node2):`**
true if *node1* and *node2* are connected through an inverter
 - `arcinv(node1,node2,TRUE):`**
true if *node1* drives *node2* through an inverter
 - `arcval(node1,node2):`**
true if *node1* and *node2* are connected (regular arc)
 - `arcval(node1,node2,TRUE):`**
true if a *node1* drives *node2* (directed arc)
 - `cmdarg(option,value):`**
true if *option* was set to *value* through a command line argument
 - `comp(sitename):`**
true if site *sitename* is used in the design
 - `config(sitename,primname,config):`**
true if attribute *primname* of site *sitename* is set to value *config*
 - `memory(sitename,bank,bit):`**
true if bit *bit* in bank *bank* of site *sitename* is 1
 - `nodeused(node):`**
true if node *node* is used in the design
 - `noderouted(node):`**
true if node *node* belongs to a routed net in the design
 - `pminfo(sitename,config,bit):`**
callback function linked to internal Xilinx code
 - `pminfo(sitename,config):`**
callback function linked to internal Xilinx code
 - `pminfo(parameter):`**
callback function linked to internal Xilinx code
 - `readback(sitename,node):`**
true if flip-flop output *node* is 1 during readback
 - `sitetype(sitename,sitetype):`**
true if site *sitename* is of type *sitetype*
 - `tiletype(tilename,row,col):`**
true if tile at offset [*row,col*] is of type *tilename*

Many of these functions relate to logic configurations or memory values rather than routing. Other functions are never used in any of the XC4000 or Virtex class BFD files. Consequently only a subset of these functions is supported by ADB.

In many cases, bits are logically grouped in order to implement look-up tables or multiplexers. Figure 3.11 shows a simple 6-to-1 multiplexer. This multiplexer has inputs V6A1, V6B1, V6C1, V6D1, V6M1, and V6N1, and output I3. It is controlled by bits I76.I63.I1, I76.I63.I7, I76.I63.I8, and I76.I63.I11.

```

1 # I3 multiplexer
2 I76.I63.I1 = arcval (V6N1, I3) || arcval (V6M1, I3);
3 I76.I63.I11 = arcval (V6N1, I3) || arcval (V6A1, I3) || arcval (V6B1, I3);
4 I76.I63.I7 = arcval (V6A1, I3) || arcval (V6C1, I3);
5 I76.I63.I8 = arcval (V6B1, I3) || arcval (V6D1, I3);

```

Figure 3.11: BFD multiplexer

3.2.8 Arc Ambiguities

A closer look at the code in Figure 3.11 reveals the conspicuous absence of directed arcs. For example in line 2, bit I76.I63.I1 is set if V6N1 connects to I3 or if V6M1 connects to I3, regardless of the directions in which those connections are made. By all appearances the code seems to suggest that these connections are bidirectional. From a routing perspective one should then be able to connect from I3 to V6N1 or from I3 to V6M1, but in fact I3 cannot drive either of these nodes.

This directional ambiguity is the primary reason for the demise of WireDB. Unlike the Xilinx tools which only use the BFD data to generate bitstreams, WireDB sought to use the BFD data to determine how nodes could connect to each other within a device.

WireDB originally assumed that all *regular arcs* were bidirectional, which is indeed a reasonable assumption. When the directionality problem surfaced, WireDB was modified to assume that regular arcs were in fact directed arcs, with the first node driving the second one. This approach also failed for two significant reasons. Firstly because some regular arcs really do identify bidirectional connections.⁴ And secondly because some unidirectional connections are listed “backwards” in violation of the rule assumed above.⁵ The unwelcome conclusion was that the BFD data alone could not reveal valid arc directions within devices.

If the directionality issue is temporarily set aside, it may suddenly become nonobvious that these equations in fact constitute a multiplexer. The comment on line 1 was inserted for clarity and does

⁴For example connections between two *singles* in the Virtex family.

⁵For example *singles* appearing to drive *hexes* in the Virtex family.

not appear in the original code. The argument that all of the *arcval()* functions appear to have I3 as a sink falls apart when the arc directions are no longer known.

Nevertheless these equations do constitute a multiplexer, and they can indeed be grouped on the basis of the second node, in this case I3. Though nothing in the BFD syntax requires this behavior, the Virtex, Virtex-E, and Virtex-II families follow this rule with no known exceptions. In fact BFD files deliberately reverse the direction of some arc nodes in order to prevent incorrect multiplexer groupings. This justifies the “backwards” arcs described above.

3.3 BXD Files

BXD files are internal and proprietary Xilinx files which define all tile instances, wires, and logic sites within each specific device. Because a BXD file represents an actual device, it never includes any of the abstract tile types which may appear in the family’s BFD file.

Each tile in a device is identified in the BXD file by tile name and tile type, and includes a list of contained wires and sites. The tile name and tile type information makes it possible to derive a two-dimensional tile map of the device. While the BFD information indicates what tile types may occur in family devices, the BXD information specifies each tile type that actually does occur.

The BXD information also indicates which wires in seemingly different tiles are actually part of a common net. These intertile connections do not represent configurable routing resources at work, but rather indicate how wires in adjacent or nearby tiles abut and connect with one another. Where the BFD information provides a very useful tile type abstraction, the BXD information specifies how real tiles work together in the physical device.

BXD files are normally included with Xilinx software tools in a binary format suitable only for tool use. The BXD files used here are extracted text versions of the originals. The extraction is performed by an internal Xilinx tool named *bxttest*. The extraction process is scripted and generates two different text files. The first file is a *.dbxd* file (dump BXD) which describes all intertile connections. The second file is a *.map* file which lists the tile name, tile type, and coordinates for each tile in the device.

All wire names used in BXD files are global names.

3.3.1 File Structure

A BXD file generally consists of a two-level tree of `BX_INSTANCE` entries. The top-level instance represents the entire device with all of its tiles, and each child instance represents a tile in the device.

Figure 3.12 shows a typical BXD header. Line 1 begins with keyword `BX_DB` and provides database version information. Lines 2 through 5 show the top-level instance for a sample device. Line 2 begins with keyword `BX_INSTANCE` and declares the top-level instance. The identifiers `top_ass` and `top` simply designate the top-level assembly for this device. The two sets of coordinates `(-6514, -4088)` and `(6744, 4048)` specify the bounding box for all of the device tiles. The next three lines indicated the number of wires, the number of sites, and the number of child instances declared for the device.

```

1 | BX_DB:A:3
2 | BX_INSTANCE: "top_ass" "top" (-6514, -4088) (6744, 4048)
3 | 0 # wire count
4 | 0 # site count
5 | 589 # instance count

```

Figure 3.12: BXD header declaration

Since this instance is the top-level declaration for the device, it does not declare any wires or sites, so the number of wires and the number of sites are both zero. Had either number been non-zero, the specified number of wire or site declarations would have appeared on lines immediately following the appropriate counts. However on line 5 this instance declares 589 child instances, which represent each of the 589 tiles in this device. Since the 589 tiles are child instances of the top-level assembly, they technically belong inside this declaration, but were omitted here for clarity.

3.3.2 Tile Declaration

Figure 3.13 shows a BXD tile declaration. Line 1 begins with keyword `BX_INSTANCE` and specifies the tile name as `BC1` and the tile type as `BOT`. The coordinates `(-5710, -4088)` and `(-5236, -3752)` represent the bounding box for this tile. Line 2 specifies 283 wires, whose declarations appear on the immediately succeeding lines. Line 3 begins with keyword `BX_WIRE` and declares a wire named `BOT_CLK0` and anchored at coordinates `(-5466, -4040)`. The wire names declared here correspond to the global wire names used in BFD files.

After all 283 wires have been declared, line 13 specifies 4 sites for this tile. The sites are declared on lines 14 through 17. Line 14 begins with keyword `BX_SITE` and declares a site of type `BOT_EMPTYIOB` at coordinates `(-5562, -4045)`. After all 4 sites have been declared, line 18 specifies 0 subtiles, because tiles are not allowed to have children.

```

1 BX_INSTANCE: "BC1" "BOT" (-5710, -4088) (-5236, -3752)
2 283 # wire count
3 BX_WIRE: "BOT_CLK0" (-5466, -4040)
4 BX_WIRE: "BOT_CLK1" (-5502, -4040)
5 BX_WIRE: "BOT_CLK2" (-5538, -4040)
6 BX_WIRE: "BOT_CLK3" (-5574, -4040)
7 BX_WIRE: "BOT_CLKFB" (10, -3950)
8 BX_WIRE: "BOT_CLKIN" (14, -3950)
9
10
11
12
13
14
15
16
17
18

```

Figure 3.13: BXD tile declaration

3.3.3 Wire Anchoring

As described above, wire declarations consist of a wire name and of anchor coordinates. These anchor coordinates are very important because they identify connected wire pieces that span tiles. Following the WireDB convention, the set of connected wires in one or more tiles is called a *segment*. This terminology may be confusing at first since it appears to be backwards, however it is strictly adhered to in the remainder of this thesis.

It is important to note that a segment consists of individual wires that are physically and permanently connected inside the device. This differs significantly from wires that are connected through configurable resources to form a logical net. A segment may well connect to other segments through configurable resources, but the segment itself is a single electrical node.

Figure 3.14 takes liberties with the format of BXD files in order to show the tile wires that make up a particular segment. The `BX_INSTANCE` lines identify the tiles which this segment spans. Each tile has its own tile name, type, and coordinates. The `BX_WIRE` lines show the wire names that the segment takes on in each of the tiles that it spans, and also show the wire's anchor coordinates. The fact that each of these wires share the anchor coordinates (-5947, -696) is what identifies them as belonging to the same segment.

```

1 | BX_INSTANCE: "LBRAMR12" "LBRAM" (-6184, -1896) (-5710, -40)
2 | BX_WIRE:    "BRAM_EB4" (-5947, -696)
  | :
  | :
3 | BX_INSTANCE: "LR10" "LEFT" (-6514, -968) (-6184, -504)
4 | BX_WIRE:    "LEFT_E4" (-5947, -696)
  | :
  | :
5 | BX_INSTANCE: "R10C1" "CENTER" (-5710, -968) (-5236, -504)
6 | BX_WIRE:    "W4" (-5947, -696)

```

Figure 3.14: BXD segment extraction

This sample segment spans tiles LBRAMR12, LR10, and R10C1, of types LBRAM, LEFT, and CENTER, respectively. As the segment crosses these tiles, it takes on the wire names BRAM_EB4, LEFT_E4, and W4.

3.4 Tilemap Files

Tilemap files are derived from BXD files by the *bxtest* tool as described earlier. These files identify each tile in the device by row and column grid coordinates, and also identify the name and type of each tile.

3.4.1 File Structure

A tilemap declaration begins with a header line which provides the row and column counts. The remaining lines in the file declare one tile per line. Figure 3.15 shows part of a tilemap file.⁶ Line 1 declares the size of the tile grid to be 19 rows by 31 columns. The tiles consequently take on row coordinates 0 through 18 and column coordinates 0 through 30. Line 2 declares a tile of type UL at row 0 and column 0, named TL.

3.4.2 Grid Coordinates

The BFD and BXD source data files refer to tiles by unique name, by relative offset, or by unique site name. While these files imply the existence of the tiles they refer to, they do not sufficiently describe how the tiles are laid out in the device tile grid.⁷

⁶The `Enter command --` sequence on Line 1 is an artifact of the extraction script, ignored by the remaining tools.

⁷The BXD instance coordinates actually do indicate relative tile positions, but also introduce a few exceptions which are not easily resolved. It is much simpler to refer to the tilemap data since the *bxtest* tool generates it readily.

```

1 Enter command -- Map is 19 x 31
2   Row 0 Column 0 UL      TL
3   Row 0 Column 1 BRAM_TOP LBRAM_TOP
4   Row 0 Column 2 TOP     TC1
5   Row 0 Column 3 TOP     TC2
6   Row 0 Column 4 TOP     TC3
7   :
8   Row 18 Column 25 BOT    BC21
9   Row 18 Column 26 BOT    BC22
10  Row 18 Column 27 BOT    BC23
11  Row 18 Column 28 BOT    BC24
12  Row 18 Column 29 BRAM_BOT RBRAM_BOT
    Row 18 Column 30 LR     BR

```

Figure 3.15: Tilemap declaration

The tilemap file makes it possible to map tile names to grid coordinates. It also makes it possible to look up the name and type of a tile at some relative offset from the current tile coordinates. In short the tilemap file specifies which tiles appear where, and brings some concreteness to the abstractions of the BFD and BXD files.

3.5 XDL Files

XDL (Xilinx Design Language) files are user accessible files which describe physical design information or device resource information. The Xilinx Foundation software includes an *xdl* tool which can convert proprietary NCD (Native Circuit Description) files to XDL files (in `-ncd2xdl` mode) and vice versa (in `-xdl2ncd` mode) [31]. The same *xdl* tool can also generate information about device resources, including routing resources, intertile connection information, and logic configuration resources (in `-report` mode).

*It is the device resource information generated in `-report` mode that is of interest here, but surprisingly the syntax for the `-report` mode is completely different from that of the `-ncd2xdl` and `-xdl2ncd` modes. While the `-ncd2xdl` and `-xdl2ncd` modes use the language described in [31], the `-report` mode output looks very much like EDIF. Nevertheless, even in `-report` mode the *xdl* tool generates files with `.xdl` extensions, so these files will still be called “XDL” files.*

Because XDL files may be generated with intertile connection information included, they theoretically provide a superset of the BXD information, but the connection information was found to be incomplete in the Xilinx 4.1i Foundation tools. In addition, the XDL files consist of text which can approach 2 GB in size for the largest devices. These factors led to a somewhat more conservative use of XDL files.

One benefit of XDL files in this project is their description of all configurable connections between routing resources.⁸ This provides a much needed supplement to the BFD information, which does not include connections that are always on (fake arcs). Without knowledge of these extra connections, many routing resources would remain obscured, and a large number of nets would be entirely unusable.

Another benefit of the XDL files is the fact that they explicitly state arc directions. Once again this supplements the BFD data, which in most cases only provided vague and sometimes incorrect direction hints.

Unlike the family-specific and hierarchical BFD data, the XDL data is device-specific and flat. These two differences significantly increase the difficulty of reconciling one file type to the other.

Because the XDL data is used to supplement and interpret the BFD data, and because the XDL data is device-specific, it is distilled into a more usable form before use. This permits the database build tools to work with files around 500 KB in size instead of files thousands of times larger.

3.5.1 File Structure

Figure 3.16 shows the general format of an XDL file. The internal structure is defined by keywords and parentheses. Lines 2 through 17 declare the report. Line 2 begins with keyword `xdl_resource_report` and declares the XDL version, the device package, and the family name.

It is important to note that these reports are actually package-specific, and will present differences across the packages for a single device. IOB sites which may be bonded in some packages may be unbonded in others. A bonded site is one which has a wirebond from the die pad to the package leadframe. And unbonded site does not have such a wirebond, and is consequently inaccessible from the package leads.

Lines 4 through 11 declare the grid dimensions and all of the tiles in the device. Line 4 begins with keyword `tiles` and specifies a grid 19 rows by 33 columns in size. Lines 6 and 7 and lines 9 and 10 declare two different tiles. Line 6 begins with keyword `tile` and declares tile `TL` of type `UL` at grid coordinates `[0,0]`. Line 9 also begins with keyword `tile` and declares tile `BRAM_TOPCO` of type `LBRAM_TOPS_GCLK` at grid coordinates `[0,1]`. The tile declarations will be revisited in greater depth in the next section.

Lines 13 and 14 declare primitive site definitions. One site definition is included for each type of site referenced in the tile declarations. The site definitions provide additional information about logic and routing resources internal to the site. This information is not currently used by ADB and will not be discussed here.

⁸The few exceptions to this rule will be discussed later on.

```

1  # report header
2  (hdl_resource_report v0.1 xv50ecs144-8 virtexe
3    # grid dimensions and tile declarations
4    (tiles 19 33
5      # tile declaration
6      (tile 0 0 TL UL 2
7        :
8      )
9      # tile declaration
10     (tile 0 1 BRAM_TOPCO LBRAM_TOPS_GCLK 0
11       :
12     )
13     # primitive site definitions
14     (primitive_defs 14
15       :
16     )
17   # report summary
18   (summary tiles=627 sites=1957 sitedefs=14 numpins=24716 numpips=686950)
19 )

```

Figure 3.16: XDL file structure

And finally line 16 summarizes the number of tiles, number of sites, number of site types, number of site inputs and outputs, and number of pips⁹ in the entire device.

3.5.2 Tile Declaration

Figure 3.17 shows the format of an XDL tile declaration. Line 2 begins with keyword `tile` and declares tile TL of type UL at grid coordinates [0,0], with 2 declared sites.

Lines 4 and 5 declare a tile site. Line 7 declares a tile wire. And lines 9 declares a tile arc. The site, wire, and arc declarations are each optional, but may occur as many times as necessary.

Line 11 begins with keyword `tile_summary` and summarizes tile TL of type UL as having 14 inputs or outputs, 103 wires, and 329 arcs. The tile name and type used in this summary are expected to match those of the tile currently being declared.

⁹A pip is a Programmable Interconnect Point as defined in Section 2.6; when a connection is in place an arc can be said to exist, so a pip may be thought of as a possible arc.

```

1  # tile declaration
2  (tile 0 0 TL UL 2
3      # site declarations
4      (primitive_site BSCAN BSCAN internal 10 0
5          :
6          :
7          :
8          # wire declarations
9          (wire UL_DRCK1 0)
10         :
11         :
12         # pip declarations
13         (pip TL UL_H6C2 -> UL_TD02)
14         :
15         :
16         # tile summary
17         (tile_summary TL UL 14 103 329)
18     )

```

Figure 3.17: XDL tile declaration

3.5.3 Site Declaration

Figure 3.18 shows an XDL site declaration. Line 2 begins with keyword `primitive_site` and declares internal site `BSCAN` of type `BSCAN`, with index 0, and 10 declared input or output pins. A site may be *internal* if it is not an IOB, or *bonded* if it is an IOB that is wirebonded to the leadframe, or *unbonded* if it is an IOB that is not wirebonded to the leadframe.

```

1  # site declaration
2  (primitive_site BSCAN BSCAN internal 10 0
3      # site inputs and outputs
4      (pinwire RESET output UL_RESET)
5      (pinwire DRCK1 output UL_DRCK1)
6      (pinwire DRCK2 output UL_DRCK2)
7      (pinwire SHIFT output UL_SHIFT)
8      (pinwire TDI output UL_TDI)
9      (pinwire TD01 input UL_TD01)
10     (pinwire TD02 input UL_TD02)
11     (pinwire UPDATE output UL_UPDATE)
12     (pinwire SEL1 output UL_SEL1)
13     (pinwire SEL2 output UL_SEL2)
14 )

```

Figure 3.18: XDL site declaration

Line 4 begins with keyword `pinwire` and declares a site `output` pin with local name `RESET` and global name `UL_RESET`. A pinwire is simply a site input or output.

3.5.4 Wire Declaration

Figure 3.19 shows XDL wire declarations. Lines 2 through 5 show a wire connected in two neighboring tiles. Line 2 begins with keyword `wire` and declares a wire with global name `TBUF2` and two intertile connections. Line 3 begins with keyword `conn` and declares a permanent physical connection to wire with global name `TBUF1` in the tile named `R8C19`.

```

1 # wire connected to two tiles
2 (wire TBUF2 2
3   (conn R8C19 TBUF1)
4   (conn R8C21 TBUF3)
5 )
6 # wire connected to one tile
7 (wire TBUF3 1
8   (conn R8C19 TBUF2)
9 )
10 # wire not connected outside current tile
11 (wire TBUF0 0)

```

Figure 3.19: XDL wire declarations

Lines 7 through 9 show a wire connected in only one neighboring tile. And line 11 shows a wire without connections in any neighboring tiles.

The wire connections described here are partial *segment* declarations. No configurable resources come into play in these connections. The correct interpretation of lines 2 through 5 is that the wire named `TBUF2` in the current tile is part of a single electrical node which takes on the name `TBUF1` in tile `R8C19` and the name `TBUF3` in tile `R8C21`.

It is also important to realize that lines 2 through 5 do not necessarily define the entire segment. The declaration of `TBUF3` in tile `R8C21` may list additional connections to yet other tiles. In fact the only way to infer entire segments is to accumulate all of the cross-referenced connections throughout the entire device.

Unfortunately some of these declarations are incomplete, and are therefore undependable as a source of device segment information. If these declarations had been complete, they could have been used instead of the BXD data, and the BXD processing step could have been skipped entirely.

3.5.5 Pip Declaration

Figure 3.20 shows XDL pip declarations. Lines 2, 4, 6, and 8 each begin with keyword `pip` and the name of the tile in which wires exist. In practice pips always seem to be declared in the tiles that the wires live in,¹⁰ so the tile name always matches the current tile name.

```

1 | # unidirectional arc
2 | (pip R8C20 H6W2 -> V6S2)
3 | # bidirectional arc - this pip controls only one direction
4 | (pip R8C20 V6S2 == H6E1)
5 | # bidirectional arc - this pip controls both directions
6 | (pip R8C20 W15 == N10)
7 | # unidirectional arc routed through a logic site
8 | (pip R8C20 S0_G_B2 -> S0_Y (_ROUTETHROUGH-G2-Y SLICE))

```

Figure 3.20: XDL pip declarations

Line 2 declares a unidirectional connection (indicated by `->`) from `H6W2` to `V6S2`.

Line 4 declares a bidirectional connection (indicated by `==`) between wires `V6S2` and `H6E1`, and indicates that each direction is controlled by a separate set of configuration bits.

Line 6 declares a bidirectional connection (indicated by `==`) between wires `W15` and `N10`, and indicates that both directions are controlled by the same set of configuration bits.

Line 8 declares a unidirectional connection (indicated by `->`) from wires `S0_G_B2` and `S0_Y`, and further indicates that this connection is achieved by routing the signal through a logic site.¹¹ This type of connection is not presently supported by ADB.

3.6 .xdlrules Files

Because the XDL data is *device* specific (and furthermore *package* specific) and because XDL files are so large, it is convenient to distill that data into the more compact and manageable *.xdlrules* format. While this no longer represents *raw source data*, it still used as source data by the database build tools, and will therefore be described here. *.xdlrules* files are generated in a preliminary pre-processing step.

¹⁰In the case of remote arcs, this is at odds with the BFD practice of declaring the arcs in the tile type that the configuration bits appear in. This difference substantially increases the difficulty in reconciling BFD and XDL arcs.

¹¹A routethrough achieves a connection by routing a signal through logic resources instead of normal routing resources. See *Routethrough* in *Section 2.6*.

As stated earlier, the XDL data is used to supplement the BFD tile type data, so XDL *tile* data must be converted into *tile type* data. Furthermore data for the entire range of devices in a family can be combined into a single representative set. These two steps together permit many gigabytes of data to be distilled into less than one megabyte while still retaining all necessary information. The processing steps required to generate the *.xdlrules* files will be described in Chapter 5.

Not all of a family's tile types necessarily appear in each device in the family. In such cases it is helpful to identify a *minset*¹² of devices which encompass all of the family's tile types.

The *.xdlrules* files specify:

- The tile types which occur in a family, and the minset of devices in which they occur.
- The site types which occur in each tile type, and the names that those sites are given. *Note that because only a necessary subset of devices is considered, the site names will not include every possible name used in the entire family. However this will identify the globally unique site names which are needed when processing the BFD data.*
- The wires in each tile type which are site inputs or outputs.
- The set of all possible directed arcs in each tile type.

All wire names used in *.xdlrules* files are global names.

3.6.1 File Structure

An *.xdlrules* file consists of a sequence of tile type declarations. A tile type declaration terminates when the next tile type is encountered or when the end of the file is reached.

3.6.2 Tile Type Declaration

Figure 3.21 shows the general format of an *.xdlrules* tile declaration. Line 2 begins with keyword XDL_TILE and declares tile type BL_DLLIOB, specifying that this tile type occurs in minset devices XCV405E and XCV50E.

Lines 4 and 5 declare sites that occur within the current tile type. Line 4 begins with keyword XDL_SITE and declares site type DLLIOB, specifying that this site type takes on the names AM18 and M6.

Lines 7 and 8 declare wires that can be identified as inputs or outputs. Line 7 begins with keyword XDL_PINWIRE and declares the wire named BL_DLLIOB_IOFB to be an output.

¹²The term *minset* is not attributed any rigorous mathematical meaning in the present context.

```

1 # tile type declaration
2 XDL_TILE: BL_DLLIOB XCV405E XCV50E
3 # site declarations
4 XDL_SITE: DLLIOB AM18 M6
5 XDL_SITE: EMPTYIOB EMPTY221 EMPTY222 EMPTY89 EMPTY90
6 :
7 :
8 # input and output declarations
9 XDL_PINWIRE: BL_DLLIOB_IOFB output
10 XDL_PINWIRE: BOT_CLKO input
11 :
12 :
13 # pip declarations
14 XDL_PIP: BOT_FAKE_LHO BOT_H6E0 BOT_H6E1 BOT_H6E5 BOT_H6W0 BOT_H6W1 BOT_H6W4
15 XDL_PIP: BOT_FAKE_LH6 BOT_H6E2 BOT_H6E3 BOT_H6E4 BOT_H6W2 BOT_H6W3 BOT_H6W5
16 :
17 :

```

Figure 3.21: .xdlrules file structure

And Lines 10 and 11 declare possible directed arcs. Line 10 begins with keyword `XDL_PIP` and declares that wire `BOT_FAKE_LHO` can drive wires `BOT_H6E0`, `BOT_H6E1`, `BOT_H6E5`, `BOT_H6W0`, `BOT_H6W1`, and `BOT_H6W4` in the current tile.

The `XDL_SITE`, `XDL_PINWIRE`, and `XDL_PIP` declarations are all optional, but may occur as many times as necessary in each tile type.

3.7 Family Specific Perl Code

Each of the device families supported by ADB contains various exceptions. Instead of hardcoding family-specific exceptions in the database build code, separate scripts were provided for each family. Because the database build code exists as a collection of Perl scripts, the family-specific code was also written in Perl. Only the script for the appropriate family is invoked when its databases are built. Though these scripts were not developed by Xilinx, most of the functions that they contain are based on extensive correspondence with Jeff Lindholm at Xilinx.

The family scripts may contain private functions and data, but each must also implement a set of required functions. The required functions are as follows:

node_is_hidden(): Determine whether a node should be marked as hidden. Hidden nodes can be omitted from traces, which can substantially increase the signal-to-noise ratio of the trace output. This can prevent a trace from reporting thousands of connections that exist even in empty designs.

- correct_xdl_data():** Correct errors or omissions in the XDL data. In fact this applies to the distilled *.xdlrules* data since the build code does not read XDL data directly. *Patching the data at runtime seems more proper than tampering with the source data.*
- correct_bfd_lines():** Correct errors or omissions in the BFD data. *Patching the data at runtime seems more proper than tampering with the source data.*
- filter_bfd_expression():** Filter or override BFD expressions. This can be used as a pseudo-preprocessor for BFD files that target multiple families.
- get_relative_remote_tile_type():** Return the tile type which a given relative remote node may live in. This is required because the relative [row,col] offsets provided in BFD data are expected to “magically” skip over certain tile types. *This function is a look-ahead form of adjust_remote_offset() for use before the device data has been processed.*
- adjust_remote_offset():** Adjust the given relative [row,col] offset if necessary. This is required because the relative [row,col] offsets provided in BFD data are expected to “magically” skip over certain tile types.
- arc_is_trimmed_from_bfd():** Indicate whether a given arc has been trimmed from the BFD data. Arcs that were unexpectedly trimmed can make it difficult to reconcile BFD and XDL data.
- arc_is_trimmed_from_xdl():** Indicate whether a given arc has been trimmed from the XDL data. Arcs that were unexpectedly trimmed can make it difficult to reconcile BFD and XDL data.
- invert_system():** Invert a system of BFD equations. This function usually passes the system along to the real inversion code, but in a very small number of cases it may include hints or even directly return a hardcoded result.
- adjust_segments():** Adjust BXD segments if necessary. This is a last-ditch effort to handle a small class of permanent connections which are not specified in any of the source data.

The next three sections describe these functions as implemented for the Virtex, Virtex-E, and Virtex-II. Many of these descriptions are deliberately simplified in order to protect Xilinx intellectual property.

3.7.1 Virtex

- node_is_hidden():** Nodes CO_0_LOCAL, CO_1_LOCAL, CIN_0, CIN_1, and F5_COUPLING1 – F5_COUPLING4 in CENTER tile types are marked hidden.
- correct_xdl_data():** *Nothing to do.*
- correct_bfd_lines():** The BFD file fails to declare a few nodes in two tile types.

filter_bfd_expression(): The same BFD file is shared by the Virtex and Spartan-II families. This function sets all Spartan-II-related functions to **FALSE**.

get_relative_remote_tile_type(): Relative remote nodes always live in **CENTER** tile types.

adjust_remote_offset(): Incorporate Xilinx code to “magically” skip over certain tile types.

arc_is_trimmed_from_bfd(): A small number of “unused” arcs are trimmed from various IOB type tiles.

arc_is_trimmed_from_xdl(): *Nothing to do.*

invert_system(): Assist with systems in which Spartan-II code has obscured the mutual exclusivity of multiplexer arcs. Handle three bidirectional equations which the inverter is unable to process. Provide hints for two arcs which could not be brute-forced. Remove routing sniffers which add complexity but don’t change the function. Remove references to clock buffer components since they can’t be inferred from bitstreams anyhow.

adjust_segments(): *Nothing to do.*

3.7.2 Virtex-E

node_is_hidden(): Nodes **CO_0_LOCAL**, **CO_1_LOCAL**, **CIN_0**, **CIN_1**, and **F5_COUPLING1–F5_COUPLING4** in **CENTER** tile types are marked hidden.

correct_xdl_data(): *Nothing to do.*

correct_bfd_lines(): Corrects two incorrectly declared nodes, renames a few others, and removes redundant declarations.

filter_bfd_expression(): *Nothing to do.*

get_relative_remote_tile_type(): Relative remote nodes always live in **CENTER** tile types.

adjust_remote_offset(): Incorporate Xilinx code to “magically” skip over certain tile types.

arc_is_trimmed_from_bfd(): A substantial number of “unused” arcs are trimmed from various IOB type tiles.

arc_is_trimmed_from_xdl(): Two dozen arcs in **MBRAM** tile types appear to be missing. These are artifacts originating with the definition of the parent tile type.

invert_system(): Handle three bidirectional equations which the inverter is unable to process. Provide hints for two arcs which could not be brute-forced. Remove routing sniffers which add complexity but don’t change the function. Remove references to clock buffer components since they can’t be inferred from bitstreams anyhow.

adjust_segments(): *Nothing to do.*

3.7.3 Virtex-II

node_is_hidden(): The following nodes are marked hidden:

DIFFO_OUT0, DIFFO_OUT2, PADOUT0–PADOUT3, and VCC_WIRE in tile types LR_IOIS and TB_IOIS.

SOPOUT0–SOPOUT3, FX0–FX3, F50–F53, COUT0–COUT3, BXOUT0–BXOUT3, BXINVOUT0–BXINVOUT3, BYOUT0–BYOUT3, BYINVOUT0–BYINVOUT3, DIG0–DIG3, SHIFTOUT0–SHIFTOUT3, DX0–DX3, DY0–DY3, and VCC_PINWIRE in tile type CENTER.

VCC_WIRE in tile types UL, LL, UR, LR, CLKT, CLKB, and BRAM_IOIS.

correct_xdl_data(): Because of inheritance the clock tiles contain synonyms, and the BFD and XDL data disagree on which names to use. This function causes the XDL data to adopt the BFD convention. In addition five tile types are missing 15 to 20 arcs which show up in the BFD data. *The case of arcs missing from the XDL data is a blatant and troublesome exception to the rule that the XDL data specifies all of the arcs.*

correct_bfd_lines(): Insert some missing remote default nodes in the clock tiles.

filter_bfd_expression(): *Nothing to do.*

get_relative_remote_tile_type(): Relative remote nodes may live in CENTER or GCLKC tile types.

adjust_remote_offset(): Incorporate Xilinx code to “magically” skip over certain tile types.

arc_is_trimmed_from_bfd(): *Nothing to do.*

arc_is_trimmed_from_xdl(): A substantial number of “unused” arcs are trimmed from various BRAM type tiles.

invert_system(): Handle a single bidirectional equation which the inverter is unable to process.

adjust_segments(): There is no configuration bit to control certain arcs in the left-most CENTER tiles. Since these connections are permanent, the segments on either side of the arcs are simply merged.

Chapter 4

Database Design

4.1 Overview

Because the primary purpose of ADB is to support routing and tracing services, it must understand the wiring inside devices, and must understand how wiring connections are made. ADB draws that understanding from a set of family and device database files.

In fact ADB consists of the wiring data as well as the code which utilizes it. The code is responsible for opening the database files, for managing the data that they contain, and for providing interfaces to external clients.

Although ADB may be used in standalone mode, it is expressly designed to work with JBits. This requirement makes Java a fairly obvious choice for a language, since JBits is also written in Java. An obvious benefit of Java is its cross-platform compatibility. But the choice of Java as a language also has some substantial downsides.

It turns out that Java is highly inefficient in dealing with small objects. This makes the cost of data encapsulation prohibitively high, and precludes the use of object-oriented features for the overwhelming majority of the data. On the other extreme, Java's lack of support for multiple inheritance results in artificially large class hierarchies to interface with JBits. In a strange way, Java is both too simple and too complex for ADB's needs.

This chapter explains how the data is represented in memory and in the database files. Significant attention is paid to the quantity of data involved, and efficient representation schemes. A variety of compression schemes are used together to reduce the size of data in memory and on disk, without significant performance penalties.

The chapter also details the kinds of information that must be represented, and how the various pieces relate to each other. Figure 4.1 shows the major components of the database.

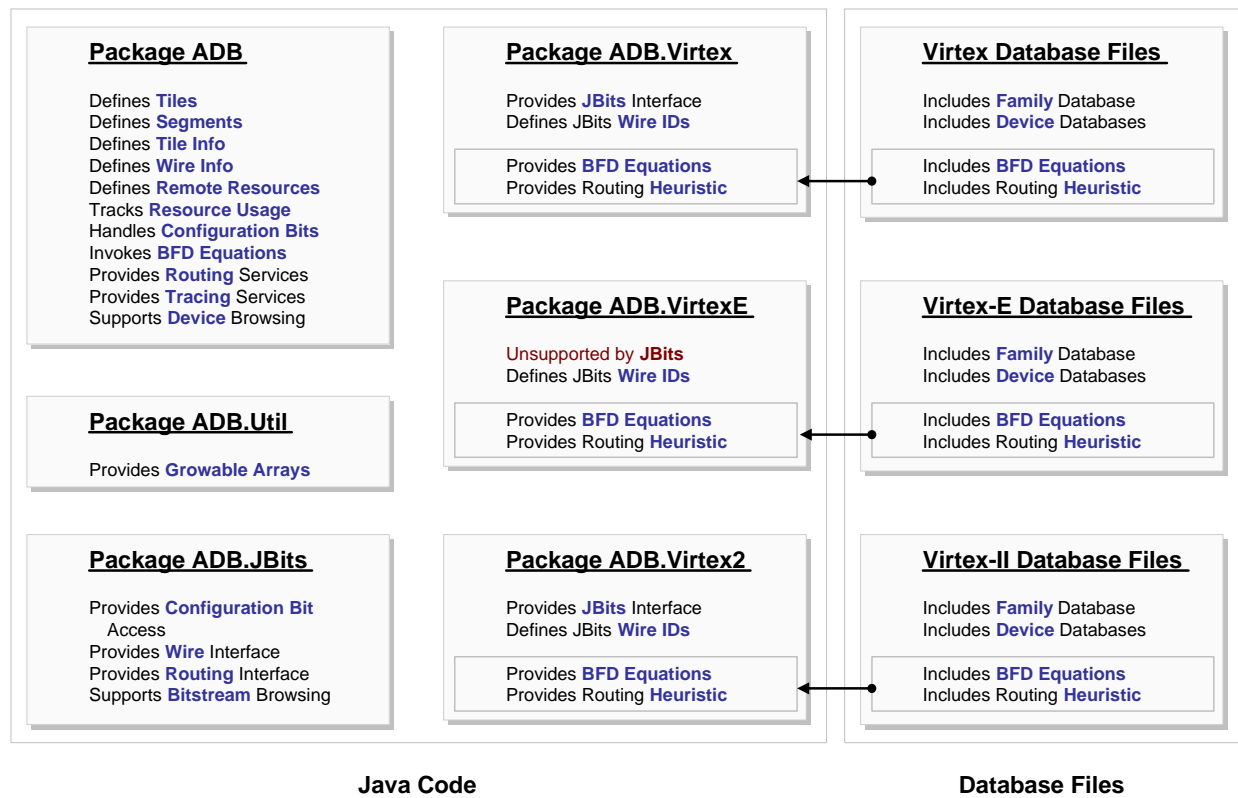


Figure 4.1: Database components

4.2 Objectives

ADB has the following design objectives:

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

4.3 Prior Work

Both JRoute and WireDB include wiring databases and present design ideas which are worth examining.

4.3.1 JRoute

JRoute is the wire database and router which is bundled with JBits. When JRoute was initially introduced it only provided routing support for CENTER tiles in Virtex devices [30], though subsequent versions added support for IOB and BRAM tiles.¹

JRoute's internal database was encoded as a set of Java classes, comprised of a large number of arrays. The arrays specified the wire names, wire indexes, wire classes, directions, sources and sinks, and the configuration bit patterns needed to set or clear arcs.

Because JRoute only supported CENTER, IOB, and BRAM tiles, it was relatively independent of the device in use. The CENTER tiles were assumed to form a grid in the middle of the device, with the IOB tiles around the periphery, and the BRAM tiles just inside the IOB columns on the right and left sides of the device.

While this approach provided very good performance, it did not easily support coverage of other wires in other tiles. Nevertheless JRoute has been used extensively by JBits users, and has a good and well deserved track record.

4.3.2 WireDB

WireDB was an independent wire database intended for use with JBits. Unlike JRoute it used device-specific databases to store the wiring inside each device.

While JRoute determined intertile segment connections at runtime, WireDB precomputed all such connections, and provided fully resolved databases in the form of interrelated Java classes. During the database build these classes were serialized and written to disk in compressed form, so that when the databases were opened, the classes could simply be decompressed and deserialized.

Two performance issues arose with WireDB. Firstly, certain Java Virtual Machines (JVMs) were excessively slow in deserializing the classes. Secondly, these fully resolved databases were enormous both on disk and in memory. For example the database file for the Virtex XCV600 was 7.4 MB on disk in compressed form, and over 32.6 MB in memory when uncompressed. This would clearly not scale well for the larger Virtex-E and Virtex-II devices.

¹The current version of JBits separates the routing function from the wiring database, so that the user is free to mix and match routers and databases according to their needs.

4.4 Device Sizes

In order create an efficient database design it is important to understand the quantity of data involved, including the number of tiles, the number of segments, and the number of individual wires in each device. Tables 4.1, 4.2, and 4.3 provide this information for Virtex, Virtex-E, and Virtex-II families.

Table 4.1: Virtex dimensions

<i>Device</i>	<i>Rows</i>	<i>Columns</i>	<i>Tiles</i>	<i>Segments</i>	<i>Wires</i>
XCV50	19	31	589	104,339	216,070
XCV100	23	37	851	157,935	325,170
XCV800	59	95	5,605	1,138,035	2,334,806
XCV1000	67	107	7,169	1,477,195	3,020,606

Table 4.2: Virtex-E dimensions

<i>Device</i>	<i>Rows</i>	<i>Columns</i>	<i>Tiles</i>	<i>Segments</i>	<i>Wires</i>
XCV50E	19	33	627	105,947	223,214
XCV100E	23	39	897	159,927	333,942
XCV2600E	95	159	15,105	3,040,135	6,284,550
XCV3200E	107	181	19,367	3,868,867	8,019,870

Table 4.3: Virtex-II dimensions

<i>Device</i>	<i>Rows</i>	<i>Columns</i>	<i>Tiles</i>	<i>Segments</i>	<i>Wires</i>
XC2V40	15	17	255	39,199	111,506
XC2V80	23	17	391	69,651	191,590
XC2V6000	115	105	12,075	3,171,411	7,779,782
XC2V8000	131	121	15,851	4,318,059	10,463,182

The numbers in the larger devices are clearly problematic in light of the market trend toward more and more powerful devices. While the XC2V40 in Table 4.3 contains a mere 39,199 segments and 111,506 wires, the XC2V8000 contains 4,318,059 segments and 10,463,182 wires. The quantity of

data involved has direct implications for the database files as well as their memory footprints. As a result the data size and overhead issues take on critical importance.

4.5 Java Data Types

Because ADB is implemented in Java, the available Java data types are important in this discussion. Table 4.4 shows the Java primitive data types presented in [32] and [33].

Table 4.4: Java primitive data types

<i>Type</i>	<i>Description</i>	<i>Size</i>	<i>Range</i>
<code>boolean</code>	<code>true</code> or <code>false</code>	1 bit	n/a
<code>char</code>	unicode character	16 bits	\u0000 through \uFFFF
<code>byte</code>	signed integer	8 bits	-128 through 127
<code>short</code>	signed integer	16 bits	-32,768 through 32,767
<code>int</code>	signed integer	32 bits	-2,147,483,648 through 2,147,483,647
<code>long</code>	signed integer	64 bits	-2^{63} through $2^{63} - 1$
<code>float</code>	floating point	32 bits	$\pm 1.4 \times 10^{-45}$ through $\pm 3.4 \times 10^{38}$
<code>double</code>	floating point	64 bits	$\pm 4.9 \times 10^{-324}$ through $\pm 1.8 \times 10^{308}$

Java's primitive data types do not include unsigned integers. The signed integer types use 2's complement representation, with the most significant bit as a sign flag. It is possible to store unsigned integer data in signed integers, but the effective range of the unsigned integers is reduced by a factor of two.

A factor to consider when choosing data types is the way they are handled inside the JVM. In particular, types `byte`, `char`, and `short` are not directly supported by most bytecode instructions. Instead these types undergo conversions to and from `int` when they are used [33]. While this does not change the memory size of these types, it does add a slight performance penalty. In other words, operations on type `int` are faster than operations on types `byte`, `char`, and `short`. This behavior is part of the formal JVM specification, and therefore inherent in every JVM implementation.

Another JVM issue to bear in mind is that “[t]he Java virtual machine does not mandate any particular internal structure for objects” [33]. This means that there may be no way of knowing what overhead is associated with an object on any given JVM. At this point it is simply noted that objects include an unspecified amount of overhead.

4.6 Data Type Selection

From a routing or tracing perspective, an FPGA is a very large collection of tiles and wires. The quantity of data, and the frequency with which it must be used, call for efficient data types and algorithms. While this chapter focuses mainly on data representation, it does so with the understanding that algorithm performance may be significantly affected by the chosen representation.

Since the BFD, BXD, and XDL files refer to tiles and wires by name, one possible approach would be to simply use those names in ADB. But names must generally be encoded as strings, and strings are not very efficient data types for storage or for comparison. Furthermore while names are helpful to people, they are completely unnecessary for computers.

A more sensible approach is to encode tiles and wires with integer indexes, while still maintaining a mapping from indexes to names. Since each name only needs to be stored one time, the mapping overhead should not be very significant. Furthermore, the indexes can directly address the corresponding names if these names are stored in arrays.

4.6.1 Object Overhead Issues

ADB deliberately avoids the use of objects for small but numerous data types. This is partly a result of the lessons learned from WireDB, and partly a result of empirical experimentation. Purists may argue that this defies good object-oriented programming techniques [34], but pragmatists will counter that the size reduction is a valid and necessary optimization [35].

The two-fold root of the problem is that 1) *every* Java object inherits from `java.lang.Object`, and 2) Java has to track objects in order to provide garbage collection. This means that an object that does nothing more than encapsulate an `int`, will acquire all the overhead of `java.lang.Object`, with its virtual function table, and an associated reference somewhere in the JVM to be used for garbage collection. All of this greatly increases the size of the `int` which was to be encapsulated, and makes Java's object penalty too costly. Naturally in the case of larger objects, the overhead represents a smaller percentage of the total size, so the penalty associated with large objects may be much more affordable.

Table 4.5 shows the resulting memory usage and overhead for 1,000,000 `java.lang.Integer` objects. The `java.lang.Integer` type encapsulates a 32 bit `int`, so with no object overhead the memory usage should be 4,000,000 bytes. The *Overhead* column in the table expresses the overhead as the actual memory usage divided by 4,000,000 bytes. Thus a `java.lang.Integer` takes up to *five* times more memory than an `int`. Figure 4.2 shows the `IntegerTest` class used to generate the data for Table 4.5.²

²It is interesting to note that while the Sun 1.2.2 JVM has a lower overhead than other JVMs, it was *excessively* slower in opening WireDB database. Obviously there is an internal tradeoff between memory overhead and speed.

Table 4.5: java.lang.Integer object overhead (1,000,000 objects)

<i>Operating System</i>	<i>JVM</i>	<i>Bytes Used</i>	<i>Overhead</i>
Linux	Sun 1.1.8	20,000,016	5.000,004
Linux	Sun 1.3.0.02	19,999,888	4.999,972
Linux	IBM 1.3.0	20,001,328	5.000,332
Windows NT	Sun 1.2.2 JRE	15,999,968	3.999,992
Windows 2000	Sun 1.3.1	20,000,016	5.000,004

```

1 public class IntegerTest {
2     public static void main(String args[]) {
3
4         // initialization
5         int size = 1000000;
6         Runtime runtime = Runtime.getRuntime();
7
8         // prime the garbage collector
9         Integer[] dummy = new Integer[size];
10        dummy = null;
11
12        // force garbage collection and determine memory usage
13        System.gc();
14        long initial_usage = runtime.totalMemory() - runtime.freeMemory();
15
16        // allocate the specified number of integers
17        Integer[] big_integer = new Integer[size];
18        for(int i = 0; i < size; i++) big_integer[i] = new Integer(i);
19
20        // force garbage collection and determine memory usage
21        System.gc();
22        long total_usage = runtime.totalMemory() - runtime.freeMemory() - initial_usage;
23
24        // report usage statistics
25        double int_overhead = (double) (total_usage) / (double) (size) / 4;
26        System.out.println("Used " + (total_usage) + " bytes for " + size
27            + " elements; int overhead factor: " + int_overhead);
28        System.out.println("Java " + System.getProperty("java.version") + " on "
29            + System.getProperty("os.name") + ". " + System.getProperty("java.vendor"));
30
31    }
32 }

```

Figure 4.2: java.lang.Integer object overhead test code

ADB could have been implemented more simply and elegantly in C++. Though C++ is sometimes maligned in the Java community, it has the following advantages: 1) C++ allows classes that do not inherit from a base class, which eliminates superclass overhead. 2) Classes which contain no virtual functions are not given virtual function tables, which eliminates virtual function table overhead. And 3) C++ does not perform automatic garbage collection, which eliminates the need for external object references. In other words C++ can efficiently encapsulate small data types in objects that carry no size penalty, as long as virtual functions are not needed [36].

The decision to avoid using objects for small but numerous data types has a significant side effect. The `java.util.Collection` and `java.util.Map` interfaces work with objects but do not work with primitive types. This means that the commonly used `LinkedList`, `ArrayList`, `Vector`, `Stack`, `HashSet`, `TreeSet`, `HashMap`, `TreeMap`, `WeakHashMap`, and `Hashtable` are unusable [32]. Encapsulating primitive types in objects like `java.lang.Integer` simply reintroduces the overhead described above.

However, most of the ADB data can be efficiently stored in arrays. In cases where no index is readily available and ADB has to search for an integer entry, the data is generally presorted in order to allow logarithmic time binary searches.

4.6.2 Tile Index

A *tile index* uniquely identifies a tile within a device. While the source data uses tile names, the tilemap data allows conversions between tile names and coordinates. These coordinates can in turn be combined into tile indexes with the following formula:

$$\text{tile_index} = (\text{row} \times \text{column_count}) + \text{column}$$

Tables 4.1, 4.2, and 4.3 show that the device with the largest number of tiles is the Virtex-E XCV3200E, with 19,367 tiles.

ADB represents a tile index as a 16 bit unsigned integer. This data type can accommodate a maximum of 65,536 tiles, or 256 rows \times 256 columns. It is likely that future devices will contain more than 65,536 tiles (though that may or may not occur during ADB's lifetime), in which case the ADB databases and code would have to be modified. *This is a known limitation of ADB.*

As shown in Table 4.4, Java does not provide any unsigned data types. Therefore a 16 bit signed `short` cannot be used to store a tile index. This turns out not to be a problem because tile indexes are always 1) bundled in a larger data type, as described below, or 2) temporarily placed in a 32 bit signed `int`.

4.6.3 Wire Index

A *wire index* is used to identify a wire *within* a tile type. The wire index has no meaning apart from the tile or tile type with which it is associated.

The largest number of wires in any tile type in the Virtex, Virtex-E, or Virtex-II families is 1,267. As in the case of tiles, the BFD, BXD, and XDL files refer to wires by local or global name. These names are mapped to indexes by the database build code.

ADB represents a wire index as a 15 bit unsigned integer, which can therefore be stored as a **short**. This data type can accommodate a maximum of 32,768 wires, which is vastly more than the required 1,267 wires.

4.6.4 Tilewire

A *tilewire* is a data type which contains a tile index and a wire index. A tilewire can uniquely identify any wire in a device.

Because tilewires are used so frequently and in such large quantity, the tile and wire indexes are combined into a 32 bit **int**, as shown in Figure 4.3.

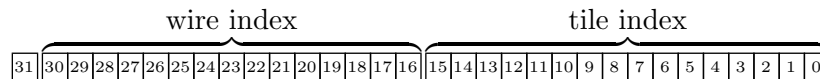


Figure 4.3: Tilewire bit usage

Because of the overhead issues described earlier, the tilewire data is not encapsulated into a class, so the wire and tile indexes must be packed or unpacked whenever they are used.

4.7 Tiles

All of the logic and routing resources inside FPGAs are organized into tiles. Each tile has a name and a set of coordinates, and is associated with a tile type. The tile type serves as a template for identical tiles which may occur many times inside a device.

4.7.1 TileInfo Class

Figure 4.4 shows the structure of the **TileInfo** class. A **TileInfo** object exists for each tile in the device.

ADB.TileInfo	
short	type_index
short	row
short	col
String	name

Figure 4.4: TileInfo class

The `type_index` field holds the appropriate tile type index for this tile. The tile type index performs the mapping between the device tile and the family tile types.

The `row` and `col` fields hold the tile's row and column coordinates. The `name` field holds the tile's name.

4.7.2 WireInfo Class

Figure 4.5 shows the structure of the `WireInfo` class. A `WireInfo` object exists for each wire in every tile type in the family, and stores the wire's name as well as its attributes and properties.

ADB.WireInfo	
short	offset
short	attributes
String	name
short	dependencies[]
short	groups[]
short	sinks[]
short	sources[]
short	tied_sinks[]
short	tied_sources[]

Figure 4.5: WireInfo class

The `offset` field indicates how many arcs exist in preceding wires in the tile type. If the *first* wire in a tile type could connect to *five* other wires, then *its* `offset` would be *zero*, and the *second* wire's `offset` would be *five*. This field is used by the `ArcUsage` class described later.

The `attributes` field is treated as a bitfield to indicate if the wire:

- should be *hidden* in traces
- is an *input* to a logic site
- is an *output* from a logic site

- references a *remote* wire in another tile
- is referenced by a *remote* wire in another tile

The `name` field holds the wire's name. This name matches the local name used in the BFD data.

The `dependencies[]` array holds this wire's dependent equation groups. A dependent equation group is an equation group which must be re-evaluated if this wire becomes used or unused. For example, if the entries in the `dependencies[]` array are (10,11,204), and this wire becomes used or unused, then equation groups 10, 11, and 204 must be re-evaluated. Dependencies originate with `nodeused()` functions in BFD equations.

The `groups[]` array holds the equation group index corresponding to each sink that this wire can drive. If an arc is made from this wire to another one, the corresponding equation group listed here must be re-evaluated. For example if an arc is made from this wire to the fifth wire in the `sinks[]` array, then the fifth equation group in the `groups[]` array must be re-evaluated.

The `sinks[]` and `sources[]` arrays hold the indexes of all the wires that this one can drive or be driven by. This wire can connect to any wire in the `sinks[]` array, and all the wires in the `sources[]` array can connect to this wire. As explained above, if a connection is made from this wire to a wire in the `sinks[]` array, the corresponding equation group in the `groups[]` array must be re-evaluated.

The `tied_sinks[]` and `tied_sources[]` arrays hold the indexes of all the wires that this one permanently drives or is permanently driven by. Entries in these arrays represent *fake arcs* which are always on and do not require any configuration bits. Because no configuration bits are involved, no equation groups are associated with `tied_sinks[]` entries.

Normal routing and tracing functions proceed in a forward direction, from sources to sinks. However the JBits internal Wire Database API requires the ability to enumerate and work with a wire's sources. The source arrays are provided in order to efficiently support this requirement. These arrays are not stored on disk but are instead dynamically generated from the `sinks[]` and `tied_sinks[]` arrays when the database is initialized.

4.7.3 Tiles Class

Figure 4.6 shows the structure of the `Tiles` class. ADB uses a single `Tiles` object to keep track of all tile information in the entire device.

The `bit_rows[]` and `bit_cols[]` arrays are ordered by tile type index, and hold the configuration bit row and column counts for each tile type. For example if a particular tile type has 18 rows by 48 columns of configuration bits, then its entries in `bit_rows[]` and `bit_cols[]` will be 18 and 48 respectively. The information is currently unused but reserved for future expansion.

ADB.Tiles	
short	bit_rows[]
short	bit_cols[]
String	tile_types[]
boolean	abstract_tile[]
TileInfo	tiles[]
int	tile_map[][]
WireInfo	wires[][]
short	groups[][]
short	jbits_id[][]
short	row_count
short	col_count

Figure 4.6: Tiles class

The `tile_types[]` array is ordered by tile type index, and holds the name of each tile type in the family. The `abstract_tile[]` array is ordered by tile type index, and holds a flag indicating whether or not each tile type is abstract. An abstract tile type is one which is never instantiated by any device in the family. Abstract *tile types* may be incomplete because there is no XDL data to describe them, but abstract *tiles* cannot exist and therefore cannot occur in real devices, so this does not present a problem.

The `tiles[]` array is ordered by tile index, and holds a `TileInfo` object for each tile in the device.

The `tile_map[][]` array is two-dimensional, ordered first by row and then by column. This array holds the corresponding tile index for each set of row and column coordinates. For example, to obtain the tile index for the tile at row 12 and column 23, one can look at `tile_map[12][23]`. The reverse mapping can be obtained with the help of the `tiles[]` array.

The `wires[][]` array is two-dimensional, ordered first by tile type index and then by wire index. This array holds the `WireInfo` entry for each set of tile type and wire indexes.

The `groups[][]` array is ordered by tile type, and holds an array of equation group indexes for each tile type. This array allows ADB to quickly look up the equation groups used in a tile to determine which ones need to be re-evaluated. This field is used by the `GroupUsage` class described later.

The `jbits_id[][]` array is two-dimensional, ordered first by tile type index and then by the JBits wire ID. The JBits wire ID is assigned by ADB during the database build and is made available to JBits users in the family's `Wire.java` file. Wire indexes used by ADB only make sense in conjunction with a tile type, while wire IDs used by JBits are independent of tile type.

JBits users can construct a `Pin` object with row and column coordinates and with a constant defined in `Wire.java`, and ADB can perform a binary search of `jbits_id[][]` to identify the corresponding

ADB wire index. If a JBits user specifies a wire ID which does not exist in the given tile, ADB will throw an exception.

The `row_count` and `col_count` hold the total number of rows and columns in the device.

4.8 Remote Resources

BFD tile types define equations for local configuration bits, but they sometimes refer to wires or arcs from other tiles. These wires or arcs are treated as *remote* resources since they in fact reside in *remote* tiles.

Remote resources vary in location from one device to the next, and therefore cannot be associated with tile types as normal resources are. The remote resources are independently resolved for each device by the database build code, and are used to supplement the normal tile type resources.

4.8.1 RemoteNode Class

Figure 4.7 shows the structure of the `RemoteNode` class. A `RemoteNode` object exists for every remotely referenced node in the device.

ADB.RemoteNode	
short	local_wire_index
int	remote_tile_index
short	remote_wire_index

Figure 4.7: RemoteNode class

The `local_wire_index` field holds the node's wire index as it is specified in the local tile. The `remote_tile_index` and `remote_wire_index` fields hold the tile index and wire index of the resolved node. In other words, a wire in the local tile with index `local_wire_index` actually refers to the wire in tile `remote_tile_index` with index `remote_wire_index`.

`RemoteNode` objects are always stored under the local tile index, so there is no need to include a field for the local tile index in the object.

4.8.2 RemoteArc Class

Figure 4.8 shows the structure of the `RemoteArc` class. A `RemoteArc` object exists for every remotely referenced arc in the device.

ADB.RemoteArc	
short	source_wire_index
short	sink_wire_index
int	remote_tile_index
int	bits_tile_index
short	equation_group_index

Figure 4.8: RemoteArc class

The `source_wire_index` and `sink_wire_index` fields hold the indexes of the arc's source and sink wires.

The `remote_tile_index` field holds the index of the remote tile. In the case of a forward `RemoteArc` object, the `sink_wire_index` is associated with the `remote_tile_index`, and the `source_wire_index` is associated with the local tile. In the case of a reverse `RemoteArc` object, the `source_wire_index` is associated with the `remote_tile_index`, and the `sink_wire_index` is associated with the local tile.

`RemoteArc` objects are always stored under the local tile index, so there is no need to include a field for the local tile index in the object.

The `bits_tile_index` field holds the index of the tile in which the configuration bits for this arc reside. The `equation_group_index` field holds the equation group index which must be evaluated to set or clear this arc.

For example, to set a forward arc between wire `source_wire_index` in the current tile and wire `sink_wire_index` in tile `remote_tile_index`, it is necessary to mark the arc as used, and evaluate equation group `equation_group_index` in tile `bits_tile_index`.

While the source and sink wires always reside in the same tile for Virtex, Virtex-E, and Virtex-II families, that fact was not apparent when the `RemoteArc` class was first designed. Since the extra `int` penalty is small, and since the BFD syntax allows for arcs connecting wires in separate tiles, the `remote_tile_index` field was retained in case some future family requires this functionality.

4.9 Segments

The wiring resources inside FPGAs consist of *segments*. Because FPGAs are divided into tiles, a segment may either be entirely contained within a single tile, or may span multiple tiles. A segment may then reasonably be divided into a collection of *wires*, each of which is associated with exactly one tile.

If a segment consists of exactly one wire, it is called a *trivial segment*, and must necessarily be contained within a single tile. However, in the general case a segment spans a number of tiles, and

consists of one wire for each tile that it spans.³ As described in Section 2.6, a segment is a single electrical node, and the wires that make up that segment all belong to that single node.

The number of segments in devices can become very large, as previously shown by tables 4.1, 4.2, and 4.3, and anything that can be done to reduce the size of the data on disk or in memory is desirable.

4.9.1 Compacted Segments

The regular structure of FPGAs encourages the search for compression schemes. In addition to reducing disk and memory requirements, a suitable scheme must also operate without a large performance penalty.

Given any FPGA family, it is easy to see that certain types of segments occur again and again throughout the devices, as if replicated by a cookie cutter. A closer look confirms that these segments often retain their *shape* and differ only by a tile offset.

For example, the Virtex family includes a class of segment called a *single*. In general a *single* connects similarly named wires in two adjacent CENTER tiles. This kind of segment will exist every time two CENTER tiles abut, which clearly occurs many times even in the smaller devices. Figure 4.9 shows a single segment which connects two wires in adjacent CENTER tiles: the wire E7 in tile [1,2], and the wire W7 in tile [1,3]. In shorthand notation these wires will be called E7@[1,2] and W7@[1,3].

A straightforward way to store this segment information is to remember the full wire identifiers E7@[1,2] and W7@[1,3]. This scheme can easily accommodate a similar segment in the next tile to the right, connecting E7@[1,3] and W7@[1,4], and can in fact accommodate segments of any complexity. If segments like these occur one hundred times in a device, then one hundred pairs of wire identifiers would have to be stored. This adds up to two hundred wire identifiers.

$$\begin{aligned} \text{segment: } & (E7@[1, 2], W7@[1, 3]) \\ \text{segment: } & (E7@[1, 3], W7@[1, 4]) \\ & \vdots \\ \text{segment: } & (E7@[m, n], W7@[m, n + 1]) \end{aligned}$$

An alternate representation takes advantage of the *shape* retention.⁴ The original segment may be thought to connect E7@[0,0] and W7@[0,1] from a starting offset [1,2]. To represent the similar segment in the next tile to the right, one could simply refer to the same shape, and provide the starting offset [1,3]. Again, assuming that similar segments occurred one hundred times, it would only be necessary to store one pair of wires, and one hundred starting offsets. This adds up to two wire identifiers and one hundred offsets.

³In a few unusual cases a segment may include more than one wire in a single tile.

⁴WireDB used a similar but slightly more restrictive scheme.

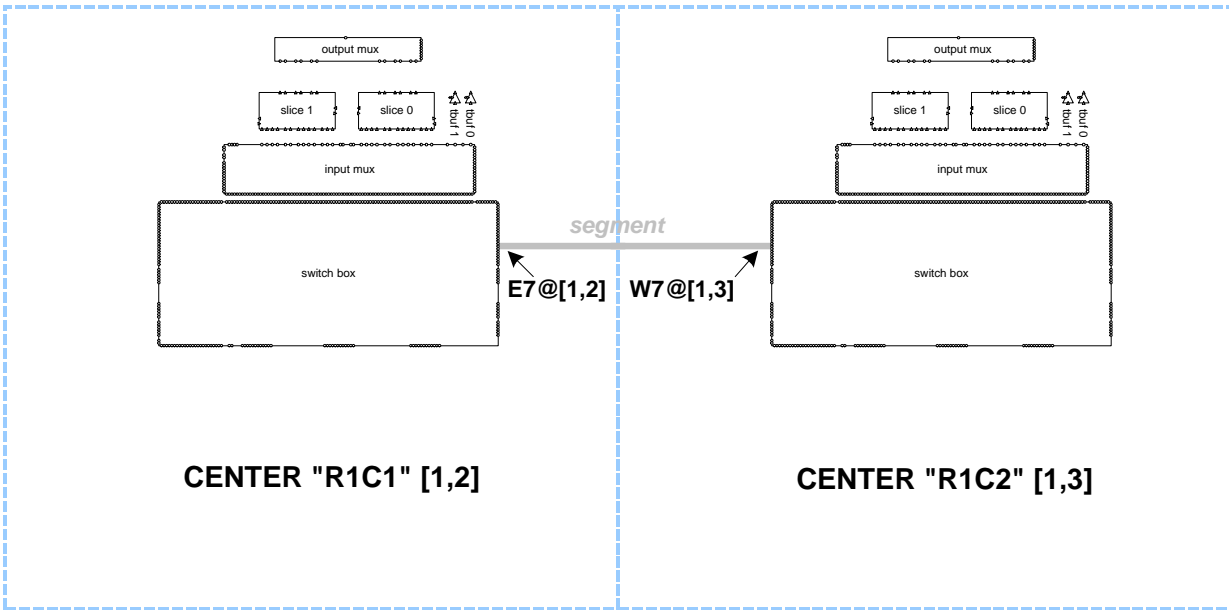


Figure 4.9: Virtex Single segment

segment: $(E7@[0,0], W7@[0,1])$ instantiated at $([1,2], [1,3], \dots, [m,n])$

While these structures are technically *compact segments*, they will generally be called *segments*, because they encode all of the full segment information and because ADB never actually stores full segments.

The exact size reduction depends on the data representation, but for segments which contain more than two wires, like the Virtex *hex* or *long* classes, the compression ratio increases.

Tables 4.6, 4.7, and 4.8 show the results of segment compaction for Virtex, Virtex-E, and Virtex-II data. While no rigorous study was undertaken to establish this scheme as optimal, it is simple, it carries little performance penalty, and its results are quite good.

All but the smallest devices achieve segment compression of 80% or better. This is especially helpful for the larger devices, such as the Virtex-II XC2V8000 whose segment memory footprint drops over 90% from 32,858,964 bytes to 3,187,214 bytes.

The number of compacted segments in a device corresponds to the number of unique *shapes* that segments may take on. This helps explain why the number of compacted segments in a family remains relatively constant even though the number of expanded segments may vary by up to two orders of magnitude.

Table 4.6: Virtex segment compaction

<i>Device</i>	<i>Segment Counts</i>		<i>Size in Bytes</i>		
	<i>Expanded</i>	<i>Compacted</i>	<i>Expanded</i>	<i>Compacted</i>	<i>% Comp.</i>
XCV50	104,339	2,246	660,480	138,026	79.10
XCV100	157,935	2,270	991,928	178,914	81.96
XCV800	1,138,035	2,410	7,098,944	888,058	87.49
XCV1000	1,477,195	2,394	9,182,832	1,127,018	87.73

Table 4.7: Virtex-E segment compaction

<i>Device</i>	<i>Segment Counts</i>		<i>Size in Bytes</i>		
	<i>Expanded</i>	<i>Compacted</i>	<i>Expanded</i>	<i>Compacted</i>	<i>% Comp.</i>
XCV50E	105,947	3,140	684,528	171,450	74.95
XCV100E	159,927	3,056	1,022,232	209,650	79.49
XCV2600E	3,040,135	4,546	19,174,808	2,322,882	87.89
XCV3200E	3,868,867	4,198	24,492,992	2,892,274	88.19

Table 4.8: Virtex-II segment compaction

<i>Device</i>	<i>Segment Counts</i>		<i>Size in Bytes</i>		
	<i>Expanded</i>	<i>Compacted</i>	<i>Expanded</i>	<i>Compacted</i>	<i>% Comp.</i>
XC2V40	39,199	5,780	353,136	204,504	42.09
XC2V80	69,651	6,830	657,332	250,558	61.88
XC2V6000	3,171,411	8,267	24,591,716	2,449,182	90.04
XC2V8000	4,318,059	8,267	32,858,964	3,187,214	90.30

4.9.2 Compacted Segment Example

Consider the segment (E7@[1,2], W7@[1,3]) previously shown in Figure 4.9. On the Virtex XCV50 the following mappings exist:⁵

⁵These mappings reflect internal values, valid for a particular build of the ADB database for Virtex XCV50, and subject to change at any time.

- Tile coordinates [1,2] map to tile index 33
- Tile coordinates [1,3] map to tile index 34
- Tiles 33 and 34 are of type CENTER
- Wire name E7 maps to wire index 29 in CENTER tiles
- Wire name W7 maps to wire index 435 in CENTER tiles
- Segment (E7@[1,2], W7@[1,3]) is encoded in compact segment #1,446 ⁶

Given these mappings, E7@[1,2] can be expressed as tilewire 29@33, and W7@[1,3] can be expressed as tilewire 435@34. This is in fact the way ADB represents these wires internally.

Now compact segment #1,446 is defined as:

#1,446: (29@0, 435@1) *instantiated at* (33, 34, ..., 554)

This shorthand notation actually represents the following:

(29@0, 435@1) *instantiated at* 33 : (29@0 + 0@33, 435@1 + 0@33) → (29@33, 435@34)
 (29@0, 435@1) *instantiated at* 34 : (29@0 + 0@34, 435@1 + 0@34) → (29@34, 435@35)
 ⋮
 (29@0, 435@1) *instantiated at* 554 : (29@0+0@554, 435@1+0@554) → (29@554, 435@555)

Therefore the correct interpretation of compact segment #1,446 is that it represents the following regular segments:

{(29@33, 435@34), (29@34, 435@35), ..., (29@554, 435@555)}

Using wire names and tile coordinates, this also corresponds to:

{(E7@[1,2], W7@[1,3]), (E7@[1,3], W7@[1,4]), ..., (E7@[17,27], W7@[17,28])}

4.9.3 Segment Lookup Tables

The compacted segment information allows ADB to efficiently determine what wires make up a segment when the segment index is already known, but does not allow ADB to easily determine which segment index to use if only the tile and wire index are known. In order to map tile and wire indexes to segment indexes, ADB uses lookup tables.

⁶Those readers who try to relate the compact segment index 1,446 to the tile or wire indexes will be disappointed to find that there is no connection, and no grand conspiracy afoot here.

The lookup tables are implemented as two dimensional arrays, ordered first by tile index, and then by wire index. Because one entry is required for each wire in the entire device, the size of the table may exceed ten million entries, as shown for the Virtex-II XC2V8000 in Table 4.3. Once again the quantity of data involved mandates a compact representation.

ADB represents a lookup table entry as a combination of the segment index and the tile index at which the segment is instantiated. This data is encoded into a 32 bit `int` in a manner similar to the *tilewires* in Section 4.6.4.

Figure 4.10 shows the internal format of a segment lookup entry. The tile index is subject to the same constraints as in the tilewire type, namely a maximum of 65,536 tiles per device. The segment index is restricted to 32,678 unique segment shapes, which well exceeds the maximum of 8,267 for the Virtex-II XC2V8000 in Table 4.8.



Figure 4.10: Segment lookup entry bit usage

The real segment information is in effect split between the compact segment entries and the segment lookup tables. Together these elements allow ADB to translate between wires and segments as needed.

There is a special case which is handled differently than regular segments. *Trivial segments*, as described earlier, consist of exactly one wire, completely contained within a single tile. Such a segment is fully described by its one wire, so there is no need to include it in the compact segment entries. Trivial segments are encoded with the value zero in the segment lookup tables. This encoding results in a smaller memory footprint and in improved performance. While this memory footprint reduction was not analyzed separately, it does figure into sizes shown in tables 4.6, 4.7, and 4.8.

4.9.4 Segment Lookup Table Example

Consider the wire `W7@[1,3]`. Recalling the internal mappings used in Section 4.9.2, this wire can also be expressed as `435@34`, where 435 is the wire index and 34 is the tile index.

To determine what segment wire `435@34` belongs to, ADB looks up the entry for wire 435 in tile 34, and finds `1446@33`:

```

tile 34 :
  ⋮
  wire 435 : 1446@33
  ⋮

```

The entry 1446@33 tells ADB that wire 435@34 belongs to compact segment #1,446, instantiated at tile 33. And returning yet again to the example in Section 4.9.2, ADB finds that segment #1,446 instantiated at tile 33 expands into the real segment:

(*E7*@[1, 2], *W7*@[1, 3])

Therefore by using the compact segment entries and the segment lookup tables, ADB can determine everything that it needs to know about a segment.

4.9.5 Segments Class

Figure 4.11 shows the structure of the `Segments` class. A single `Segments` object exists for the entire database.

ADB.Segments	
⋮	
int	segments[][]
int	tile_segments[][]
RemoteArc	remote_sinks[][]
RemoteArc	remote_sources[][]
RemoteNode	remote_nodes[][]
RemoteNode	local_nodes[][]

Figure 4.11: Segments class

The `segments[][]` array is ordered by compact segment index, and holds an array of tilewire entries for each wire in the segment. These entries represent the possible segment *shapes* as described earlier, but do not indicate where the shapes are instantiated.

The `tile_segments[][]` array is two-dimensional, ordered first by tile index and then by wire index. This array represents the segment lookup tables described earlier, and provides information about where compact segments are instantiated.

The `segments[][]` and `tile_segments[][]` arrays together allow ADB to translate between wires and segments as needed.

The `remote_sinks[][]` and `remote_sources[][]` arrays are ordered by tile index, and hold arrays of `RemoteArc` elements. The `remote_sinks[][]` data allows ADB to determine which remote wires

may be driven by wires in the current tile. The `remote_sources` [] [] data allows ADB to determine which remote wires may drive wires in the current tile.

The `remote_nodes` [] [] and `local_nodes` [] [] arrays are ordered by tile index, and hold arrays of `RemoteNode` elements. The `remote_nodes` [] [] data allows ADB to determine which remote wires may be referenced by wires in the current tile. The `local_nodes` [] [] data allows ADB to determine which remote wires may reference wires in the current tile.

Because of the way in which the BFD data defines nodes and arcs, any arc represented by a `RemoteArc` object will also result in `RemoteNode` objects for the wires that it connects.

Normal routing and tracing functions proceed in a forward direction, from sources to sinks. However, the JBits internal Wire Database API requires the ability to enumerate and work with a wire's sources. The `remote_sources` [] array is provided in order to efficiently support this requirement. The `remote_sources` [] and `local_nodes` [] arrays are not stored on disk but are instead dynamically generated from the `remote_sinks` [] and `remote_nodes` [] arrays when the database is initialized.

4.9.6 Segment Optimization on Disk

A closer look at the compact segment information reveals some redundancy which can be stripped from the database files. Consider compact segment #1,446 used in previous examples:

#1,446: (29@0, 435@1) instantiated at (33, 34, ..., 554)

As demonstrated earlier this can be expanded into:

{(29@33, 435@34), (29@34, 435@35), ..., (29@554, 435@555)}

From this information it is not difficult to generate the following lookup table entries:

```

tile 33 :
    wire 29 : 1446@33
tile 34 :
    wire 29 : 1446@34
    wire 435 : 1446@33
tile 35 :
    wire 435 : 1446@34
    :
tile 554 :
    wire 29 : 1446@554
tile 555 :
    wire 435 : 1446@554

```

In other words, the lookup tables do not need to be stored on disk since they can be generated at runtime. The time required to generate these tables is found to be negligible.⁷

However there are a few types of wires whose lookup table entries cannot be generated automatically. Entries for wires which belong to *trivial segments* cannot be generated since they are not included in the compact segments. But the correct entry for those wires is zero, and if the table entries are pre-initialized to zero, nothing more needs to be done. Entries for remote wires, which are referenced in one tile but reside in another tile, cannot be generated either since the segment information is associated with the tile in which the wire resides. The correct entry for remote wires is -1, but generating these entries dynamically is inefficient.

ADB avoids storing the segment lookup tables on disk. Instead it stores the number of wires in each tile, and stores any special cases which it cannot infer dynamically. This allows it to create and pre-initialize all of the segment lookup tables, and also fill in the special cases before generating the remaining entries.

Because the number of special case entries in the segment lookup tables is small, the disk footprint savings is roughly equal to the memory footprint of the segment lookup tables. This works out to the number of wires in the device, shown in tables 4.1, 4.2, and 4.3, multiplied by 4 bytes per wire.

For the Virtex-II XC2V8000, the segment lookup tables require 10,463,182 wires multiplied by 4 bytes per wire, or 41,852,728 bytes. By contrast the compact segments for the same device only require 3,187,214 bytes. By storing only the compact segment information instead of that plus the segment lookup tables, ADB reduces the disk footprint for the segment information for Virtex-II XC2V8000 by over 90%.

4.10 Resource Usage

ADB tracks resource usage for two reasons. Firstly, it keeps track of the wires and arcs that are in use, so that it can safely route new nets without interfering with existing ones. Secondly, it keeps track of each equation group which was touched by the routing and which will need to be evaluated when the bitstream is updated.

The equation group usage is only updated during routing. The wire and arc usage are updated during routing and during tracing. When ADB routes nets, it is effectively allocating configurable resources, which it must track. When it traces through an existing bitstream it is marking previously allocated resources so that it does not inadvertently reuse them.

A design decision was made to track every resource in a tile, and represent each one with a single bit, rather than tracking only the used resources, and representing each one with a integer index.

⁷The amount of time necessary to generate the tables may actually be less than the amount of time to read them from disk and uncompress them.

In the case of sparse designs this will result in some unnecessary overhead, but in the case of dense designs, where memory may already be a significant issue, the usage can be represented much more efficiently. With this decision ADB aspires to perform well even in very full designs on the largest devices.

For example, each CENTER tile in the Virtex XCV50 contains 462 wires. If each one is indexed and represented with a single bit, the wire usage in each CENTER tile will require roughly 58 bytes. By contrast if each wire in use is represented by a 16 bit index, the same 58 bytes can only track 29 wires. Therefore for center tiles in which more than 29 wires are used, it is more efficient to track all wires with bits. In addition if each resource is tracked in an array, then each one can be indexed directly, but if only used resources are tracked, then some searching and perhaps sorting will be necessary. And furthermore if only used resources are tracked, the usage can be expected to grow as a design is routed, which entails additional memory management.

The resource usage classes take advantage of the `java.util.BitSet` class. The `BitSet` class is extremely efficient for storing large sets of boolean values, and the resource usage classes simply flag used resources as `true` and unused resources as `false`. The resources themselves are mapped to bit indexes in the `BitSet`.

Each resource usage class allows for one `BitSet` per tile, but does not create the `BitSet` for a tile until at least one resource in that tile becomes used. Therefore if a design leaves some tiles unused, no memory overhead is incurred for those tiles. If a particular tile does not have a `BitSet` allocated, then its resources are understood to be unused.

4.10.1 WireUsage Class

Figure 4.12 shows the structure of the `WireUsage` class. A single `WireUsage` object exists for the entire database. This object tracks the usage of every wire in the device.

ADB.WireUsage	
⋮	
int	bit_count
BitSet	tiles[]
⋮	

Figure 4.12: WireUsage class

The `bit_count` field holds the total number of bits tracked by the object. This field equals the number of tiles which have at least one wire in use, multiplied by the maximum number of wires in each of those tiles. The number of wires in a tile depends on the tile type, so different tiles generally have different numbers of wires. This field is used for statistical purposes.

The `tiles[]` sparse array is ordered by tile index, and holds `BitSet` objects. Any tile with no wires used will have the corresponding `tiles[]` entry set to `null`. Wires in `BitSet` objects are addressed by wire index.

4.10.2 ArcUsage Class

Figure 4.13 shows the structure of the `ArcUsage` class. A single `ArcUsage` object exists for the entire database. This object tracks the usage of every arc in the device.

ADB.ArcUsage	
⋮	
int	bit_count
BitSet	tiles[]
⋮	

Figure 4.13: ArcUsage class

The `bit_count` field holds the total number of bits tracked by the object. This field equals the number of tiles which have at least one arc in use, multiplied by the maximum number of arcs in each of those tiles. The number of arcs in a tile depends on the tile type, so different tiles generally have different numbers of arcs. This field is used for statistical purposes.

The `tiles[]` sparse array is ordered by tile index, and holds `BitSet` objects. Any tile with no arcs used will have the corresponding `tiles[]` entry set to `null`.

Arcs in `BitSet` objects are addressed by arc offset. The arc offset is a sequentially assigned index for each arc in a tile type.

For example, consider wire 29 in tile 33. The `WireInfo` object for this wire indicates that its `offset` value is 65 and its `sinks` array contains (193, 289, 435). The `offset` 65 indicates that 65 arcs occur in wires 0 through 28. Wire 289 is the second wire in the `sinks` array, so its zero-based index is 1. Therefore, if an arc is made from wire 29 to wire 289, ADB calculates its offset as $65 + 1 = 66$, and bit 66 in the `BitSet` object for tile 33 will be set. If this is the first arc used in tile 33, its `BitSet` object will first be created.

4.10.3 GroupUsage Class

Figure 4.14 shows the structure of the `GroupUsage` class. A single `GroupUsage` object exists for the entire database. This object tracks every equation group in the device that needs to be evaluated when updating the bitstream.

The `bit_count` field holds the total number of bits tracked by the object. This field equals the number of tiles which have at least one group to be evaluated, multiplied by the maximum number

ADB.GroupUsage	
⋮	
int	bit_count
BitSet	tiles[]
⋮	

Figure 4.14: GroupUsage class

of groups in each of those tiles. The number of groups in a tile depends on the tile type, so different tiles generally have different numbers of groups. This field is used for statistical purposes.

The `tiles[]` sparse array is ordered by tile index, and holds `BitSet` objects. Any tile with no groups used will have the corresponding `tiles[]` entry set to `null`.

Groups in `BitSet` objects are addressed by group offset. The group offset is a sequentially assigned index for each group in a tile type. ADB performs a binary search on the tile's `groups[]` array in the `Tiles` object to identify the group offset.

ADB updates the bitstream by evaluating the marked equation groups in each tile which has a non-`null` `tiles[]` entry. This delayed updating approach avoids situations where equation groups are evaluated more than once, but most importantly defers the equation evaluation until the entire design is complete. Many BFD equations require knowledge of other parts of the design, so if such equations are evaluated too early, they may produce incorrect bitstreams.⁸ The `GroupUsage` tracking avoids that situation.

4.11 Database

The database consists of the tile and segment information, remote resource information, BFD equations, and resource usage information. But in good object-oriented fashion, the database also includes all of the functionality necessary to work with this information and to trace or route any design.

4.11.1 Equations Class

The `Equations` class holds groups of forward and reverse BFD equations for the current family. Forward equation groups are used to modify the bitstream when routing, and reverse equation groups are used to analyze the bitstream when tracing.

⁸In particular, the Virtex and Virtex-E BFD equations make connections between certain kinds of unused wires to force them into stable states. If left floating, these wires may cause destructive oscillation within the device.

Equations are invoked by group index, where the group indexes correspond to the numbers used in the `Tiles` and `WireInfo` classes.

The `Equations` class is read from the appropriate family database file, with the help of a special `java.lang.ClassLoader` subclass. The equation groups themselves are implemented as Java functions, derived in direct or inverted form from the BFD equations.

4.11.2 BitServer Interface

The `BitServer` interface provides access to configuration bits in a bitstream. Because ADB does not read and write bitstreams directly, an external bit server is required.

ADB includes an implementation of this interface in class `ADB.JBits.BitServer`, which provides configuration bit access for Virtex and Virtex-II families through `JBits`.

This interface exists primarily for the benefit of families which are not supported by `JBits`, including Virtex-E, as described below. Figure 4.15 shows the `BitServer` interface declaration.

```

1 package ADB;
2
3 public interface BitServer {
4     // set bits specified by bits[] in tile [row,col] to values val[]
5     abstract public void setTileBits(int row, int col, int bits[][], int val[])
6         throws Exception;
7     // return bits specified by bits[] in tile [row,col]
8     abstract public int[] getTileBits(int row, int col, int bits[][])
9         throws Exception;
10 }
```

Figure 4.15: BitServer interface

The two functions declared are `setTileBits()` and `getTileBits()`. These functions use the same form as their `JBits` counterparts, but differ in their interpretation of `row`. ADB considers row zero to be at the top of the device, as the Xilinx Foundation tools also do. This differs from the `JBits` convention of placing row zero at the bottom of the device. Class `ADB.JBits.BitServer` includes translation to account for this difference.

The `row` and `col` fields are the coordinates of the tile in which the configuration bits reside.

The `bits[][]` array is a list of configuration bit row and column pairs. The `val[]` array is a list of values to which the configuration bits should be set.

It is possible for the user to provide an alternate bit server. For example an experimental Virtex-E bit server was derived from JBits Virtex code, even though JBits does not support Virtex-E. Using the Virtex-E bit server with ADB allows it to route or trace any Virtex-E design.

4.11.3 Router Class

The `Router` class provided with ADB is an implementation of the GRAPHSEARCH algorithm adapted from Nilsson [37]. This implementation provides a directed search, guided by an appropriate heuristic.

Family-specific heuristics for the Virtex, Virtex-E, and Virtex-II are provided in internal classes `ADB.Virtex.Heuristic`, `ADB.VirtexE.Heuristic`, and `ADB.Virtex2.Heuristic`.⁹

It is possible for the user to provide an alternate router or an alternate heuristic.

4.11.4 Tracer Class

The `Tracer` class provided with ADB traces bitstream nets from sources to sinks.

In addition to providing information about the routing inside a design, this class also updates wire and arc usage information. This allows ADB to route in an existing design without interfering with existing nets.

4.11.5 DB Class

Figure 4.16 shows the structure of the DB class. A single DB object exists for the entire database. This object owns the tile and segment information, tracks every configurable wiring resource, and provides routing and tracing services.

4.11.6 ADB Package

Figure 4.17 shows the structure of the ADB package with its main classes. A variety of internal fields and classes have been omitted for clarity.

One of the classes omitted from Figure 4.17 is the `Browser` class. This class provides a simple console interface which allows the user to manually explore device wiring.

⁹These classes are integrated into the family database files.

ADB.DB	
Segments	segments
Tiles	tiles
Equations	equations
WireUsage	wire_usage
ArcUsage	arc_usage
GroupUsage	group_usage
BitServer	bit_server
Router	router
Tracer	tracer
	⋮

Figure 4.16: DB class

4.11.7 Database Files

ADB maintains the intrinsic tile type paradigm of the FPGAs by separating the family data from the device data. The family data is common to every device in the family, and consists primarily of information about the tile types. The device data, on the other hand, applies only to the current device, and must be available for each device which ADB is expected to support. The device data consists primarily of segment information, and remote node and arc information, since this information varies from one device to the next.

This approach allows the creation of a single database file for each supported FPGA family, and one database file for each supported device in the family. Because the family data is fairly large, keeping it separate from the device data results in significantly smaller device databases. Nonetheless the device and family data are dependent on each other, and whenever ADB opens a device database, it also loads the appropriate family database.

The databases used by each of the supported families are as follows:

Virtex: Virtex.db, XCV50.db, XCV100.db, XCV150.db, XCV200.db, XCV300.db, XCV400.db, XCV600.db, XCV800.db, XCV1000.db

Virtex-E: VirtexE.db, XCV50E.db, XCV100E.db, XCV200E.db, XCV300E.db, XCV400E.db, XCV405E.db, XCV600E.db, XCV812E.db, XCV1000E.db, XCV1600E.db, XCV2000E.db, XCV2600E.db, XCV3200E.db

Virtex-II: Virtex2.db, XC2V40.db, XC2V80.db, XC2V250.db, XC2V500.db, XC2V1000.db, XC2V1500.db, XC2V2000.db, XC2V3000.db, XC2V4000.db, XC2V6000.db, XC2V8000.db

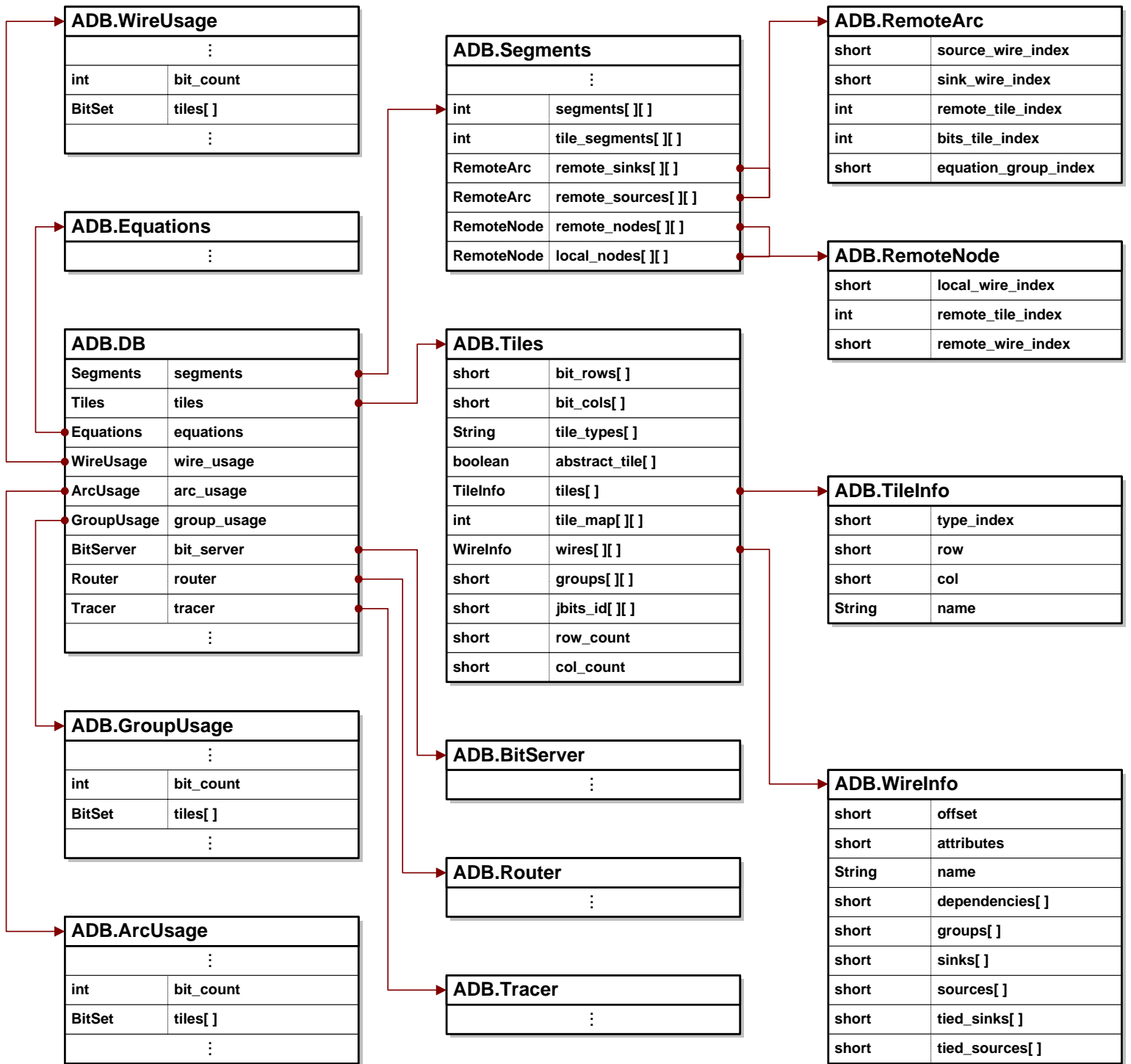


Figure 4.17: ADB Package

The contents of the family database files are:

- Family name
- Tile type names
- Information for each wire in each tile type, including attributes, names, equation group indexes, arcs, and dependencies
- Compiled forward and reverse BFD equations for all tile types
- Compiled default routing heuristic for family

The contents of the device database files are:

- Name of family database file
- Tile map, including tile names, tile type indexes, and tile coordinates
- Remote wires which reside in other tiles
- Remote arcs which are controlled by configuration bits in other tiles
- Compact segment information, including segment shapes and tiles where they are instantiated
- Segment exceptions, including cases where the segment of a particular tilewire cannot be automatically inferred

Both the family and device database files are compressed in ZIP format, and read by ADB with the help of the `java.util.zip.ZipInputStream` class [32].

4.12 JBits Support

One of the primary applications of ADB is to support routing or tracing for JBits users. In order to do this, ADB must interface with various JBits classes, hopefully in a manner that is largely transparent to JBits users.

The architecture of JBits, along with the security features of Java, preclude an interface as simple as one might like. In most cases ADB must subclass key JBits classes. In addition it must deal with the fact that JBits is strongly tied to the FPGA family, so a number of classes have to be replicated for each family that ADB supports.

Many of these issues could have been resolved very simply with the multiple inheritance available in C++. Java offers the **interface** paradigm to take the place of multiple inheritance, but since an **interface** contains no code [32], it does not eliminate the need for multiple classes.¹⁰

A variety of features require ADB to subclass more than it would like to. For example ADB doesn't update the configuration bits until the bitstream is ready to be written to disk. In order to perform this delayed updating in a transparent manner, ADB must provide subclasses of the `JBits` class for each supported family.

JBits support is provided through four packages: `ADB.JBits`, `ADB.Virtex`, `ADB.VirtexE`, and `ADB.Virtex2`.

4.12.1 ADB.JBits Package

The `ADB.JBits` package contains functionality common to all supported ADB families. Its classes are as follows:

BitServer: Uses `JBits` to get or set configuration bits from the current bitstream

Browser: Subclasses `ADB.Browser` to allow opening a specified bitstream and tracing through it at startup

DB: *Abstract* Subclasses `ADB.DB` to work around certain Java security issues, convert between ADB and `JBits` tile coordinates, and provide various other common services

Lookup: *Abstract* Supports `JBits Wire Database` interface by translating `JBits Pin` objects into corresponding `Wire` objects

PinInterface: *Interface* Interface shared by family-specific `Pin` classes

Router: Subclasses `ADB.Router` and implements the `JBits RouterInterface` interface, to make ADB look like a generic `JBits` router

Wire: *Abstract* Supports `JBits Wire Database` interface by implementing the required functions

Apart from the `Browser` class, which may be used to manually explore device wiring, none of these classes would normally be instantiated or subclassed by a `JBits` user.

¹⁰Java traditionally contends that multiple inheritance is burdensome; there is a reasonable and demonstrable counter-argument that it does not have to be. Some suggest “inheritance-by-composition” as an alternative, but such a scheme is neither transparent to the user nor easy to implement without design-time control of each class or tree involved. And the dismissive argument that “I’ve never used multiple inheritance in C++, so it must not be useful” presumably reflects lack of exposure to proper uses of multiple inheritance.

4.12.2 ADB.[family] Packages

The family packages are `ADB.Virtex`, `ADB.VirtexE`, and `ADB.Virtex2`. Nearly all of these files are generated from templates by the database build code. Each family includes the following classes:

DB: Subclasses `ADB.JBits.DB` to enable delayed bit updating

Equations: Subclasses `ADB.Equations` to support family BFD equations; *this class is bundled into the family database file*

Heuristic: Subclasses `ADB.Heuristic` to support family routing heuristics; *this class is bundled into the family database file*

JBits: Subclasses the appropriate `JBits` class to initialize ADB on startup and to ensure delayed bit updating

Lookup: Subclasses `ADB.JBits.Lookup` to return `Wire` objects of the appropriate family

Pin: Subclasses `JBits.Pin` objects to encapsulate ADB and `JBits` style coordinates for improved performance

Wire: Subclasses `ADB.JBits.Wire` to provide `JBits` wire IDs for each wire in the family

It should be noted that the `ADB.VirtexE` package is not currently useable because `JBits` does not support the Virtex-E family. While ADB knows all that it needs to know about the Virtex-E family, there is no way of interfacing it with `JBits`, and thus the Virtex-E is currently only supported by ADB in standalone mode.

The `JBits` user will generally create a `JBits` object for the appropriate family, in order to initialize both `JBits` and ADB. The user may then create `Pin` objects with the wire ID constants provided in the `Wire` class. These `Pin` objects may be used for routing or may be translated into `Wire` objects by the `Lookup` class, in support of the `JBits` Wire Database interface.

Chapter 5

Database Build

5.1 Overview

The database design presented in Chapter 4 shows how ADB uses its data internally. However the internal representation differs significantly from that of the source data presented in Chapter 3, and the conversion from source data to internal representation is decidedly non-trivial.

The conversion from source data to internal representation is called the *build process*. The build process is responsible for generating the family and device database files, and the family-specific Java class files for each supported family. This process draws on source data, family-specific exception and heuristic code, Java template files, and external tools. Figure 5.1 shows an overview of the build process.

The build code exists entirely as a collection of Perl scripts. The choice of Perl is based on a number of advantages that the language presents, and on a strong track record with Perl in similar contexts.

Some of the advantages of Perl are that 1) it has robust *regular expression* support for search and replace functions, 2) it works very well with large files, 3) it has built-in support for hash tables, 4) it is an interpreted language, allowing short debug cycles, 5) it allows runtime execution of arbitrary interpreted code, 6) it is robust and fast, and 7) it is reasonably cross-platform compatible. One notable disadvantage of Perl is its inelegant support of object oriented features.

The terms *local*, *global*, and *remote* are used frequently in this chapter, and the reader is cautioned that the term *local* may be used in contrast to the term *global* or to the term *remote*. In the first case the distinction is between an identifier used only in the current tile type and the corresponding identifier used throughout the rest of the device. In the second case the distinction is between a resource which exists in the current tile and a resource which exists in another tile. While the

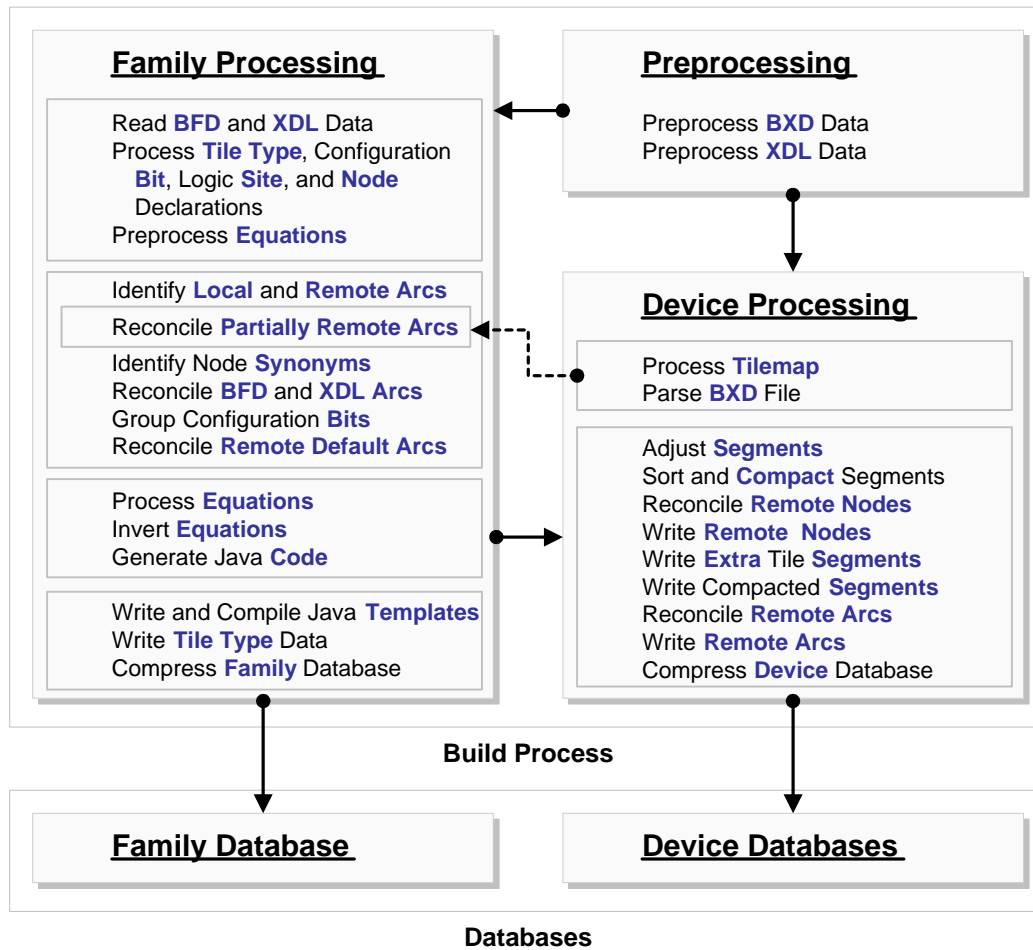


Figure 5.1: Build process roadmap

context should reduce the confusion, it can still be perplexing to think about the *local* name in a *remote* tile, or the *global* name in the *local* tile. *Caveat lector*.

The reader may also note the build code's very stringent handling of arcs. This is warranted by the many different classes of arcs which exist, including some that are easy to miss altogether. The build code processes arcs very meticulously in order to ensure that each one is properly accounted for, and the inability to account for any arc is considered a fatal error.¹

¹The build code is found to be robust, and yet by the end of this chapter the reader may be inclined to agree with the author's assessment that it's a miracle any of it works at all. The overly enthusiastic reader is referred to the source code, which may be easier to understand than this chapter.

5.2 Preprocessing

A few aspects of the build process are handled as preprocessing steps, either in order to reduce memory requirements, or simply because it is convenient to separate them from the rest of the process. The BXD preprocessing in particular can be *very* memory intensive.

In many cases the preprocessing steps were executed only a few times during development, while the remaining steps required a very large number of iterations to identify and handle the various exceptions that turned up.

5.2.1 BXD Preprocessing

The BXD files available with the source data are actually proprietary Xilinx binary files whose format is unknown. Fortunately the internal Xilinx tool *batest* provides text dumps of the data. This process is automated through a Perl script which generates a text dump of the BXD data and a corresponding device *tilemap*. Because the BXD data is device-specific, the script includes the ability to batch-process all devices from all supported families.

Although the tilemap information is ready for immediate use by the build code, the BXD data requires one more preprocessing step.

The BXD data includes embedded segment information, but not in a readily useable form. Segments are inferred by grouping all the wires that share the same coordinates. The grouping process essentially builds a large hash table as it proceeds through the file, with unique wire coordinates serving as hash keys. Each unique set of coordinates is also given a sequential segment index, and the coordinates are replaced by the segment index in the resulting output file.

Figures 5.2 and 5.3 show a fragment of BXD data before and after preprocessing. For example all wires which share coordinates (-5852, -596), such as wire CIN_0 in line 1, are given segment index 20311.

When the regular build code processes this data, it can simply assemble segments on the basis of the segment index. Since the BXD files can reach many hundreds of megabytes in size, this preprocessing step can substantially reduce the memory burden placed on the regular build code.

5.2.2 XDL Preprocessing

To simplify the regular build code, the XDL data for each family is preprocessed and distilled into a single *.xdlrules* file.

```

1 BX_WIRE: "CIN_0" (-5852, -596)
2 BX_WIRE: "CIN_1" (-5986, -596)
3 BX_WIRE: "CO_0" (-5852, -132)
4 BX_WIRE: "CO_0_LOCAL" (-5852, -592)
5 BX_WIRE: "CO_1" (-5986, -132)
6 BX_WIRE: "CO_1_LOCAL" (-5986, -592)
7 BX_WIRE: "EO" (-5695, -704)
8 BX_WIRE: "E1" (-5695, -702)
9 BX_WIRE: "E10" (-5695, -684)
10 BX_WIRE: "E11" (-5695, -682)

```

Figure 5.2: Standard BXD

```

1 BX_WIRE: "CIN_0" #20311
2 BX_WIRE: "CIN_1" #20312
3 BX_WIRE: "CO_0" #20313
4 BX_WIRE: "CO_0_LOCAL" #20314
5 BX_WIRE: "CO_1" #20315
6 BX_WIRE: "CO_1_LOCAL" #20316
7 BX_WIRE: "EO" #20317
8 BX_WIRE: "E1" #20318
9 BX_WIRE: "E10" #20319
10 BX_WIRE: "E11" #20320

```

Figure 5.3: Preprocessed BXD

The XDL data is *flat*, *tile*-based, and *device*-specific, but must be reconciled with BFD data that is *hierarchical*, *tile type*-based, and *family*-specific. In addition the XDL files can approach 2 GB in size for the largest devices, while the information that needs to be extracted is less than 1 MB in size.

As explained in Chapter 3, the XDL data is generated by the Xilinx *xdl* tool used in *-report* mode. This tool can generate XDL data with or without *pip* (or *arc*) information, but including the pip information results in files that are roughly 200 times larger than they would be otherwise. For all combined devices in the Virtex, Virtex-E, and Virtex-II families, the data would approach 10 GB in size.

Identifying Minsets

The XDL data is used for its inherent *tile type* information, so a reasonable optimization is to look at only enough devices to describe all the tile types in a family. Such a subset of devices sufficient to describe each tile type in the family is called a *minset*.² And therefore it is only necessary to look at the pip information for the devices in the minset. The use of minsets allows the build code to neatly distill many gigabytes of data into less than one megabyte.

However, identifying a family's minset requires at least a brief look at all of the devices, to determine which ones contain which tile types. The preprocessing code therefore generates XDL data *without* pip information for all devices in a family, identifies a minset which comprises all the tile types, and then generates XDL data *with* pip information for the devices in the minset.³

²As explained previously, this term is meant to denote a *small* subset of devices which includes all tile types, but not necessarily the *smallest possible* such subset. Defining the smallest possible subset would be computationally burdensome, and the entire reason for using a minset is to reduce the computational burden.

³The reader may note that the device tilemaps derived from the BXD data should be enough to define minsets, since those tilemaps specify tile types. This approach would eliminate the need to generate XDL data *without* pip information for all the devices in a family. The reasons for the adopted approach are largely historical, stemming from the fact that the XDL data was still being evaluated and explored while ADB development occurred, and that database *build* time is relatively inconsequential as long as it doesn't adversely affect database *execution* time.

It is important to note that the minset cannot possibly describe *abstract* tile types, since abstract tile types are by definition those *not included in any device*.

Figure 5.4 shows the minsets obtained for Virtex, Virtex-E, and Virtex-II:

Virtex: (XCV400)
Virtex-E: (XCV50E, XCV405E)
Virtex-II: (XC2V500)

For example, in order to extract information for each tile type in the Virtex-E family, it is necessary to process XDL data with pip information for the XCV50E and the XCV405E devices.

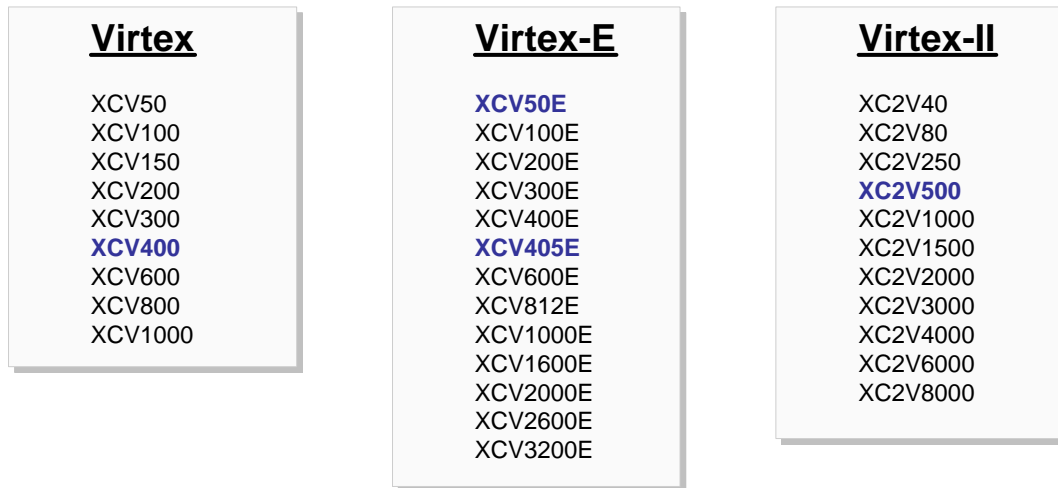


Figure 5.4: Family minsets

Extracting Tile Type Information

The build code processes the XDL data for each device in the minset, and classifies all tile information according to the tile type in which it appears. Along with each tile type name, the names of the devices in which the tile type occurs are also retained. The *tile type to device name* association plays an important part in later BFD processing.

Along with each tile type, the build code includes information about logic sites, site inputs and outputs, and arcs:

Sites: Includes the *site type* name, and each encountered *site* name

Inputs and Outputs: Includes the wire name and categorizes it as an input or output

Arcs: Includes the source wire name and all possible sinks that it can drive

While the XDL data uses global wire names, ADB uses only local wire names. However the BFD data includes both local and global names, so the necessary mapping can be accomplished during the regular build process.

Once the tile type data for an entire family has been extracted, it is stored in a single *.xdlrules* file for later use by the regular build code.

5.3 Family-Specific Data

As evidenced in Chapter 3, the source data that ADB uses is divided into family-specific and device-specific data. This scheme provides a number of benefits for the source data, including simplified syntax and reduced data footprint, and is adopted directly into ADB's design.

Working against this straightforward approach is the added complexity which surfaces in a few places along the family–device interface. The problem is that some of the family data cannot be fully understood without the context of a specific device. And though the build code tries to shoulder as much of this complexity as it can, ADB is still appreciably affected by it. This family–device interface complexity results in a small fraction of the data being responsible for a grossly disproportionate amount of code and build time.

The family data is derived from the BFD and XDL data, though the build code must occasionally consult the BXD data for the reasons just described. While the BFD equations are an integral part of the family data, they constitute an entire topic by themselves, and will be discussed in a separate section.

Because the family data generally serves as the basis for interpreting the device data, the family data must be processed before any devices can be properly considered.

With the exception of the separate equation discussion noted above, the following sections are described in the order in which they occur during the build process.

5.3.1 XDL Processing

The XDL data is used by the build code to supplement the BFD data. It assists the build code in 1) identifying *fake arcs*, 2) determining correct arc directions, 3) identifying site input and output wires, and 4) mapping site names to enclosing tile types.

The build code deals with the preprocessed form of the XDL data available through *.xdlrules* files. This data is read and parsed, and stored in a number of hash tables. The hash tables indicate 1) which devices belong to the minset, 2) which devices each tile type occurs in, 3) what site types and site names occur in each tile type, 4) how site names map to tile type names, 5) which wires are site inputs or outputs, and 6) which sinks any given wire can drive.

Once the data has been stored in hash tables, the `correct_xdl_data()` function from the family script is invoked. For Virtex and Virtex-E this function does nothing. For Virtex-II it resolves synonym ambiguities and inserts arcs which are absent from the XDL data.

5.3.2 Preliminary BFD Processing

The build code reads the BFD data as a set of lines. It then combines the lines into a single string, strips embedded comments, converts each sequence of spaces, tabs, or line breaks into a single space, and divides the string into tile type declarations. At this point the BFD string consists of a set of lines, each of which declares one tile type.

The `correct_bfd_lines()` function from the family script is invoked to apply any necessary corrections to the BFD data. For Virtex and Virtex-II this consists of inserting node declarations for a few wires that appear in the BXD data but not in the BFD data.⁴ For Virtex-E it consists of correcting a few typographical errors and removing some duplicate node declarations.⁵

It is important to note that these corrections are necessary *for the manner in which the ADB build code* uses the BFD data, and not necessarily *for the manner in which the Xilinx Foundation tools* use it. In particular it is not reasonable to conclude from this that the Xilinx Foundation tools conceal flaws which could result in bitstream errors.

5.3.3 Processing Tile Type Declarations

The build code extracts the tile type name, parent tile type name, number of configuration bit rows and columns, and number of grid rows and columns from each tile type declaration. The tile type name is always present. The parent tile type name is only present if the tile type inherits from another tile type. The remaining data may or may not be present in any given tile type declaration, but the configuration bit rows and columns must be declared at least once for each tile type that is instantiated.

A tile type index is assigned to each declared tile type. This index is used by the build code to store all of the tile type's information. It is also used by the ADB code and databases because it is more compact and easier to search by than the tile type name.

⁴Inserting these nodes is necessary to support the build code's requirement that the BFD data identify *all* nodes.

⁵The typographical errors show up in global node names, and prevent the build code from correctly reconciling BFD and BXD data.

For each tile type that inherits from a parent, the build code copies all of the parent's information into the child tile type. This requires that parent tile types be declared before any child attempts to reference them, which is found to be the case in BFD files. Any declarations included in the child tile type override the parent's data.

The build code uses the XDL data to determine if the tile type ever occurs in a real device. If it does not, the tile type is flagged as abstract.

The remaining tile type data is divided into sections. These sections may include configuration bit declarations, logic site declarations, node declarations, and equation declarations.

5.3.4 Processing Configuration Bits

Configuration bits are declared in rows or columns of arbitrary length, with the constraint that all required bits be declared at least once for concrete tile types. Each configuration bit is given a name or declared to be unused. Unused bits are those for which there is no defining BFD equation.

Each configuration bit declaration specifies a row or column address. When a row address is provided, the bit names correspond to coordinates $([row, 0], [row, 1], \dots, [row, n])$. When a column address is provided, the bit names correspond to coordinates $([0, col], [1, col], \dots, [m, col])$.

Because the BFD equations reference configuration bits by name, a hash table is generated to map bit names to coordinates.

5.3.5 Processing Logic Sites

Logic site declarations are used to describe local sites which exist in the current tile type, or to reference sites elsewhere in the device which have unique global names. Local sites are of no use to the build code, but global sites are important because they provide a mapping from local site name to global site name.

Global site names allow the build code to associate certain kinds of remote nodes with specific device tiles, most typically clock and DLL tiles.

5.3.6 Remote Node Discussion

Nodes (or wires) may be local or remote with respect to the tile type in which they are declared. Local nodes exist within the declared tile type (and implicit tile), while remote nodes exist in other tiles, which may or may not be of the same tile type.

In the case of remote nodes, the BFD data specifies additional information to identify the tiles in which the nodes reside. Unfortunately the additional information is device-dependent and cannot be fully resolved without the help of the BXD data.

Remote nodes are specified in one of two ways:

Relative Offset Tile: Node resides in the tile at offset $[row,col]$ from the current tile

Site Anchored Tile: Node resides in the tile which contains logic site *site*

In the case of relative offset tiles, the build code must either evaluate the relative offset from each affected tile in the device or rely on a heuristic in order to determine the remote tile type.

In the case of site anchored tiles, the build code relies on the XDL data to map the site name to the enclosing tile type.

For the sake of processing the family data, the build code does not need to fully resolve remote nodes, but it does need to determine which tile types the remote nodes reside in. For relative offset tiles the build code invokes the `get_relative_remote_tile_type()` function from the family script to determine what tile type the node is associated with. For Virtex, Virtex-E, and Virtex-II there are only one or two possible remote tile types implicated in relative offset nodes. The use of this function significantly reduces processing complexity and sidesteps circular dependencies in the data.

During device processing all remote nodes will indeed have to be fully resolved. But the simplification described above defers that requirement long enough for all family processing to complete.

Once the remote tile type is known, the build code can use that tile type's node mapping to translate node names into node indexes.

5.3.7 Remote Node Examples

Remote nodes are easier to understand with examples. This section provides an example of a remote node referenced to a relative tile offset, and another example of a remote node referenced to a site name.

Relative Offset Remote Node Example

Figure 5.5 presents a fragment of the Virtex BFD declaration for tile type `CENTER`.

Lines 2 through 4 declare nodes which reside in the remote tile one column to the right in the tile map (indicated by `Nodes Tile row=0 col=1`), while lines 5 through 7 declare nodes which reside in the current tile (indicated by `Nodes`).

```

1 | Tile "CENTER" rows=18 cols=48 grows=2 gcols=3 {
  |   :
2 |     Nodes Tile row=0 col=1 {
3 |       TBUF3_R1 = TBUF3,
  |       :
4 |     }
5 |     Nodes {
  |       :
6 |       TBUF3 = TBUF3,
  |       :
7 |     }
  |     :
8 | }

```

Figure 5.5: Virtex CENTER tile type fragment

Consider the node declared in Line 3. This node has the local name `TBUF3_R1` and the global name `TBUF3`. In other words tile type `CENTER` refers to this node as `TBUF3_R1` but the `BXD` and `XDL` data refer to it as `TBUF3`. Furthermore because this is a remote node, the tile one column to the right must also declare a node with global name `TBUF3`.

Determining which tile exists one column to the right of the current one requires access to the device-specific tile map. In fact since a device generally contains hundreds or thousands of `CENTER` tiles, the remote node has to be resolved for each tile individually. This is indeed how the build code resolves the remote nodes when processing device data, but a simpler approach is very desirable at this stage.

The build code invokes the `get_relative_remote_tile_type()` function from the family script, and passes it the local tile type name, the local node name, the remote node name, and the offset row and column. This function only returns the tile type in which the remote node occurs, which is a simplification, but one sufficient for the immediate needs of the build code.

In this example, the `get_relative_remote_tile_type()` function returns tile type `CENTER`. This means that while the local and remote tiles are distinct from one another, they both happen to be of type `CENTER`.

The build code now knows to look for global node name `TBUF3` in tile type `CENTER`. It readily finds the node declared in Line 6, with local name `TBUF3` and global name `TBUF3`. While the local and global names are identical in this case, they are not required to be.

Site Anchored Remote Node Example

Figure 5.6 presents a fragment of the Virtex BFD declaration for tile type BRAM_TOP.

```

1 | Tile "BRAM_TOP" rows=18 cols=91 grows=2 gcols=0 {
2 |     Sites {
3 |         DLL3 sitename=DLL3,
4 |         :
5 |     }
6 |     Nodes Site DLL3 {
7 |         DLL_CLKPADO = CLKT_CLKPADO,
8 |         :
9 |     }
10| }

```

Figure 5.6: Virtex BRAM_TOP tile type fragment

Line 3 indicates that the site locally referred to as DLL3 also has the global site name DLL3. While the names are identical in this case, they are not required to be.

Lines 5 through 7 declare nodes which reside in the remote tile which contains site DLL3 (indicated by `Nodes Site DLL3`).

Consider the node declared in Line 6. This node has the local name `DLL_CLKPADO` and the global name `CLKT_CLKPADO`. In other words tile type `BRAM_TOP` refers to this node as `DLL_CLKPADO` but the `BXD` and `XDL` data refer to it as `CLKT_CLKPADO`. Furthermore because this is a remote node, the tile which contains site `DLL3` must also declare a node with global name `CLKT_CLKPADO`.

Now with the help of the `XDL` data, the site named `DLL3` is found to reside in *the only instance* of tile type `CLKT`. Figure 5.7 presents a fragment of tile type `CLKT`.

Consider the node declared in Line 3, with local name `CLKPADO` and global name `CLKT_CLKPADO`. The matching global name `CLKT_CLKPADO` in both the `BRAM_TOP` and `CLKT` tile types indicates that local name `DLL_CLKPADO` in tile type `BRAM_TOP` corresponds to local name `CLKPADO` in tile type `CLKT`.

The full reconciliation of these remote nodes does not occur until the device processing stage, however the arc-related code will need hints before that time, as evidenced by other examples.

The discussion now returns to the regular build process.

```

1 | Tile "CLKT" rows=20 cols=8 grows=2 gcols=0 {
  |   :
  |   :
2 |   Nodes {
  |     :
  |     :
3 |     CLKPADO = CLKT_CLKPADO,
  |     :
  |     :
4 |   }
  |   :
  |   :
5 | }

```

Figure 5.7: Virtex CLKT tile type fragment

5.3.8 Processing Nodes

Nodes are processed by the same function whether they are local or remote. Each node is declared with a local name and a global name. The local name is used in the tile type's BFD equations, while the global name is used in the BXD and XDL data. Only the local names are retained by ADB.

The build code assigns a wire index to each node that it encounters. These wire indexes are used by the ADB code and databases.

If a local name is already defined through inheritance, the build code maps the local name to the previously assigned wire index, and the occurrence is logged.

If the local name is not defined, but the global name is already defined through inheritance, then the build code maps the local name to the previously assigned wire index, and the occurrence is logged. This situation only occurs with Virtex-II.

If neither the local nor the global name is already defined, the build code maps the local name to the next sequential index as the wire index.

If the node resides in the current tile type, the build code maps the global name to the same wire index as the local name, and gathers attribute information. The build code checks the XDL data to see if the wire should be classified as a site input or output. It also invokes the `node_is_hidden()` function from the family script to determine if the node should be omitted from traces.

If the node resides in a remote tile, the build code maps the local name to the remote data. The remote data consists of the specified global node name (which also exists as a global node name in a remote tile) and the data which identifies the remote tile. As explained earlier, the data which identifies the remote tile will either be a site name or a relative offset.

Each unique local node name in the entire family is assigned a JBits wire ID number. The node name and the wire ID are included in the `Wire.java` file, and are provided to give JBits users a way of communicating with ADB. As explained in Section 4.7.3, ADB wire indexes only have meaning within the context of a tile or tile type, but JBits wire IDs are constant throughout the entire family. However, while a JBits user may specify any wire ID in any tile, ADB will throw an exception if no matching wire exists in that tile.

5.3.9 Preprocessing Equations

Although the bulk of the equation processing is deferred until a later step, the build code does some preliminary processing when it first examines the equations.

The equations are split into configuration bit names and right-hand-side expressions, and the `filter_bfd_expression()` function from the family script is invoked on the expressions. For Virtex, which shares a BFD file with Spartan-II, BFD functions specific to Spartan-II are set to `FALSE`. This effectively allows a single BFD file to be preprocessed for Virtex use or for Spartan-II use. For Virtex-E and Virtex-II this function does nothing.

Because ADB is only interested in equations that relate to wires or arcs, all expressions that do not reference arcs are ignored. In addition, expressions that include `config()` functions are ignored.⁶

A small volume could be written about BFD equations and their intricacies, but since the underlying data is proprietary Xilinx information, it cannot be discussed here.

The expressions that remain are stored in a hash table and keyed by configuration bit name. They are then modified so that wire names are changed to wire indexes in all `arcval()`, `arcinv()`, and `nodeused()` functions, as shown in figures 5.8 and 5.9.

```

1 # I3 multiplexer
2 I76.I63.I1 = arcval (V6N1, I3) || arcval (V6M1, I3);
3 I76.I63.I11 = arcval (V6N1, I3) || arcval (V6A1, I3) || arcval (V6B1, I3);
4 I76.I63.I7 = arcval (V6A1, I3) || arcval (V6C1, I3);
5 I76.I63.I8 = arcval (V6B1, I3) || arcval (V6D1, I3);

```

Figure 5.8: Regular BFD equations

Finally all remaining arcs are recorded, and associations are set up from arc to bit name, and from bit name to arc. These associations will be used in subsequent processing steps.

⁶This behavior is a little controversial, but the assumption is that if an equation contains `config()` functions, then it probably controls something other than a mux. (There are a few possible reasons why this may occur in an equation.) All equations ignored in this manner are logged, and the logs give no indication that important wire or arc information is being overlooked.

```

1 # I3 multiplexer
2 I76.I63.I1 = arcval (42, 2) || arcval (38, 2);
3 I76.I63.I11 = arcval (42, 2) || arcval (22, 2) || arcval (26, 2);
4 I76.I63.I7 = arcval (22, 2) || arcval (30, 2);
5 I76.I63.I8 = arcval (26, 2) || arcval (34, 2);

```

Figure 5.9: Preprocessed BFD equations

5.3.10 Remote Arc Discussion

The build code needs to know whether arcs are local, remote, or partially remote. It determines this by examining the nodes which make up the arcs, and classifies each arc in the following manner:

Local Arc: Both nodes reside in the local tile

Remote Arc: Both nodes reside in a remote tile

Partially Remote Arc: One node resides in the local tile and one node resides in a remote tile

Remote arcs and partially remote arcs deserve some additional explanation. One might ask if any remote arc ever connects wires from two different tiles, and the answer is that *that situation never occurs in the Virtex, Virtex-E, or Virtex-II families*. Certainly for wires which belong to non-trivial segments, one can argue that an arc between them effectively connects wires from many different tiles. But the important thing from the perspective of the database is that *every* arc can be defined as a connection between two local wires in *some* tile.

In fact a remote arc is best defined as one *whose configuration bits reside in a different tile than the arc itself*. From a routing perspective one might more accurately speak of *remote bits* than *remote arcs*. However since the primary purpose of the BFD file is to generate bitstream information, it chooses to define all configuration bits as local, relegating arcs to remote status when necessary.

The allusion to segments also helps to shed some light on partially remote arcs. Whenever an arc connects a local wire in tile *local* and a remote wire in tile *remote*, it can be shown that the local wire in tile *local* belongs to the same segment as another wire in tile *remote*. In other words with some extra effort it is possible to define the arc as a connection between two different wires in tile *remote*, with configuration bits in tile *local*.

Each partially remote arc that the build code identifies will later be resolved into a fully remote arc, with the help of device segment data. Fortunately the segment data can be generalized enough that only one of the miniset devices needs to be examined during family processing. This allows all of the family processing to occur before the device processing begins, which conveniently sidesteps circular dependencies in the data.

Some of these facts about remote and partially remote arcs were not understood when development began, which explains why ADB still allows for remote arcs which connect wires in two different tiles, even though the build code no longer does. As explained earlier, ADB's functionality was retained for the eventuality of future expansion.

5.3.11 Remote Arc Examples

Like remote nodes, remote arcs are easier to understand with examples. This section provides an example of a fully remote arc and an example of a partially remote arc. Both of these arcs are defined in tile type BRAM_TOP, and both of them connect wires in tile type CLKT.

Figure 5.10 presents a fragment of the Virtex BFD declaration for tile type BRAM_TOP.

```

1 | Tile "BRAM_TOP" rows=18 cols=91 grows=2 gcols=0 {
2 |     Sites {
3 |         DLL3 sitename=DLL3,
4 |         :
5 |     }
6 |     Nodes Site DLL3 {
7 |         DLL_CLKPADO = CLKT_CLKPADO,
8 |         :
9 |         CLKT_CLK2XL = CLKT_CLK2XL,
10 |        :
11 |        CLKT_GCLKBUF1_IN = CLKT_GCLKBUF3_IN,
12 |        :
13 |    }

```

Figure 5.10: Virtex BRAM_TOP tile type fragment

Line 3 indicates that the site locally referred to as DLL3 also has the global site name DLL3. By consulting the XDL data, the build code is able to determine that site DLL3 resides in *the only instance* of tile type CLKT. This is no different than the remote node example.

Lines 5 through 9 declare nodes which reside in the remote tile which contains site DLL3 (indicated by `Nodes Site DLL3`), which was just determined to be tile type `CLKT`. Lines 10 through 12 declare nodes which reside in the current tile (indicated by `Nodes`).

Since the remote nodes reside in tile type `CLKT` they can only be reconciled with the help of that tile type declaration. Figure 5.11 presents a fragment of the declaration for tile type `CLKT`.

```

1 | Tile "CLKT" rows=20 cols=8 grows=2 gcols=0 {
  | :
  | :
2 |   Nodes {
  |     :
  |     CLK2XL = CLKT_CLK2XL,
  |     :
  |     CLKINL = CLKT_CLKINL,
  |     CLKINR = CLKT_CLKINR,
  |     CLKPADO = CLKT_CLKPADO,
  |     :
  |     GCLKBUF1_IN = CLKT_GCLKBUF3_IN,
  |     :
  |   }
  | :
  | :
9 | }

```

Figure 5.11: Virtex `CLKT` tile type fragment

Lines 2 through 8 declare nodes which reside in the current tile (indicated by `Nodes`).

From these two tile type fragments it is possible to equate the following wire names (the numbers to the left of the local names are the wire indexes assigned by the build code):

#	<i>BRAM_TOP</i> Local Name	#	<i>CLKT</i> Local Name	Global Name
0	DLL_CLKPADO	20	CLKPADO	CLKT_CLKPADO
2	CLKT_CLK2XL	8	CLK2XL	CLKT_CLK2XL
3	CLKT_GCLKBUF1_IN	27	GCLKBUF1_IN	CLKT_GCLKBUF3_IN
7	CLKIN		<i>n/a</i>	BRAM_TOP_CLKIN
	<i>n/a</i>	18	CLKINL	CLKT_CLKINL
	<i>n/a</i>	19	CLKINR	CLKT_CLKINR

Wires `CLKIN`, `CLKINL`, and `CLKINR` are included here in anticipation of the solution, but no relationship between them can be defined at this point.

Fully Remote Arc Example

Tile type BRAM_TOP defines an arc between local wires CLKT_CLK2XL and CLKT_GCLKBUF1_IN:

Local Wire Names: CLKT_CLK2XL \rightarrow CLKT_GCLKBUF1_IN *Wire Indexes:* 2 \rightarrow 3

The build code determines that these nodes are remote and that they reside in tile type CLKT. Using the local wire names in tile type CLKT, the arc can be expressed as:

Local Wire Names: CLK2XL \rightarrow GCLKBUF1_IN *Wire Indexes:* 8 \rightarrow 27

The build code understands that the arc connects wires CLK2XL and GCLKBUF1_IN in tile type CLKT, but that the arc is controlled by configuration bits in tile type BRAM_TOP.

Partially Remote Arc Example

Tile type BRAM_TOP also defines an arc between local wires DLL_CLKPADO and CLKIN:

Local Wire Names: DLL_CLKPADO \rightarrow CLKIN *Wire Indexes:* 0 \rightarrow 7

In this case DLL_CLKPADO is a remote node which resides in tile type CLKT, but CLKIN is a local node in tile type BRAM_TOP, so the arc does not adhere to proper form.

Further analysis reveals that local wire CLKIN in tile type BRAM_TOP belongs to a segment which extends to tile type CLKT, and therefore it should be possible to re-express CLKIN by the local name that its segment takes on in tile type CLKT.

In order to follow the segment which CLKIN belongs to, it is necessary to decide *which of the two* BRAM_TOP tiles in the device to use. In fact the correct answer is that the analysis must be performed on *both* BRAM_TOP tiles, for reasons which will become clear shortly.

In the Virtex miniset device XCV50, the two BRAM_TOP tiles have tile indexes 1 and 29, and the CLKT tile has tile index 15.

The first case begins with wire index 7 at tile index 1, for which the following segment information exists:

(7@1, 5@2, 5@3, 5@4, 5@5, 5@6, 5@7, 1@8, 5@9, 5@10, 5@11, 5@12, 5@13, 5@14,
18@15)

The build code looks at each wire in the segment, searching for one which occurs in tile 15, and finds 18@15. Now wire index 18 in tile index 15 has the local name CLKINL, so this partially remote arc can be expressed in tile type CLKT as:

Local Wire Names: CLKPADO → CLKINL *Wire Indexes:* 20 → 18

The second case begins with wire index 7 at tile index 29, for which the following segment information exists:

(19@15, 5@16, 5@17, 5@18, 5@19, 5@20, 5@21, 1@22, 5@23, 5@24, 5@25, 5@26, 5@27, 5@28, 7@29)

The build code looks at each wire in the segment, again searching for one which occurs in tile 15, and finds 19@15. Now wire index 19 in tile index 15 has the local name CLKINR, so this partially remote arc can *also* be expressed in tile type CLKT as:

Local Wire Names: CLKPADO → CLKINR *Wire Indexes:* 20 → 19

In fact the correct interpretation of this partially remote arc is that it represents *two* arcs in tile type CLKT, one controlled by configuration bits in BRAM_TOP tile 1, and the other controlled by configuration bits in BRAM_TOP tile 29:

Controlled by BRAM_TOP tile 1:

Local Wire Names: CLKPADO → CLKINL *Wire Indexes:* 20 → 18

Controlled by BRAM_TOP tile 29:

Local Wire Names: CLKPADO → CLKINR *Wire Indexes:* 20 → 19

Fully Remote Arc Addendum

The fact that a partially remote arc in tile type BRAM_TOP resulted in two arcs in tile type CLKT may leave the reader wondering why this isn't a consideration for the fully remote arc presented earlier. The answer is that fully remote arcs define both remote nodes unambiguously. In contrast, partially remote arcs only define one of the remote nodes unambiguously, and leave the other one dependent on segment information.

The discussion now returns from examples back to the build code processing.

5.3.12 Identifying Local and Remote Arcs

Once the build code has completed initial processing for all of the tile types, it begins to address the many unresolved issues. The first step is to analyze every arc in every tile type and classify it as local, remote, or partially remote.

Extracting Remote Node Information

For every remote node, the build code attempts to determine the remote tile type name, the remote node name, and the remote node index.

If the remote node resides in a site anchored tile, the build code uses the XDL data to determine the tile type. If the remote node resides in a relative offset tile, the build code invokes the `get_relative_remote_tile_type()` function from the family script to determine the tile type.

Once the tile type is known, the build code looks up the remote tile type's node mapping. With the help of the node mapping, it converts the global node name to a wire index in the remote tile type, and it also determines the local wire name in that tile type.

Classifying Arcs

The build code determines whether an arc's nodes are remote by checking to see if remote node information is available for them. It then classifies the arc as local, remote, or partially remote as described earlier.

If the arc is local, an entry is made in the local arcs hash table for the local tile type, unless the reverse arc has already been defined.⁷

If the arc is remote, an entry is made in the local tile type pointing to the remote arc, and another entry is made in the remote tile type pointing back to the local arc. This effectively says that 1) there are configuration bits in the current tile type which control an arc in a remote tile type, and that 2) there is an arc in a remote tile type controlled by configuration bits in the current tile types. While this may appear to be redundant, it simplifies processing considerably.

If the arc is partially remote, the build code looks for a minset device which contains both the local and remote tile types. It then pushes the arc's information into that device queue to be reconciled later.

At this point the build code knows which tile type each arc resides in, though it still lacks some of the wire name and index information about partially remote arcs.

5.3.13 Reconciling Partially Remote Arcs

When all of the partially remote arcs have been identified, the build code sets out to reconcile them into fully remote arcs. Unfortunately this step requires BXD segment information, and the

⁷Arc directions will be reconciled with the help of the XDL data, but this cannot be accomplished until all remote arcs have been assigned to the proper tile types. It is simpler for the build code to discard reverse arcs at this stage.

convenient separation between family and device processing lapses briefly. Purists will however be relieved to hear that the segment information is obtained from the regular device processing code.

The build code processes all of the minset devices which were assigned partially remote arcs in the previous step. For each of these devices, the build code processes the device tile map and the device BXD file. All of the segment information is extracted from the BXD file, and is made available to the build code.

For each partially remote arc, the build code begins by identifying all of the tiles which match the local and remote tile type names. It is assumed that only one tile matches the remote tile type name, and if this assumption fails, the failure is logged as an error.

The partially remote arc must be reconciled for each tile which matches the local tile type name. If the build code did not enforce this, it would fail to define certain arcs, as demonstrated in earlier examples.

The build code begins by looking up the node mappings for the remote tile and for each admissible choice of local tiles, and uses this information to obtain the appropriate wire indexes. The wire indexes can then be used to search the segment information, allowing the build code to identify the wire name and index in the remote tile which correspond to the wire in the local tile.

The partially remote arc is now re-expressed as a fully remote arc, and is stored in the same manner as the other fully remote arcs. An entry is made in the local tile type pointing to the remote arc, and another entry is made in the remote tile type pointing back to the local arc.

5.3.14 Searching for Node Synonyms

In general a segment includes exactly one wire for each tile that it spans, however Virtex-II contains a small number of exceptions to this rule. This situation occurs when a segment leaves a tile, and later passes through it again, such that the segment has two valid node names in that particular tile.

The build code could reasonably ignore this situation, except for the fact that arcs which include these nodes are denoted differently by the BFD and XDL data. Because of this, the build code is unable to account for the arcs, which violates its requirement that all arcs be properly accounted for.

The build code scans the segment information for node synonyms, and simply reports any synonyms that it encounters as reminders to the developer. Resolving these problems is left to the `correct_xdl_data()` function from the family script, as described earlier.

5.3.15 Rewriting XDL Arcs With Local Indexes

The XDL arcs are expressed in terms of global node names, while the BFD arcs are expressed in terms of local node names. The build code resolves this difference by converting local or global wire names into wire indexes, and so the XDL arc information is converted in place from global node names to wire indexes.

5.3.16 Arc Classes Discussion

Although ADB only deals with local arcs, fake arcs, and remote arcs, the build code faces a much more complex situation because of the way the source data is presented. At build time, each arc belongs to one of the following classes.

Trimmed Arc (not in XDL): Not described in XDL, sometimes because of errors, but usually because the arc connects to an unused logic site. Always explicitly confirmed with Xilinx.

Local Arc: Described in both BFD and XDL. Arc and configuration bits both belong to current tile.

Remote Arc: Described in both BFD and XDL. Arc and configuration bits belong to different tiles.

Trimmed Arc (not in BFD): Not described in BFD, typically because the arc connects to an unused logic site. Always explicitly confirmed with Xilinx.

Default Arc: Described only in XDL because no BFD equation exists. The BFD equations assume the bitstream is initialized to zero, and therefore equations are only provided for bits that must be set to one. A default arc exists when none of the multiplexer's other arcs exist. Arc and implicit configuration bits both belong to current tile.

Fake Arc: Described only in XDL because no configuration bits are involved. Arc belongs to current tile, *and is always on*.

Remote Default Arc: Described only in XDL because no BFD equation exists. Similar to a default arc, but belongs to remote tile.

5.3.17 Reconciling BFD and XDL Arcs

The rules for classifying arcs are complex and error prone, so the build code keeps a list of all known arcs, removing each one as it is accounted for. Arcs which remain on the list at the end are treated as fatal errors.

The build code begins by gathering references to the current tile type's data, and gathering all of the BFD and XDL arc information. The arcs are then processed in the following order:

Trimmed Arc (not in XDL)

All BFD arcs are passed to the `arc_is_trimmed_from_xdl()` function in the family script. Each arc for which the function returns `TRUE` is removed from the BFD data.

Arcs in this category typically connect to unused logic sites, and while there may be BFD equations and configuration bits associated with these arcs, nothing is lost by discarding them.

At this point every arc remaining in the BFD data also exists in the XDL data.

Local Arc

All BFD arcs are passed to an internal function which compares them with the XDL data. The function indicates whether the arc occurs in the forward direction, in the reverse direction, or in both directions, and removes the indicated arcs from the XDL data.

Confirming arc directions is a necessary step because the BFD data does not reliably include that information. The function mentioned above may respond that 1) the arc exists as specified, 2) the arc exists, but is backwards, or 3) the arc exists as specified, and the reverse arc also exists.

Remote Arc

All remote BFD arcs which reside in the current tile type are passed to an internal function which compares them with the XDL data. These are arcs which were defined in other tile types, but included in the current tile type during previous steps. Each one is classified as forward and/or reverse, and is removed from the XDL data.

The remote arcs processed here have to be associated with *each* of the tile types in which their configuration bits reside.⁸

The remote arcs are processed after the local arcs, in order to give the local arcs higher priority. This is necessary for cases in which an arc is controlled by the local tile type, but is also referenced by other tile types.⁹

⁸This is necessary in the case of some Virtex CENTER tile arcs for instance, because they may be controlled by bits in tile types CENTER, LEFT, LEFT_PCLTOP, or LEFT_PCLBOT, depending on their position in the device tile map.

⁹Other tile types may indeed include equations which reference the arc in the current tile type, but these equations are merely “sniffing out the routing” for other purposes, and are not actually controlling the arc.

At this point all of the arcs in the BFD data have been accounted for. All remaining arcs originate with the XDL data, and are therefore explicitly directed arcs, so there is no more need to check for forward and reverse directions.

Trimmed Arc (not in BFD)

All remaining XDL arcs are passed to the `arc_is_trimmed_from_bfd()` function in the family script. Each arc for which the function returns TRUE is removed from the XDL data.

Arcs in this category typically connect to unused logic sites. Because the BFD equations do not include these arcs, there are no configuration bits to control them, and nothing is lost by discarding them.

While this step may appear to be unnecessary, it actually plays an important part in removing arcs which would hamper the remaining steps.

Default Arc

The build code examines each remaining XDL arc and determines which ones belong to multiplexers. An arc is deemed to belong to a multiplexer if its sink wire can be driven by more than one source wire.

Every remaining XDL arc which belongs to a multiplexer is considered to be a default arc.¹⁰ The premise is that if all of the multiplexer's other wires were properly handled as local or remote arcs, then the remaining one must exist in the absence of all others, and is therefore a default arc. These arcs are removed from the XDL data after being processed.

At this point all arcs which involve configuration bits have been accounted for.

Fake Arc

All remaining arcs are considered to be fake arcs, since they involve no configuration bits. These arcs are known to be forward arcs since they come directly from XDL data, and they complete this stage of the reconciling process.

¹⁰The build code does check for one particularly problematic condition. If more than one of the multiplexer's arcs remains in the XDL data, then each of those arcs would appear to be a default arc. This condition is considered an error because it is not possible for a multiplexer to have more than one default arc.

Remote Default Arc

Remote default arcs are very complicated to reconcile, and are therefore processed separately from other arcs. A subsequent section describes this process.

5.3.18 Grouping Configuration Bits

Configuration bits frequently operate in groups in order to control multiplexers. In all such cases, the build code is responsible for determining how these groups are composed, and it does this by referring to hash tables set up during the preliminary equation processing.

The hash tables in question map bit names to arcs and arcs to bit names. The build code simply picks the first bit that it encounters, and recursively absorbs all associated arcs and bits into a group. The process ends when no bits remain in the original hash table and all configuration bits are organized into groups.

For example, when the equations presented in Figure 5.8 are originally processed, the build code generates the following hash entries:

<i>Bit Name</i>	<i>Arcs</i>
I76.I63.I1:	(V6N1→I3, V6M1→I3)
I76.I63.I11:	(V6N1→I3, V6A1→I3, V6B1→I3)
I76.I63.I7:	(V6A1→I3, V6C1→I3)
I76.I63.I8:	(V6B1→I3, V6D1→I3)

<i>Arc Name</i>	<i>Bits</i>
V6A1→I3:	(I76.I63.I11, I76.I63.I7)
V6B1→I3:	(I76.I63.I11, I76.I63.I8)
V6C1→I3:	(I76.I63.I7)
V6D1→I3:	(I76.I63.I8)
V6M1→I3:	(I76.I63.I1)
V6N1→I3:	(I76.I63.I1, I76.I63.I11)

This example is a little contrived because it only includes bit names and arcs which are already known to form a single group, but it still demonstrates the general process. Beginning with bit I76.I63.I1, and absorbing bits and arcs recursively, the process looks like this (elements already processed are colored in light gray):

```

I76.I63.I1: (V6N1→I3, V6M1→I3)
  V6N1→I3: (I76.I63.I1, I76.I63.I11)
    I76.I63.I11: (V6N1→I3, V6A1→I3, V6B1→I3)
      V6A1→I3: (I76.I63.I11, I76.I63.I7)
        I76.I63.I7: (V6A1→I3, V6C1→I3)
          V6C1→I3: (I76.I63.I7)
            V6B1→I3: (I76.I63.I11, I76.I63.I8)
              I76.I63.I8: (V6B1→I3, V6D1→I3)
                V6D1→I3: (I76.I63.I8)
          V6M1→I3: (I76.I63.I1)

```

When this process completes, the four bits `I76.I63.I1`, `I76.I63.I11`, `I76.I63.I7`, and `I76.I63.I8` are merged into a single group, and the build code understands that these configuration bits and their respective equations must be processed together. If any other bit names remained in the tile type, they would be processed in a similar manner.

5.3.19 Reconciling Remote Default Arcs

Remote default arcs are essentially a combination of remote arcs and default arcs, but considerably more difficult to reconcile. The general approach is to identify default arcs in the current tile type, and determine if they belong to remotely defined multiplexers.

The build code begins by flagging all local sinks and remote sinks in the current tile type. It then loops through each source wire and each of the source wire's sink wires. In the process, it skips 1) arcs which are not marked as defaults, 2) sinks which are marked local, and 3) sinks which are not marked remote. For any remaining arcs, it notes the number of remote references in other tile types, in order to account for each remotely defined arc.

The build code looks up the necessary data for each remote tile type in which the arcs may be defined, and then begins scanning each remote arc for instances which point back to the current tile type.¹¹ For every such arc, the build code attempts to match the source and sink wire indexes that it began with in the current tile type.

Whenever a remotely defined arc is determined to match a default arc in the current tile type, the appropriate mappings are made, and the number of remote references is decremented. When the number of remote references reaches zero the arc is deleted from the current tile type.

¹¹This is somewhat more complex than one might guess since remote arcs would only refer to the current tile type indirectly through global site names or relative offset coordinates.

5.3.20 Processing Equations

Processing the BFD equations is the next sequential step that the build code undertakes, however the processing is complex and forms a large subject all by itself. Details are provided in Section 5.4.

5.3.21 Writing Java Templates

ADB must include a variety of Java classes for each family that it supports. These classes serve to 1) interface between ADB and JBits, and to 2) encapsulate the family's heuristics and BFD equations.

The build code prepares substitution strings ahead of time, and then processes each template file by replacing markers with the appropriate substitution strings. The template files and necessary substitutions are as follows:

DB.java: family name used in package name

Equations.java: family name used in package name; forward and reverse BFD equations and *pseudo-jump* tables included for bitstream support

Heuristic.java: *no substitution performed; included with other files for simplicity*

JBits.java: family name used in package name and in references to JBits classes

Lookup.java: family name used in package name

Pin.java: family name used in package name and in references to JBits and ADB classes

Wire.java: family name used in package name; wire ID constants provided for ADB access by JBits users

The template files are read, customized, and written to the appropriate ADB family directory.

5.3.22 Compiling Java Code

Once the template files have been customized and copied into the appropriate directories, they are compiled into Java *.class* files.

The *pseudo-jump* tables provide single entry points which transfer control to the appropriate equation groups. Because Java does not support function pointers, the jump tables are implemented with large switch statements. Unfortunately, the IBM 1.3.0 Java compiler under Linux was unable

to compile the large jump tables, and so the build code explicitly invokes the Sun 1.3.0_02 Java compiler.¹²

5.3.23 Writing Tile Type Data

The tile type data consists of the following:

Tile Type Name: *self-explanatory*

Abstract Flag: Identifies abstract tile types

Node Information: *for each node*

Node Attributes: Flags signaling input, output, or hidden status

JBits Wire ID: *self-explanatory*

Node Name: *self-explanatory*

Fake Arcs: List of permanent sink connections

Arcs & Groups: List of regular sinks and corresponding equation groups to evaluate; remote arc placeholders are also included

Node Dependencies: List of equation groups which must be re-evaluated if this node becomes used or unused

Exhaustive Groups: List of all equation groups referenced by all nodes in the tile type

This data is written for each tile type in the family.

5.3.24 Writing Compiled Code

The family database requires the tile type data that was just processed, as well as the compiled BFD equations and default routing heuristic.

The build code reads the compiled *.class* files for the equations and heuristic classes, and writes them in the family database.

¹²The IBM compiler appears to support a maximum of 999 cases inside *switch* statements. This appears to be a cumulative limit for all of the switch statements in a class, and therefore efforts to split the jump tables into smaller switch statements failed to solve the problem.

5.3.25 Database Compression

The resulting database file is compressed in ZIP format, which the `java.util.zip.ZipInputStream` class can read. The component sizes and compression statistics are presented in Table 5.1.

Table 5.1: Family database compression

<i>Family</i>	<i>Component Size in Bytes</i>			<i>Total Size in Bytes</i>		
	<i>Tile Types</i>	<i>Equations</i>	<i>Heuristic</i>	<i>Uncompressed</i>	<i>Compressed</i>	<i>% Comp.</i>
Virtex	179,906	1,355,898	3,129	1,538,933	451,467	70.66
Virtex-E	267,810	1,526,415	3,130	1,797,355	506,925	71.80
Virtex-II	543,697	2,399,269	1,065	2,944,031	791,486	73.12

The compiled BFD equations account for over 80% of the uncompressed database sizes. This could be reduced with generated or handcoded C or assembly, but the benefit of reduced size would probably be outweighed by the runtime overhead of interfacing ADB with non-Java code. However no detailed analysis has been performed, so a better solution may exist.

This concludes the family database build, except from the equation processing which remains to be discussed.

5.4 Bitstream Equations

The primary purpose of BFD files is to assist the Xilinx Foundation *bitgen* tool in translating logic and routing configurations into bitstreams, as is quite evident from the composition of the files. At the heart of BFD files are the BFD equations, which perform a one-way mapping from design to configuration bits.¹³ The equations for a tile type are presumably evaluated all at one time by *bitgen*.

ADB also deals with bitstreams, but its needs are somewhat different. Instead of only mapping routing resources to bitstreams, it must also map bitstream information back to routing resources. In addition, it only evaluates the subset of equations necessary for the resources that have changed.

In order to perform the forward mapping from design to bits, ADB can use some form of the BFD equations. But to perform the reverse mapping from bits to design, ADB needs to derive the inverse of the function that the BFD equations represent. This derivation can be quite complex, particularly without the right tools for the job.

The build code translates groups of BFD equations into Java code, which can be compiled and included in the ADB databases. It also inverts these groups of equations, and translates the resulting functions into Java code for the databases.

¹³These equations should not be confused with LUT expressions.

5.4.1 Equation Group Example

The build code does a considerable amount of processing on BFD equation groups, beginning in earlier steps which perform node mapping and which infer multiplexer groupings. Consider the fairly simple equation group presented in Figure 5.12.

```

1 | I76.I63.I1 = arcval(V6N1,I3) || arcval(V6M1,I3);
2 | I76.I63.I11 = arcval(V6N1,I3) || arcval(V6A1,I3) || arcval(V6B1,I3);
3 | I76.I63.I7 = arcval(V6A1,I3) || arcval(V6C1,I3);
4 | I76.I63.I8 = arcval(V6B1,I3) || arcval(V6D1,I3);

```

Figure 5.12: Sample BFD equation group (raw form)

This equation group represents a simple multiplexer, which may look familiar to the reader since it was also presented in Figure 5.8. This is essentially how the equations appear in the BFD file, except for some formatting changes.

Internal encodings in both ADB and the build code use wire indexes and bit coordinates instead of wire names and bit names. In addition, the build code attaches directionality information to arcs, so each arc direction is explicitly defined. The resulting internal representation is shown in Figure 5.13.

```

1 | [4,48] = arcval(42,2,TRUE) || arcval(38,2,TRUE);
2 | [4,9] = arcval(42,2,TRUE) || arcval(22,2,TRUE) || arcval(26,2,TRUE);
3 | [4,6] = arcval(22,2,TRUE) || arcval(30,2,TRUE);
4 | [4,5] = arcval(26,2,TRUE) || arcval(34,2,TRUE);

```

Figure 5.13: Sample BFD equation group (internal representation)

The build code needs to translate this into a usable Java function, with the help of some support functions. The support functions implement `arcval()` to evaluate arc status, and `setTileBits()` to set configuration bits. This particular equation group is given the group index 770 and its forward implementation is named `f770()`, as shown in Figure 5.14.

Lines 2 through 7 check and cache the status of each referenced arc. This avoids redundantly evaluating potentially expensive functions. Lines 8 through 12 evaluate each applicable bit equation and cache the results as integers. And lines 13 and 14 call `setTileBits()` to modify the configuration bits.

The `arcval()` and `setTileBits()` functions included here are provided by the Equations class, and are redirected to the appropriate ADB or BitServer (typically JBits) implementations.

```

1  protected void f770() throws Exception {
2      boolean arcval42_2_true = arcval(42,2,true);
3      boolean arcval38_2_true = arcval(38,2,true);
4      boolean arcval22_2_true = arcval(22,2,true);
5      boolean arcval26_2_true = arcval(26,2,true);
6      boolean arcval30_2_true = arcval(30,2,true);
7      boolean arcval34_2_true = arcval(34,2,true);
8      int bit_4_48 = (arcval42_2_true|arcval38_2_true) ? 1 : 0; // I76.I63.I1
9      int bit_4_9 = (arcval42_2_true|arcval22_2_true|arcval26_2_true) ? 1 : 0;
10     // I76.I63.I11
11     int bit_4_6 = (arcval22_2_true|arcval30_2_true) ? 1 : 0; // I76.I63.I7
12     int bit_4_5 = (arcval26_2_true|arcval34_2_true) ? 1 : 0; // I76.I63.I8
13     setTileBits(tileRow,tileCol,new int[] [] {{4,48},{4,9},{4,6},{4,5}},
14               new int[] {bit_4_48,bit_4_9,bit_4_6,bit_4_5});
15 }

```

Figure 5.14: Sample BFD equation group (Java implementation)

In addition to converting the forward function to Java, the build code must also invert the function. Figure 5.15 shows the inverted function, where the arcs are defined in terms of configuration bits.

```

1  arcval_42_2_TRUE = [4,48] & [4,9];
2  arcval_26_2_TRUE = [4,5] & [4,9];
3  arcval_22_2_TRUE = [4,6] & [4,9];
4  arcval_34_2_TRUE = [4,5] ^ ([4,5] & [4,9]);
5  arcval_30_2_TRUE = [4,6] ^ ([4,6] & [4,9]);
6  arcval_38_2_TRUE = [4,48] ^ ([4,48] & [4,9]);

```

Figure 5.15: Sample inverted equation group (raw form)

As the inversion code proceeds, it sometimes replaces common sub-expressions with previously obtained results. An example of this is shown in Figure 5.16, where lines 4, 5, and 6 reuse results obtained in lines 1, 2, and 3.

```

1  arcval_42_2_TRUE = [4,48] & [4,9];
2  arcval_26_2_TRUE = [4,5] & [4,9];
3  arcval_22_2_TRUE = [4,6] & [4,9];
4  arcval_34_2_TRUE = [4,5] ^ arcval_26_2_TRUE;
5  arcval_30_2_TRUE = [4,6] ^ arcval_22_2_TRUE;
6  arcval_38_2_TRUE = [4,48] ^ arcval_42_2_TRUE;

```

Figure 5.16: Sample inverted equation group (enhanced form)

The build code needs to translate the inverse functions into Java code just as it did for forward equations. The reverse implementation of the sample equation group is named `r770()`, as shown in Figure 5.17.

```

1  protected void r770() throws Exception {
2      int values[] = getTileBits(tileRow,tileCol,new int[][] {{4,48},{4,9},{4,6},{4,5}});
3      boolean bit_4_48 = values[0] == 1; // I76.I63.I1
4      boolean bit_4_9 = values[1] == 1; // I76.I63.I11
5      boolean bit_4_6 = values[2] == 1; // I76.I63.I7
6      boolean bit_4_5 = values[3] == 1; // I76.I63.I8
7      boolean arcval42_2_true = bit_4_48 & bit_4_9;
8      boolean arcval26_2_true = bit_4_5 & bit_4_9;
9      boolean arcval22_2_true = bit_4_6 & bit_4_9;
10     boolean arcval34_2_true = bit_4_5 ^ arcval26_2_true;
11     boolean arcval30_2_true = bit_4_6 ^ arcval22_2_true;
12     boolean arcval38_2_true = bit_4_48 ^ arcval42_2_true;
13     setarc(42,2,true,arcval42_2_true);
14     setarc(26,2,true,arcval26_2_true);
15     setarc(22,2,true,arcval22_2_true);
16     setarc(34,2,true,arcval34_2_true);
17     setarc(30,2,true,arcval30_2_true);
18     setarc(38,2,true,arcval38_2_true);
19 }

```

Figure 5.17: Sample inverted equation group (Java implementation)

Line 2 calls `getTileBits()` to lookup the values of all applicable configuration bits. Lines 3 through 6 cache the integer bit values as boolean values.¹⁴ Lines 7 through 12 evaluate and cache the reverse equations with the help of the boolean bit values. And lines 13 through 18 set the applicable arcs to their inferred states.

The `getTileBits()` and `setarc()` functions included here are provided by the Equations class, and are redirected to the appropriate ADB or BitServer (typically JBits) implementations.

The inversion code relies heavily on exclusive-OR logic, recognizable by the `^` operator used in the last three figures.

The discussion now returns to the build code processing.

¹⁴Java can perform bitwise operations on `int` types, but can only perform boolean operations on `boolean` types. In particular it is not possible to mix and match `int` and `boolean` types in boolean operations.

5.4.2 Equation Conditioning

Before considering the equations, the build code makes a list of all local and remote default arcs for each tile type. It keeps these arcs in a hash table in order to insert them into the regular BFD equations where appropriate. This is necessary because default arcs never appear in BFD equations.

It is important to realize that at this stage the tile type data contains fully explicit arc directions, but the BFD equations do not. In order to resolve that situation, the build code iterates through each configuration bit and processes its corresponding expression. Any arc for which the family's `arc_is_trimmed_from_xdl()` function returns true is replaced with zero. All other arcs are re-expressed with explicit direction information if they are unidirectional, or without direction information if they are truly bidirectional.

Once the arc directions have been determined, any default arc which shares a sink node with the arcs in the expression is associated with the configuration bit. The build code then collects all of the configuration bits and expressions which belong to the predetermined equation groups, and combines them into a string.

The equation group strings are looked up as hash table keys. If a match is found, then an identical equation group has already been processed, so the build code simply reuses the previous equation group index and avoids additional processing. Otherwise if no match is found, the equation group is assigned a new equation group index and added to the hash table, and is then passed along to the Java code generation function.

Table 5.2 shows that over 40% of the equation groups in a family are redundant and can be discarded. This is a significant fact since the compiled equation classes were found to account for over 80% of the family database file size. Most of the redundancy stems from similarities between tile types, particularly when inheritance comes into play.¹⁵

Table 5.2: Equation group compression

<i>Family</i>	<i>Equation Group Counts</i>			
	<i>Total</i>	<i>Duplicates</i>	<i>Remaining</i>	<i>% Comp.</i>
Virtex	3,649	1,496	2,153	41.00
Virtex-E	5,223	2,975	2,248	56.96
Virtex-II	3,656	1,835	1,821	50.19

Now that the index of each equation group is known, all explicit and default arcs that are referenced in the group are associated with the group index. This arc-to-group index mapping is an essential part of the tile type data.

¹⁵The reader may wonder if these numbers are artificially inflated since the build code replicates all parent data even in abstract tile type, but equations in abstract tile types are never processed, so no inflation exists here.

5.4.3 Java Code Generation

Translating the forward and inverse equation groups into Java code is quite tedious. The first and easiest part is to enclose parameter strings in double quotes, and to convert BFD `TRUE` and `FALSE` constants into the equivalent Java `true` and `false`.

For the forward equations, the build code then cleans up and embeds the BFD functions into the Java code, with the result cached in boolean variables. While it does this, it also builds lists of node dependencies and of arcs referenced. The node dependencies are saved with the tile type data, while the exhaustive arc list allows the build code to verify that all expected arcs are truly defined by the inverse equation groups.

If the equation group includes more than one configuration bit, the bits are declared as boolean variables, initialized to the results of the BFD expressions, and passed to the `setTileBits()` function. If the equation group includes only one configuration bit, the process is similar but is optimized with the help of class variables to reduce unnecessary object allocation.

The `invert_system()` function from the family script is invoked to invert the system of equations. The family script can either supplement or override the inversion code. Fewer than a dozen equation groups in any of the families require assistance, and most of those that do only require a hint concerning arcs that are not separable. The remaining equation groups are processed entirely by the inversion code, and the resulting inverted systems are returned.

For the reverse equations, the build code first includes any default arcs and corresponding expressions.¹⁶

If the inverted system includes more than one configuration bit, the bits are obtained from the `getTileBits()` function and cached in boolean variables in the Java code. If the inverted system includes only one configuration bit, the process is similar but is optimized with the help of class variables to reduce unnecessary object allocation.

The inverted equations are then cleaned up and embedded into the Java code, and the result of each BFD function is cached. The defined arcs are then passed to `setarc()`, which in turn passes the information to ADB. Finally, the build code ensures that each anticipated arc has been defined, and any exceptions are logged as fatal errors.

5.4.4 Failed Equation Inverting Attempts

The Xilinx Foundation tools do not include the ability to decompile bitstreams, and consequently do not include any data which would facilitate that process. This places ADB in a difficult situation since it must be able to trace through existing designs and infer existing routing, if it is expected to

¹⁶The default arc evaluates to true if and only if all other arcs in a multiplexer evaluate to false.

safely modify designs. Since the only available bitstream information comes from BFD equations, the build code must do what it can to invert BFD equation groups.

The general problem is to separate terms from systems of boolean equations, but boolean equations do not yield results as easily as linear equations because the AND and inclusive-OR operations have no inverses.

For example, basic algebraic manipulation can transform $x + y = z$ into $x = z - y$ or $x \cdot y = z$ into $x = z/y$ when $+$ and \cdot are the linear addition and multiplication operators. But no such manipulation is possible when $+$ and \cdot are the boolean inclusive-OR and AND operators, as any boolean logic text confirms.

After a measure of fruitless initial reading, a sample system of equations was posted to the *sci.math.symbolic* newsgroup, with some interesting responses. Among those responses was one from Raphael Jolly who suggested the use of Gröbner Bases, and demonstrated that his *meditor*¹⁷ tool could solve the problem.

Gröbner Bases, first introduced by Buchberger in 1965, exhibit a number of very useful properties, and have become very important in the world of mathematics. They are supported by all major symbolic math packages, and they appear to be of considerable interest to those who understand *ideals, fields, polynomial rings, and non-linear algebras*.¹⁸

The *meditor* tool indeed solved the sample system in less than five seconds, but took over a week for one of the more complex systems. It turned out that while the basic equations were simple to represent, the accompanying constraints grew very rapidly with the number of terms. Although the average performance is unknown, if one system could be solved per day, then the 1,835 Virtex-II systems would take five years to solve, and the Virtex and Virtex-E systems would take even longer. Such time frames were clearly unacceptable.

Since Gröbner Bases had shown promise, they were pursued further through various symbolic algebra packages, but these reputedly robust packages either crashed, or failed to yield the solution in a usable form. Ed Green of Virginia Tech's Math Department lent some further insight into the matter, but it quickly became apparent that Gröbner Bases would not invert systems of BFD equations in reasonable time frames.¹⁹ In fairness it must be said that Gröbner Bases are very powerful tools, but apparently not the right ones for this particular task.

Nevertheless, a comment by Green that “[*meditor*’s] results aren’t even linear”²⁰ inspired a home-grown approach based on some previously written boolean manipulation code.

¹⁷*meditor*, available at <http://raphael.jolly.free.fr/meditor/>, is a text editor with symbolic computation capabilities.

¹⁸The interested reader should have no trouble locating pertinent *modern algebra* texts which discuss these and other exciting topics.

¹⁹The author readily acknowledges that real mathematicians could probably solve this problem in short order.

²⁰Green’s expertise is in non-commutative Gröbner Bases, where results are always linear.

5.4.5 Equation Inverting Example

Although it may be difficult to separate terms in general systems of boolean equations, one key property simplifies the task considerably in the case of BFD equations. This property is based on the fact that a multiplexer output can only be driven by a single input. In the BFD context, that means that arcs with the same sink but different sources are mutually exclusive, and their product is therefore zero.

The equations presented in Figure 5.13 can be rewritten in internal shorthand form (writing `arcval(V6N1,I3,TRUE)` as `V6N1>I3`, and taking a few liberties):

```
I76.I63.I1 = V6N1>I3 || V6M1>I3
I76.I63.I11 = V6N1>I3 || V6A1>I3 || V6B1>I3
I76.I63.I7 = V6A1>I3 || V6C1>I3
I76.I63.I8 = V6B1>I3 || V6D1>I3
```

The inversion code works with the equations using modulo 2 linear algebra, and heavy reliance on XOR operations and boolean identities. Before anything else takes place, a few translations are made:

- logical AND is translated into multiplication: `A && B` becomes `A * B`
- logical NOT is translated into modulo 2 addition with 1: `!A` becomes `A + 1`
- logical OR is translated with DeMorgan's theorem: `A || B` becomes `(A+1) * (B+1) + 1`

With these translations, the equations can be re-written as:

```
I76.I63.I1 = (V6N1>I3+1) * (V6M1>I3+1) + 1
I76.I63.I11 = (V6N1>I3+1) * (V6A1>I3+1) * (V6B1>I3+1) + 1
I76.I63.I7 = (V6A1>I3+1) * (V6C1>I3+1) + 1
I76.I63.I8 = (V6B1>I3+1) * (V6D1>I3+1) + 1
```

The equation for bit `I76.I63.I1` can be expanded as:

$$I76.I63.I1 = V6N1>I3 * V6M1>I3 + V6N1>I3 + V6M1>I3 + 1 + 1$$

But because unequal arcs are mutually exclusive, the term `V6N1>I3 * V6M1>I3` drops out, and because this is modulo 2 algebra, the sum `1 + 1` becomes zero, leaving:

$$I76.I63.I1 = V6N1>I3 + V6M1>I3$$

The `+` operator may be thought to represent modulo 2 addition or the XOR operator, since the two are in fact equivalent. These same transformations applied to the entire system of equations yield:

$$\begin{aligned}
I76.I63.I1 &= V6N1>I3 + V6M1>I3 \\
I76.I63.I11 &= V6N1>I3 + V6A1>I3 + V6B1>I3 \\
I76.I63.I7 &= V6A1>I3 + V6C1>I3 \\
I76.I63.I8 &= V6B1>I3 + V6D1>I3
\end{aligned}$$

The inversion code determines which bits reference each arc, and multiplies the right-hand-sides and left-hand-sides of the respective equations. For example, consider arc $V6B1>I3$ which is referenced in the equations for bits $I76.I63.I11$ and $I76.I63.I8$. The product of these two equations is:

$$\begin{aligned}
I76.I63.I11 * I76.I63.I8 &= V6N1>I3 * V6B1>I3 + V6A1>I3 * V6B1>I3 \\
&+ V6B1>I3 * V6B1>I3 + V6N1>I3 * V6D1>I3 + V6A1>I3 * V6D1>I3 + V6B1>I3 * V6D1>I3
\end{aligned}$$

The cross-terms $V6N1>I3 * V6B1>I3$, $V6A1>I3 * V6B1>I3$, $V6N1>I3 * V6D1>I3$, $V6A1>I3 * V6D1>I3$, and $V6B1>I3 * V6D1>I3$ drop out, since they contain mutually exclusive arcs. The only remaining term is $V6B1>I3 * V6B1>I3$, and with the help of the boolean identity $A * A = A$, the result becomes:

$$I76.I63.I11 * I76.I63.I8 = V6B1>I3$$

Or:

$$V6B1>I3 = I76.I63.I11 * I76.I63.I8$$

This equation has an important physical interpretation, because it defines an arc in terms of configuration bits. This tells ADB that arc $V6B1>I3$ exists *if and only if* bits $I76.I63.I11$ and $I76.I63.I8$ are both set.

The equations which reference the other arcs can also be processed in this manner, however individual arcs do not always separate easily. The inversion code proceeds by replacing already known and separated arcs with their corresponding equations, which eventually produces results for most or all arcs. In this case the results are:

$$\begin{aligned}
V6B1>I3 &= I76.I63.I8 * I76.I63.I11 \\
V6A1>I3 &= I76.I63.I7 * I76.I63.I11 \\
V6N1>I3 &= I76.I63.I1 * I76.I63.I11 \\
V6M1>I3 &= I76.I63.I1 + I76.I63.I1 * I76.I63.I11 \\
V6D1>I3 &= I76.I63.I8 + I76.I63.I8 * I76.I63.I11 \\
V6C1>I3 &= I76.I63.I7 + I76.I63.I7 * I76.I63.I11
\end{aligned}$$

The inversion code recognizes that some of the terms in the later equations can be replaced by known intermediate results. The final result is:

$$\begin{aligned}
V6B1>I3 &= I76.I63.I8 * I76.I63.I11 \\
V6A1>I3 &= I76.I63.I7 * I76.I63.I11 \\
V6N1>I3 &= I76.I63.I1 * I76.I63.I11 \\
V6M1>I3 &= I76.I63.I1 + V6N1>I3 \\
V6D1>I3 &= I76.I63.I8 + V6B1>I3 \\
V6C1>I3 &= I76.I63.I7 + V6A1>I3
\end{aligned}$$

The discussion now returns from examples to the build code processing.

5.4.6 Equation Inverting

The inversion code is written in Perl, just like the rest of the build code. It begins by cleaning up the equations, gathering a list of all arcs and nodes involved, and rewriting arcs and nodes in shorthand. It then parses the expressions into hierarchical token lists which are translated to modulo 2 form, simplified, expanded, and simplified again.

In some cases the forward BFD equations specify information which cannot be recovered by the reverse equations. For example, the forward equations for both Virtex and Virtex-E tie certain unused wires together in order to protect the device from destructive oscillation. However the reverse equations can only determine whether or not arcs exist, regardless of whether those arcs were part of the original design or included in order to protect the device. The constructs which force this behavior are stripped out of the equations before they are processed. This results in simpler systems to invert, without any loss of design information. ADB determines which arcs are part of a design when it traces nets. Any net that does not extend all the way from a source to a sink is discarded.

Once equations have gone from tokenizing to simplification, the inversion code starts working to separate arcs. For each arc, the code determines which equations reference the arc, and then multiplies the left-hand-sides and the right-hand-sides of those equations together. Both left- and right-hand-sides then go through simplification, expansion, and simplification again.

The reader may question the need for simplification both before and after expressions are expanded. It turns out that the expressions sometimes include polynomial products of very high degree, and anything that can be done to simplify the polynomials before they are expanded is desirable.

The inversion code is very stringent in the way it accounts for arcs, and with good reason. In a small number of cases it is possible for an arc to disappear entirely from the system of equations during simplification steps. That situation leaves the inversion code unable to describe the arc, and unable to derive an inverse function for it. When such a condition is detected, the inversion code tries to solve the remaining arcs by brute force. In the brute force approach, all of the arcs that have already been solved are set to logical zero, and the resulting equations are evaluated in the typical fashion.

While the example in the previous section does illustrate the overall approach used, it is fairly simple in that it only comprises four bits and six arcs. One of the more complex equation groups encountered comprises seventeen bits, thirty-six arcs, and seven other BFD functions. In some systems, though not necessarily the largest ones, the inversion code cannot fully separate each arc that the equations reference.

When all of the simplification is finished, the inversion code gathers all of the intermediate results and begins looking for separated arcs. Each arc that is successfully extracted may be substituted into other expressions in order to separate other arcs. This process continues until all arcs have

been separated or until a deadlock condition is detected. In deadlock cases, the family script is responsible for providing appropriate hints or solutions, but such hints or solutions are required for fewer than 0.1% of the equation groups.

Obviously the inversion code is not a general purpose symbolic computation system, but rather a specific system designed for a known set of problems. The process is robust and very rapid: The forward and reverse equation groups for each of the Virtex, Virtex-E, and Virtex-II families can be generated in under five minutes on a 1,400 MHz Pentium III machine. This is substantially faster than the many years that the Gröbner Base approach might require.

5.5 Device-Specific Data

The device-specific data consists of everything that varies from one device to the next. In general this includes the tile map, segments, remote nodes, and remote arcs. As mentioned earlier, the build code is able to batch-process any number of devices which belong to the same family. A database of device-specific data is built for each device in the family. The device database cannot be understood apart from the family database, and so the device database is stamped with the family database's name to facilitate bootstrapping.

5.5.1 Tilemap Processing

The tilemap identifies each tile in a device. It provides the tile type index, the row coordinate, the column coordinate, and the tile name. The tile type index allows ADB to map any tile in the device to the appropriate family data for the tile type, and the row and column coordinates tell it where each tile is located.

The build code reads each tile entry, and places the information in a collection of arrays and hash tables. It also generates a tile index for the tile, based on the row and column coordinates:

$$\text{tile_index} = (\text{row} \times \text{column_count}) + \text{column}$$

Arrays are used to map tile coordinates to tile indexes and vice versa, and to map each tile index to the tile name, tile type name, and tile type index. A hash table also maps the tile name to a tile index, for use in reconciling remote nodes and arcs.

The tilemap data written to disk consists of the tile type index, row index, column index, and tile name for each tile in the device.

5.5.2 Parsing BXD File

The BXD data is read and parsed by a state machine and supplementary callback functions. The callback functions provide extra processing for tile and wire declarations, and for storing the data collected within each tile declaration. The bulk of the extracted data consists of the expanded segments, the tilewire-to-segment mapping, and any encountered wire synonyms.

The parsing code is able to assemble segments dynamically if necessary, but in practice the BXD data is always preprocessed as described in Section 5.2.1. In either case, each unique segment is assigned an index number, and its tilewires are stored under the segment index. This information maps segment information to tilewires. The reverse mapping is also created, with every wire in every tile pointing to the corresponding segment index.

The reader may recall the justification offered in Chapter 4 for *trivial segments* not needing to be tracked. However when the BXD information is parsed, the build code does not know which segments will end up being trivial segments, and so it blindly tracks all segments.

Wire synonyms originate when a single segment is described by two different names in the same tile. Synonym information is collected for the sake of the BFD processing, but any necessary correction remain the responsibility of the family-specific code as explained earlier.

5.5.3 Adjusting Segments

Once all of the segment information has been collected, the `adjust_segments()` function from the family script is invoked to make any necessary corrections. For Virtex and Virtex-E this function does nothing. For Virtex-II it forces wires `TBUF3` and `TBUF3_E` in the left-most `CENTER` tiles onto the same segment. This is an unusual but practical way of dealing with the fact that there are no configuration bits to control that connection, and that it is permanently on.²¹

5.5.4 Sorting and Compacting Segments

Each segment is sorted first by tile index and then by wire index in order to simplify subsequent processing. Consider Virtex XCV50 segments #7,251 and #8,347, with wires ordered as they appear in the BXD data:

```
#7,351: (H6D0@[1,8], H6E0@[1,2], H6A0@[1,3], H6B0@[1,4], H6M0@[1,5], H6C0@[1,6],
        H6D0@[1,7], H6W0@[1,9])
```

²¹This situation could not be resolved with a fake arc entry, because that would affect every `CENTER` tile in the device. It also could not be solved with remote arc entries because there is no way to specify a remote arc that is permanently on.

```
#8,347: (H6D0@[5,22], H6E0@[5,16], H6A0@[5,17], H6B0@[5,18], H6M0@[5,19], H6C0@[5,20],
        H6D0@[5,21], H6W0@[5,23])
```

When the wire and tile names are replaced with indexes, the segments become:

```
#7,351: (64@39, 112@33, 64@34, 76@35, 124@36, 88@37, 100@38, 136@40)
#8,347: (64@177, 112@171, 64@172, 76@173, 124@174, 88@175, 100@176, 136@178)
```

The tile and wire indexes are then sorted to yield:

```
#7,351: (112@33, 64@34, 76@35, 124@36, 88@37, 100@38, 64@39, 136@40)
#8,347: (112@171, 64@172, 76@173, 124@174, 88@175, 100@176, 64@177, 136@178)
```

In order to compress the full segment information, ADB and the build code rely on compact segments, as described in Section 4.9.1. The build code determines the base tile index and subtracts it from all of the segment's wires, resulting in segments which appear to start in tile zero. The offset segments are:

```
#7,351: (112@0, 64@1, 76@2, 124@3, 88@4, 100@5, 64@6, 136@7) instantiated at 33
#8,347: (112@0, 64@1, 76@2, 124@3, 88@4, 100@5, 64@6, 136@7) instantiated at 171
```

Each zero-offset segment is treated as a hash table key. If the key already exists, then a segment identical to the current one (in all respects except for the starting tile), has already been encountered. In such a case the new segment's tile offset is appended to the list of tiles where this compact segment has been instantiated. If the key does not exist, then a new compact segment is created.

The reader may note that offset segments #7,251 and #8,347 are now identical in *shape*. These and thirty other segments become part of *compact segment #489* during processing:

```
#489: (112@0, 64@1, 76@2, 124@3, 88@4, 100@5, 64@6, 136@7) instantiated at (33,
        171, ...)
```

5.5.5 Reconciling Remote Nodes

The build code is responsible for fully reconciling all remote nodes in each device. While the family processing reconciled remote nodes to *tile types*, this step must reconcile remote nodes to device *tiles*, and it must do so for *every remote node instance*.

For example, while the Virtex family processing was content to know that wire TBUF3_R1 in CENTER tiles mapped to wire TBUF3 in other CENTER tiles, this step must go through that process for each of the hundreds or thousands of CENTER tile instances in the entire device.

The build code scans each tile in the device, looking for remote node information. For each remote node that it finds, it begins by identifying the tile index of the remote node. A closer look at the data reveals that tiles which reference remote nodes are likely to reference the same remote tile more than once in a row. A simple caching scheme is used so that sequential references to the same remote tile do not need to be evaluated each time.

In the case of remote nodes specified by *relative offset*, the process should be as simple as adding the relative offset to the current tile coordinates. Generally the reconciling *is* just that simple, but the Xilinx tools sometimes “magically” skip over certain tiles, and the build code must do likewise. The `adjust_remote_offset()` function from the family script is invoked to apply any necessary adjustments. The behavior of the `adjust_remote_offset()` function was translated directly from the “magic” code provided by Xilinx.

In the case of remote nodes specified by *site name*, the build code uses the mapping from site name to tile index which was derived from the XDL data.

Once the remote tile has been identified, the build code verifies that the tile actually contains a wire with the specified name. It also verifies that the local wire was expected to reference a remote one. Any deviation from these expectations is logged.

Sometimes the failure to identify a remote tile is not an error condition. For example, Virtex CENTER tiles define node TBUF3_R1 with a relative [row,col] offset of [0,1]. This means that TBUF3_R1 generally maps to a wire one column to the right. However no such mapping exists for the right-most column of CENTER tiles. The simple interpretation of this situation is that the right-most CENTER tiles contain configuration bits which serve no purpose.

No segment information exists for nodes that are declared remote, since the BXD data has no concept of remote tiles. However the remote nodes themselves do have associated segment information, and that information is copied to the local references of each remote node that is reconciled. In this way the build code is able to keep track of which wires have and have not been reconciled.²²

When all of the remote nodes that could be reconciled have been reconciled, the build code again scans every wire in every tile and logs each one that was never referenced as part of a segment or as part of a remote node. This includes any remote nodes that could not be reconciled, and any wires defined by the BFD data which did not appear in the BXD data. The latter case usually occurs around the periphery of the device, where regular intertile connections are interrupted.

²²Although the segment information is copied to the local references, it is not currently stored in the database files nor generated when ADB starts up. Because of the way in which remote nodes are used, the author knows of no reason why this segment information should be retained.

5.5.6 Writing Remote Nodes

Each remote node that was successfully reconciled in the last step is written to the database. The data is stored under the tile that makes the remote reference, and specifies the 1) local node index, 2) remote tile index, and 3) remote node index.

5.5.7 Writing Extra Tile Segments

The set of BFD wires is a superset of the BXD wires, because of extra remote nodes that the BFD data defines, and because of wires at the edge of the device which are trimmed from the BXD data.

As explained in Chapter 4, the information which maps tilewires to compact segments can be constructed by ADB at startup, and omitting that information from the database results in a very significant size reduction. The remaining information must be provided in some other form in order to completely define all BFD wires.

The build code scans every wire in every tile, and skips all those that belong to real or trivial segments. It then writes the total number of BFD wires declared for the tile, and writes each exception individually. This information, along with what ADB can construct at startup, accounts for every BFD wire in the device.

5.5.8 Writing Compacted Segments

The compacted segment information consists of *shapes* and instantiation tiles. The build code writes all compact segment shapes, as well as list of tile indexes at which the compact segments are instantiated.

5.5.9 Reconciling Remote Arcs

The build code is responsible for fully reconciling all remote arcs in each device. While the family processing reconciled remote arcs to *tile types*, this step must reconcile remote arcs to device *tiles*, and it must do so for *every remote arc instance*.

Even though the family processing converted partially remote arcs into fully remote arcs, it did so only for minset devices, and only for the sake of determining tile types. All partially remote arcs are fully reconciled in this step.

For each tile in the device, the build code first restructures the remote arc information into a more convenient form. It then begins processing each remote arc source wire and sink wire, collecting the appropriate remote node information as it goes. If the source or sink wires do not belong to real segments, the condition is logged, and the arc is discarded. Such conditions typically mean that neither remote node was properly reconciled, which is not uncommon around the periphery of the device.

If the sink node was local rather than remote, the build code looks at its segment information, and tries to find a wire occurring in the same tile as the source node. This re-expresses partially remote arcs as fully remote arcs.

Once the remote tile and wire indexes have been reconciled, the build code also determines the equation group index which controls the arc.

5.5.10 Writing Remote Arcs

Each remote arc that was successfully reconciled in the last step is written to the database. The data is stored under the tile that contains the arc's sink wire, and specifies the 1) remote source node index, 2) remote sink node index, 3) remote sink tile index, 4) configuration bit tile, and 5) equation group index.

5.5.11 Database Compression

The resulting database file is compressed in ZIP format, which the `java.util.zip.ZipInputStream` class can read. The component sizes and compression statistics are presented in tables 5.3, 5.4, and 5.5.

The databases exhibit compression of 75% or more for all but the smallest devices, but while this number seems fairly respectable, it leaves much unsaid. Consider the Virtex-II XC2V8000, whose 10,463,182 wires require 41,852,728 bytes of memory for the mapping from tilewire to compact segment. Along with 3,187,214 bytes for the compact segment information, and an unspecified amount for remote node and arc information, the memory footprint of the device exceeds 45,039,942 bytes. By comparison the 682,649 bytes required by the database file represent a compression ratio approaching 100 : 1 with respect to the memory footprint.

This concludes the device database build process. This process can be repeated by the build code for batch processing of multiple family devices.

Table 5.3: Virtex database compression

<i>Device</i>	<i>Segment Counts</i>		<i>Database Size in Bytes</i>		
	<i>Expanded</i>	<i>Compacted</i>	<i>Uncompressed</i>	<i>Compressed</i>	<i>% Comp.</i>
XCV50	104,339	2,246	168,146	36,981	78.01
XCV100	157,935	2,270	223,528	45,411	79.68
XCV800	1,138,035	2,410	1,203,294	179,690	85.07
XCV1000	1,477,195	2,394	1,534,602	219,184	85.72

Table 5.4: Virtex-E database compression

<i>Device</i>	<i>Segment Counts</i>		<i>Database Size in Bytes</i>		
	<i>Expanded</i>	<i>Compacted</i>	<i>Uncompressed</i>	<i>Compressed</i>	<i>% Comp.</i>
XCV50E	105,947	3,140	205,733	47,469	76.93
XCV100E	159,927	3,056	258,513	55,283	78.62
XCV2600E	3,040,135	4,546	3,192,030	460,119	85.59
XCV3200E	3,868,867	4,198	4,003,633	582,422	85.45

Table 5.5: Virtex-II database compression

<i>Device</i>	<i>Segment Counts</i>		<i>Database Size in Bytes</i>		
	<i>Expanded</i>	<i>Compacted</i>	<i>Uncompressed</i>	<i>Compressed</i>	<i>% Comp.</i>
XC2V40	39,199	5,780	208,645	70,424	66.25
XC2V80	69,651	6,830	275,192	85,280	69.01
XC2V6000	3,171,411	8,267	3,135,628	504,010	83.93
XC2V8000	4,318,059	8,267	4,097,503	682,649	83.34

5.6 Build Times

While the build time performance is of negligible importance compared to runtime performance, the reader may still find the build statistics interesting. Table 5.6 shows the database build times obtained on a 1,133 MHz Pentium III machine with 1 GB of physical memory.

Table 5.6: Database build times

<i>Databases</i>			<i>Build Times (mm:ss)</i>		
<i>Family</i>	<i>Devices</i>	<i>Segments</i>	<i>Family</i>	<i>Devices</i>	<i>Total</i>
Virtex	XCV50, XCV100, XCV150, XCV200, XCV300, XCV400, XCV600, XCV800, XCV1000	5,218,515	2:35	12:34	15:10
Virtex-E	XCV50E, XCV100E, XCV200E, XCV300E, XCV400E, XCV405E, XCV600E, XCV812E, XCV1000E, XCV1600E, XCV2000E, XCV2600E, XCV3200E	16,796,611	5:50	41:11	45:21
Virtex-II	XC2V40, XC2V80, XC2V250, XC2V500, XC2V1000, XC2V1500, XC2V2000, XC2V3000, XC2V4000, XC2V6000, XC2V8000	14,062,865	6:03	47:00	53:00

Thus the Virtex, Virtex-E, and Virtex-II databases, with a cumulative total of 6,222 equation groups and 36,077,991 segments, can be built in under two hours on this particular machine.

As was pointed out earlier, the build process can be very memory intensive. Attempting to build databases for the largest devices on machines with limited physical memory is unadvisable and results in excessive page faulting. For example, building the Virtex-II databases on a faster 1,400 MHz Pentium III with only 256 MB of physical memory takes over two hours, a more than two-fold increase in build time.

Chapter 6

Database Usage

6.1 Overview

Because ADB was designed around family and device representations, rather than any particular application, it can be used in a variety of ways. ADB can provide services to JBits clients, or it can be used in standalone mode.

JBits clients may use ADB as a router or as an external wire database. When used as a router, ADB takes the place of JRoute and the JBits internal wire database. When used as an external wire database, ADB takes the place of the JBits internal wire database in order to support JRoute and other tools.

It is important to note that at the time of this writing, JBits version 3 has not yet been released, and is therefore still subject to change. As a result of this, the interface between ADB and JBits is also subject to change. The reader should be aware that the JBits-related details presented in this chapter are based on a prerelease version of JBits 3.

In standalone mode, ADB may be used as the basis for research or for other tools. While comprehensive access to real-world FPGA wiring information is generally not available to the research community, ADB provides such information through a public Java API. This might be of interest to those developing other routers or placement tools.

Finally, ADB may be used as an interactive browser. The built-in browser is essentially a debugging tool which operates in console mode, but may nevertheless prove useful to some people.

Figure 6.1 shows ADB configured for use as a JBits router. Figure 6.2 shows ADB configured for use as a JBits wire database. And Figure 6.3 shows ADB configured for standalone use. These figures present an overview, and do not necessarily show all of the hierarchy's intervening classes.

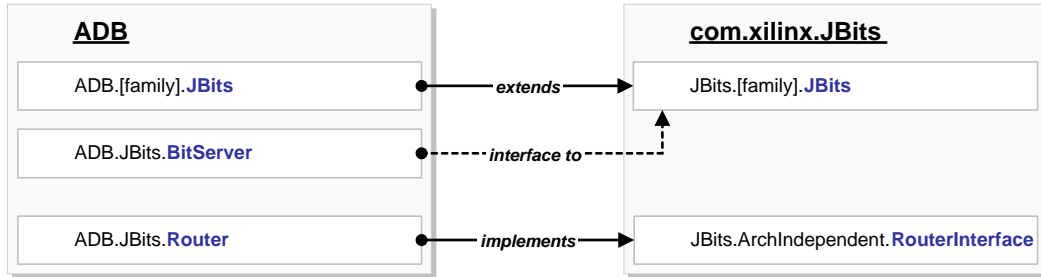


Figure 6.1: ADB as a JBits router

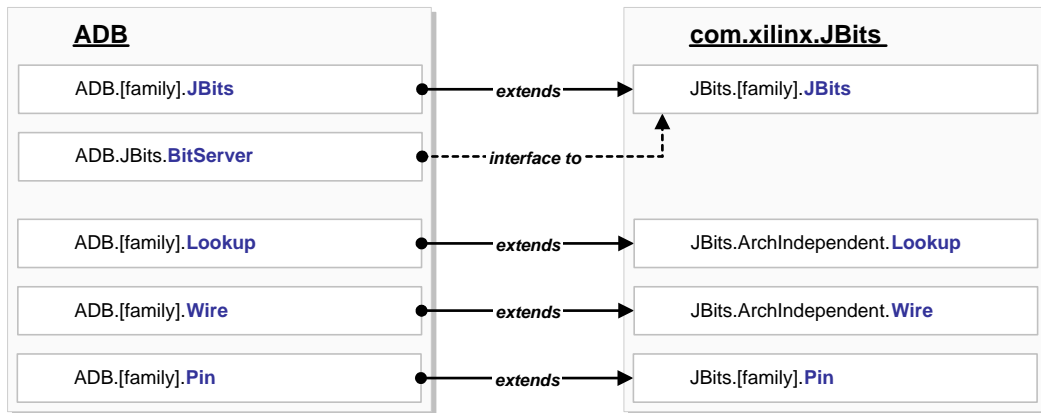


Figure 6.2: ADB as a JBits wire database

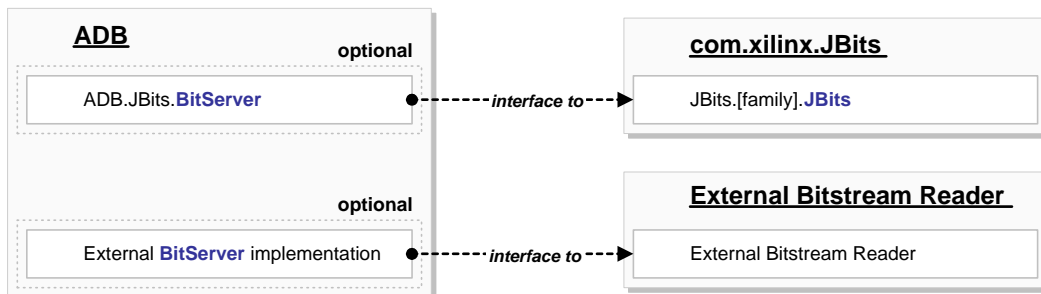


Figure 6.3: ADB as a standalone database

6.2 ADB Initialization

Regardless of the mode in which ADB is used, the databases must be loaded and initialized. In some modes, ADB must be explicitly invoked by the user, while in other modes it may be bootstrapped by related code.

In order to use ADB, the user must implicitly or explicitly create a new `ADB.DB` object, providing the device database path and file name to the constructor. ADB unzips the entire device database into memory, looks up the associated family database name, and unzips the entire family database into memory as well.

Once all the data has been preloaded into memory, ADB first processes all of the family tile data, then processes all of the device data,¹ and finally loads the family `Equations` and `Heuristic` classes from the remaining family data.

Tables 6.1, 6.2, and 6.3 show the startup time and memory usage statistics for the Virtex, Virtex-E, and Virtex-II families. Startup time statistics are provided for a 300 MHz Pentium II machine and for a 1,400 MHz Pentium III machine, each with 250 MB of memory.

In most cases, the overall initialization time is fairly short, and these startup times include the inevitable fluctuations associated with machines in a shared environment. Furthermore, after a given ADB database has been initialized once, there is often a performance boost for subsequent initializations, because of caching features in the operating system, but the numbers reported here reflect uncached performance results.

The overhead memory usage is the amount of memory that the data actually requires, while the peak memory usage is the total amount of memory allocated by the JVM. This is explained by the extra memory required during the initialization process, because ADB fully expands the compressed database files into memory before processing them.² In addition, Java's garbage collector is opaque, and it is sometimes difficult to determine exactly what else may be happening inside the JVM.

Although not shown in these tables, the time required to load the equation classes accounts for over 40% of the total startup time, for all but the largest devices.³ It seems probable that Java's bytecode verification is responsible for part of this speed penalty, and it is again difficult not to think longingly of C and the simplicity with which it supports function pointers to arbitrary code fragments.

¹This again underscores the fact that the device data has no meaning without the family data context.

²Although it would be gratifying to justify this expansion in terms of some measurable performance improvements, the real motivation for pre-expanding everything was that the streamed approach simply didn't work. `java.util.zip.ZipInputStream` and other `java.io.InputStream` objects should be cascadable, ad infinitum if necessary, but the data seen by the initialization routines contained discrete errors. The causes for these errors were not fully investigated, but if they stem from a bug in the Java API, then future versions of ADB may be able to avoid the pre-expansion step.

³In many cases this figure exceeds 50% or 60%.

Table 6.1: Virtex database initialization

<i>Device</i>	<i>Startup Time (ms)</i>		<i>Memory Usage (MB)</i>	
	<i>@ 300 MHz</i>	<i>@ 1,400 MHz</i>	<i>Overhead</i>	<i>Peak</i>
XCV50	4,278	902	6.95	10.50
XCV100	4,268	903	7.47	10.69
XCV800	5,464	1,909	17.08	26.81
XCV1000	5,900	1,366	20.35	26.87

Table 6.2: Virtex-E database initialization

<i>Device</i>	<i>Startup Time (ms)</i>		<i>Memory Usage (MB)</i>	
	<i>@ 300 MHz</i>	<i>@ 1,400 MHz</i>	<i>Overhead</i>	<i>Peak</i>
XCV50E	4,769	1,009	7.97	11.12
XCV100E	4,773	993	8.50	11.19
XCV2600E	8,259	2,065	37.02	49.56
XCV3200E	9,151	3,060	45.34	72.19

Table 6.3: Virtex-II database initialization

<i>Device</i>	<i>Startup Time (ms)</i>		<i>Memory Usage (MB)</i>	
	<i>@ 300 MHz</i>	<i>@ 1,400 MHz</i>	<i>Overhead</i>	<i>Peak</i>
XC2V40	7,354	1,590	10.71	18.81
XC2V80	7,387	1,575	11.10	14.75
XC2V6000	10,559	2,518	44.51	70.00
XC2V8000	12,074	3,023	56.26	86.75

6.3 JBits Considerations

While the ADB design is reasonably elegant and compact, some efficiency and simplicity is lost when interfacing with JBits:

- Tile coordinates must be converted whenever ADB and JBits interact. The conversion is inexpensive but very frequent. The `ADB.Virtex.Pin` and `ADB.Virtex2.Pin` classes attempt to alleviate this by caching both the JBits and the ADB coordinates.
- Wire indexes and IDs must be converted whenever ADB and JBits interact. Again, the conversion is inexpensive but very frequent, and the `Pin` classes attempt to alleviate the situation.
- The family-independent ADB functionality has to be split into family-dependent interfaces for JBits, because Java does not support multiple inheritance.
- A variety of nearly identical classes have to be replicated for the different families, because Java does not support the C++ concept of *friendship*.
- The JBits classes and data structures may be many times larger than their ADB counterparts, which could result in performance problems for large designs.

The reader may notice the conspicuous absence of Virtex-E from this discussion. ADB actually includes supporting JBits code for Virtex-E, but that code does not compile because JBits does not currently support the Virtex-E family.

6.4 ADB as a JBits Router

With a limited amount of interfacing, ADB can communicate with JBits, and provide alternate routing services. In general, an alternate router for JBits simply implements the `RouterInterface` interface. In ADB's case it is also necessary to override the `JBits` object, so that all necessary bits can be evaluated just before the bitstream is written to disk.⁴ And finally, ADB provides an alternate `Pin` class to encapsulate the coordinate and wire name conversion between JBits and ADB.⁵

6.4.1 JBits RouterInterface Interface

Figure 6.4 shows the general form of the JBits `RouterInterface` interface, with some comments simplified or removed. ADB provides the `ADB.JBits.Router` class to implement this interface, and therefore to function as an alternate JBits router.

⁴The reader may recall that many of the BFD equations have dependencies that can only be properly evaluated once the design is finalized. Because of this, ADB must wait until the user is ready to save the bitstream before it can safely evaluate the BFD equations.

⁵JBits users typically define `Pin` objects in terms of `CLB`, `IOB`, or `BRAM` coordinates, but these coordinate systems are not explicitly part of the source data, and are not used by ADB. Fortunately `Pin` objects are internally expressed in `TILE` coordinates, which are simply an *upside-down* version of the regular tile map coordinates.

```

1 package com.xilinx.JBits.ArchIndependent;
2
3 public interface RouterInterface {
4     // routes to a single sink
5     public void route(Pin src, Pin sink) throws RouteException;
6
7     // routes to multiple sinks
8     public void route(Pin src, Pin sink[]) throws RouteException;
9
10    // removes the entire net starting at src
11    public void unroute(Pin src) throws RouteException;
12
13    // removes the branch of the net that goes to sink
14    public void reverseUnroute(Pin sink) throws RouteException;
15
16 }
17

```

Figure 6.4: com.xilinx.JBits.ArchIndependent.RouterInterface interface

The `RouterInterface` interface provides functions to 1) route from one source to one sink, 2) route from one source to many sinks, 3) unroute an entire net, and 4) remove a single sink from a net.

6.4.2 ADB.JBits.Router Class

Because the `RouterInterface` interface specifies source and sink wires as JBits `Pin` objects, the `ADB.JBits.Router` class must first translate these objects into ADB tilewires. The translation requires coordinate conversions as well as wire index mapping. After the appropriate translation, ADB invokes corresponding methods from superclass `ADB.Router` to do the actual routing.

A somewhat glaring omission is that `ADB.JBits.Router` does not yet support the `unroute()` and `reverseUnroute()` methods, however the router is the subject of ongoing work, and these methods should be implemented in the near future. If the caller attempts to unroute, ADB throws an exception.

6.4.3 Routing Example

Figure 6.5 shows JBits code that demonstrates basic routing with ADB. Lines 4 through 8 import the necessary ADB classes to ensure that ADB functionality will be used. Line 14 creates a JBits object for a Virtex-II XC2V1000. The creation of an `ADB.Virtex2.JBits` object will automatically create and initialize the ADB database for the correct device.

```

1 // import the device class
2 import com.xilinx.JBits.Virtex2.XC2V1000;
3 // import the ADB classes
4 import ADB.JBits.Router;
5 import ADB.Virtex2.JBits;
6 import ADB.Virtex2.DB;
7 import ADB.Virtex2.Pin;
8 import ADB.Virtex2.Wire;
9
10 public class BasicRoutingExample {
11     public static void main(String args[]) {
12
13         // create an ADB.Virtex2.JBits object
14         JBits jbits = new JBits(new XC2V1000());
15
16         // find the DB and Router objects
17         DB db = (DB) jbits.getDB();
18         Router router = (Router) db.getRouter();
19
20         try {
21
22             // read the initial bistream
23             jbits.read("null2v1000.bit");
24
25             // route one source to one sink
26             router.route(new Pin(2,2,Wire.XQ0),new Pin(48,38,Wire.F1_B_PINWIRE0));
27             // route one source to multiple sinks
28             router.route(new Pin(3,2,Wire.YQ0),new Pin[] {
29                 new Pin(47,38,Wire.F1_B_PINWIRE3), new Pin(2,38,Wire.F1_B_PINWIRE3),
30                 new Pin(47,3,Wire.F1_B_PINWIRE3),
31             });
32
33             // write the output bitstream
34             jbits.writeFull("test2v1000.bit");
35
36         } catch(Exception exception) {
37
38             // minimal exception processing
39             exception.printStackTrace();
40             System.exit(-1);
41
42         }
43     }
44 }
45 }

```

Figure 6.5: Basic routing example

Lines 17 and 18 lookup the ADB DB (database) object and the ADB Router object. Line 23 reads the initial bitstream. Line 26 routes one pin to another pin, while lines 28 through 31 route one pin to multiple pins.

Line 34 appears to simply write the resulting bitstream, which it does in fact do, but it also indirectly evaluates all the necessary BFD equations before anything is written. This allows ADB to postpone bit updating until all design changes are final.

The reader familiar with JBits RTP cores (Run-Time Parameterizable cores), would probably appreciate a routing example based on RTP cores. Unfortunately, the JBits RTP cores explicitly use the wire IDs defined with JBits, and those wire IDs do not presently match the ones defined by ADB.⁶ This problem will be addressed during the ongoing router work, by making JBits and ADB use corresponding wire IDs.

6.5 ADB as a JBits Wire Database

An alternate wire database for JBits must provide concrete subclasses of the abstract `Lookup` and `Wire` classes. While the `Lookup` interface is straightforward, the `Wire` interface does not map very cleanly onto ADB, and the resulting efficiency loss is unfortunate. *This interface is provided for completeness, but the user should be aware of probable memory and performance issues.*

The mismatch between JBits and ADB has little to do with differing data types, and much to do with differing representations of the underlying device data. In a disturbingly large number of cases, the mapping is grossly inefficient, so while the completeness of the ADB data may be appealing to JBits users, the performance seen by the user is expected to be poor. If there is enough interest and support from the JBits team, it should be possible to tune the interface to alleviate most of these problems.

The representation conflicts and resulting mapping problems cannot be leveled at JBits, nor at Scott McMillan who designed the wire database interface and the internal JBits wire database. McMillan provided an interface proposal well in advance, and solicited comments and suggestions. Unfortunately, development on ADB had barely begun at that time, and it was entirely too early in the process to provide useful feedback from ADB's perspective. Without feedback from ADB, the wire database was influenced primarily by JRoute and other JBits features, resulting in a few choice mismatches with ADB.

⁶The earlier consensus was that the JBits and ADB wire IDs were completely independent of each other, and did not need to match. It was understood that the user code would have to be recompiled when switching between JBits and ADB routers, but no one stopped to consider that the RTP cores were explicitly based on JBits wire IDs.

6.5.1 JBits Lookup Class

Figure 6.6 shows the general form of the `Lookup` class, with simplified comments. This class provides a simple way to obtain a `Wire` object from a given `Pin` or from a set of coordinates and a wire ID. It also allows the caller to determine if a particular `Wire` or `Pin` is a logic site input or output.

```

1 package com.xilinx.JBits.ArchIndependent;
2
3 public abstract class Lookup {
4
5     // return a wire for the given pin or coordinates and name
6     public abstract Wire getWire(Pin pin) throws ConfigurationException;
7     public abstract Wire getWire(int tileRow, int tileCol, int wireID)
8         throws ConfigurationException;
9
10    // indicate whether the specified wire is a logic site input
11    public abstract boolean isInput(Wire wire);
12    public abstract boolean isInput(Pin pin);
13
14    // indicate whether the specified wire is a logic site output
15    public abstract boolean isOutput(Wire wire);
16    public abstract boolean isOutput(Pin pin);
17
18 }

```

Figure 6.6: `com.xilinx.JBits.ArchIndependent.Lookup` abstract class

It may be helpful to think of the `Lookup` class as an entry point into the wire database, since it provides access to `Wire` objects. Once a `Wire` object has been obtained, information about connecting sources or sinks can be expressed by other `Wire` objects.

6.5.2 JBits Wire Class

Figure 6.7 shows the general form of the `Wire` class, with simplified comments, and with some functions omitted. This class provides the core of the wire database API.

A high-level view of the `Wire` class reveals that it provides ways to 1) map a `Wire` back to a `Pin`, 2) inquire about and make connections with source wires, 3) inquire about sink wires, 4) obtain wire name and ID information, and 5) obtain segment information.

A reader familiar with ADB might puzzle over the number of functions that require tile row and column coordinates. The explanation is that the JBits internal wire database uses tile coordinates to initialize `Wire` objects, but does not retain those coordinates with the object. This makes `Wire`

```

1 package com.xilinx.JBits.ArchIndependent;
2
3 abstract public class Wire extends Bits {
4
5     // express the wire as a Pin object
6     abstract public Pin getPin(int tileRow, int tileCol);
7
8     // returns this wire's i'th source within the tile
9     abstract public Wire getSource(int i);
10    // returns this wire's source count within the tile
11    abstract public int numSources();
12    // connects this wire's i'th source to this wire
13    abstract public void connectSource(int i, int tileRow, int tileCol)
14        throws ConfigurationException;
15    // disconnects all of this wire's sources from this wire
16    abstract public void disconnectSource(int tileRow, int tileCol)
17        throws ConfigurationException;
18    // determines which of this wire's sources drives it
19    abstract public int getConnectedSource(int tileRow, int tileCol)
20        throws ConfigurationException;
21    // determines if any of this wire's sources drive it
22    abstract public boolean isConnected(int tileRow, int tileCol)
23        throws ConfigurationException;
24    // determines if this wire's i'th source wire connects to this wire
25    abstract public boolean isSourceConnected(int tileRow, int tileCol, int i)
26        throws ConfigurationException;
27
28    // returns this wire's i'th sink within the tile
29    abstract public Wire getSink(int i);
30    // returns this wire's sink count within the tile
31    abstract public int numSinks();
32    // expresses this wire as source index in its i'th sink
33    abstract public int getSinkIndex(int i);
34
35    // returns the device dependent wire ID
36    abstract public int getWireID();
37    // returns the numeric suffix of the wire name
38    abstract public int getWireNum();
39    // returns various forms of the wire name
40    abstract public String toString();
41    abstract public String getRootName();
42
43    // returns the segment to which this wire belongs
44    public Segment getSegment(int tileRow, int tileCol) { return null; }
45
46 }

```

Figure 6.7: com.xilinx.JBits.ArchIndependent.Wire abstract class

objects analogous to tile type wires, which may exist in many places throughout the device, and consequently the caller is responsible for providing the coordinates.

ADB implements `Wire` objects more like encapsulations of tilewires, so the original wire coordinates are always retained. If a caller provides coordinates which differ from those of the wire, ADB throws an exception, since a tilewire cannot exist at more than one place in the device.

6.5.3 ADB Lookup classes

The required JBits Lookup class is implemented by `ADB.[family].Lookup` classes, which inherit from the `ADB.JBits.Lookup` class. Together, these classes allow the user to obtain a `Wire` object, or to query the input or output status of a `Wire` or `Pin` object. The `ADB.[family].Lookup` and `ADB.JBits.Lookup` classes serve as access points into ADB, and encapsulate all of the necessary coordinate and name translations.

6.5.4 ADB Wire Classes

The required JBits Wire class is implemented by `ADB.[family].Wire` classes, which inherit from the `ADB.JBits.Wire` class. The `ADB.[family].Wire` class serves primarily to define user accessible wire ID constants, and to allow the conversion of `Wire` objects into family-dependent `Pin` objects.⁷

The `ADB.[family].Wire` class does most of the work in interfacing between JBits and ADB, through the functions described below. It also encapsulates the wire coordinates and identifiers, and provides functions and storage to cache the wire's sources, sinks, source segments, and sink segments.⁸ The value of the caching will depend greatly on the manner in which JBits uses `Wire` objects. It may indeed result in a performance boost, but the memory overhead may prove to be unacceptable. This issue will have to be resolved in conjunction with the JBits team.

getPin(int tileRow, int tileCol): Returns a `Pin` object for the current wire.

getSource(int i): Returns the requested source wire of the current wire. *Note that JBits is only interested in sources in the current tile, while ADB always considers all sources regardless of*

⁷The JBits family-dependent `Pin` classes have changed since the time this work began, and they now include explicit dependencies on the JBits internal wire database. It was initially thought best to override the family-dependent `Pin` classes, since they included support for JBits CLB, IOB, and BRAM coordinate systems, but it makes little sense to load both the ADB and JBits databases simply to support alternate coordinate systems. In the future, the JBits CLB, IOB, and BRAM coordinate systems will be built into the `ADB.JBits.Pin` class, with supporting help from the device tile maps, and therefore a number of classes currently in `ADB.[family].*` are likely to be rolled into their `ADB.JBits.*` counterparts.

⁸The term *source* is used here to specify a wire which can drive the current one, with no requirement that the source wire be a logic site output. The same applies to *sink* wires, which do not have to be logic site inputs.

location. Because of this, ADB must first identify all sources, and then filter out the ones outside of the current tile.

numSources(): Returns the number of source wires of the current wire.

connectSource(int i, int tileRow, int tileCol): Connects the specified source wire to the current wire.

disconnectSource(int tileRow, int tileCol): Disconnects all source wires from the current wire. *The caller does not specify which source to disconnect, so ADB must either identify the connected source or simply disconnect all sources. While the JBits approach simply sets the wire to an “off” value, ADB has no such value to draw upon, particularly since bit evaluation is deferred. The overhead to disconnect all sources is fairly low, so ADB takes that approach.*

getConnectedSource(int tileRow, int tileCol): Searches for a source wire that connects to the current wire, and returns the source wire’s index if applicable.

isConnected(int tileRow, int tileCol): Determines if any source wire is connected to the current wire. *JBits considers this to be an inexpensive function, since it is able to look for an “off” value, but ADB must search for connections between any source wires and the current wire.*

isSourceConnected(int tileRow, int tileCol, int i): Determines if the specified source wire is connected to the current wire.

getSink(int i): Returns the requested sink wire of the current wire.

numSinks(): Returns the number of sink wires of the current wire.

getSinkIndex(int i): Returns this wire’s source index in the specified sink wire. *In other words this function tells the caller by which index the current wire is known to the specified sink wire.*⁹ *As far as ADB is concerned, this function is deprecated.*

connectSink(int i, int tileRow, int tileCol): Connects the current wire to the specified sink wire. *This function was added to the wire database interface in order to reduce the use of getSinkIndex(), mentioned above.*

getWireID(): Returns the JBits wire ID of the current wire.

getWireNum(): Returns the numeric suffix of the current wire’s name.

toString(): Returns the full name of the current wire, with coordinates and tile information, in native ADB style. *This function probably provides much more information than what JBits users are looking for.*

⁹JBits uses this function in order to connect the current wire to a sink wire, which is supremely perplexing from ADB’s perspective, since ADB routinely connects sources to sinks without all the commotion.

getRootName(): Returns the name of the current wire. *JBits may be expecting the wire name without any numeric suffix.*

getSegment(int tileRow, int tileCol): Returns a `Segment` object which lists the current wire's permanent physical connections in other tiles. *The existence of an undirected getSegment() function is very troublesome to ADB, since a segment may look different depending on whether one is looking sourceward or sinkward.*¹⁰

In a somewhat unexpected fashion, JBits generally makes connections by looking backward toward source wires, while ADB makes connections by looking forward toward sink wires. This tendency is reflected in the many source-related functions compared to the few sink-related functions. In addition there are a few cases where ADB must throw an exception, even though the function prototype doesn't declare any. ADB sneaks around these instances by throwing `RuntimeException` objects, which do not have to be declared.

6.5.5 Wire Database Example

No user examples are provided for using ADB as a JBits alternate wire database. This interface was defined by the JBits team, and should only be used by them. All other clients should use the native ADB interface described in the next section.

6.6 ADB as a Standalone Wire Database

ADB functions best in standalone mode where it does not have to adapt to dissimilar interfaces. The native ADB API is already used by ADB's internal router and by a console-based browser utility. However the reader should note that the visibility of the interface has not been finalized. Experience shows that certain functions will require `public` access, and that other accessor functions will have to be added in order to support user requirements.

The internal design of ADB was presented in Chapter 4, but the user generally does not have or want access to the very low level functionality.¹¹ Most users will instead be interested in tile maps, wire connections, and the ability to make connections. The API is intended primarily to satisfy these needs, and it does so largely through the following classes:

ADB.DB: Main database class.

¹⁰McMillan contends that the *sourceward* or *sinkward* segments should be combined, but there are demonstrable situations where that approach will result in errors. For example, the *sinkward* view from `W_P4` in Virtex CENTER tiles is empty, but the *sourceward* view includes `W4`, as well as `E4` and `E_P4` in the adjacent tile.

¹¹In addition the low level core of ADB presumes a *correct-by-design* state, and therefore does not perform much internal error checking. One might say that ADB *trusts itself*, but it cannot reasonably trust an outside user to understand the complex interdependencies in the data.

ADB.DB.ExtendedWireInfo: Inner class providing verbose information about a single wire.

ADB.TileInfo: Class encapsulating tile name, coordinates, and tile type.

ADB.WireInfo: Class encapsulating wire name and input/output/hidden attributes.

ADB.Util.GrowableIntArray: Resizable `int` array utility class.

ADB.Util.GrowableLongArray: Resizable `long` array utility class.

ADB.Util.GrowableObjectArray: Resizable `java.lang.Object` array utility class.

ADB.JBits.DB: Database class for use in conjunction with `JBits`. *This class can only be used if a valid `JBits` object is available.*

ADB.JBits.DB.JBitsConverter: Inner class to convert between ADB and `JBits` coordinates and names. *This class can only be used if a valid `JBits` object is available.*

6.6.1 HelloWorld Example

Figure 6.8 shows the `HelloWorld` program. In fact this code does somewhat more than a traditional *hello world* program, in order to demonstrate a few additional features.

Line 11 creates and initializes the database object for use with a Virtex XCV50. The reader may reasonably object that in a real program, the database path name should be specified in a more platform-independent and flexible manner.

Lines 14 through 16 obtain tile map, tile type, and wire information from the database object. Lines 19 through 25 walk through the tile map and print it out. And lines 28 through 40 walk through the wires for each tile type and print the wire names.

Figure 6.9 shows the output of the `HelloWorld` program. Lines 4 through 16 show the database initialization transcript. Lines 19 through 24 show the tile map information. And lines 27 through 32 show the tile type wire names.

6.6.2 HelloSinks Example

Figure 6.10 shows the `HelloSinks` program. This program demonstrates simple ways of displaying a wire's attributes and connections.

Line 11 creates and initializes the database object in the same manner as the `HelloWorld` program did.

```

1 package ADB.Test;
2
3 import ADB.DB;
4 import ADB.TileInfo;
5 import ADB.WireInfo;
6
7 public class HelloWorld {
8     public static void main(String args[]) {
9         try {
10             // open and initialize the database
11             DB db = new DB("ADB/Virtex/XCV50.db");
12
13             // get the tile map, tile type names, and tile type wire info
14             TileInfo tileMap[] = db.getTileMap();
15             String tileTypeName[] = db.getTileTypeNames();
16             WireInfo tileTypeWireInfo[][] = db.getTileTypeWireInfo();
17
18             // walk the tile map
19             System.out.println("\nTile Map:");
20             for(int tileIndex = 0;tileIndex < tileMap.length;tileIndex++) {
21                 TileInfo tileInfo = tileMap[tileIndex];
22                 System.out.println("\t" + tileIndex + ": ["+ tileInfo.row() + ","
23                     + tileInfo.col() + "] " + tileTypeName[tileInfo.type_index()]
24                     + " \"" + tileInfo.name() + "\"");
25             }
26
27             // walk the tile types
28             System.out.println("\nTile Type Wires:");
29             for(int tileTypeIndex = 0;tileTypeIndex < tileTypeName.length;
30                 tileTypeIndex++) {
31                 WireInfo wireInfo[] = tileTypeWireInfo[tileTypeIndex];
32                 System.out.print("\t" + tileTypeIndex + " \""
33                     + tileTypeName[tileTypeIndex] + "\" : ");
34                 // walk the tile type wires
35                 for(int wireIndex = 0;wireIndex < wireInfo.length;wireIndex++) {
36                     System.out.print(wireInfo[wireIndex].name());
37                     if(wireIndex + 1 < wireInfo.length) System.out.print(", ");
38                 }
39                 System.out.println();
40             }
41         }
42
43         catch(Exception exception) {
44             exception.printStackTrace();
45             System.exit(-1);
46         }
47     }
48 }

```

Figure 6.8: ADB HelloWorld program

```

1 Alternate Wire Database 0.5.
2 (C) 2002 Virginia Tech. All Rights Reserved.
3
4 Opening database ADB/Virtex/XCV50.db...
5 Reading Virtex.db...
6   Reading 34 tile types...
7   Reading wire names for 34 tile types...
8   Read 179842 bytes.
9 Reading XCV50.db...
10  Reading tile map for 589 tiles (19 rows x 31 columns)...
11  Reading 589 tiles...
12  Reading 2246 segments...
13  Reading remote arcs...
14  Read 168146 bytes.
15 Reading Virtex classes...
16  Read 1359091 bytes.
17
18 Tile Map:
19   0: [0,0] UL "TL"
20   1: [0,1] BRAM_TOP "LBRAM_TOP"
21   2: [0,2] TOP "TC1"
22   :
23   :
24   586: [18,28] BOT "BC24"
25   587: [18,29] BRAM_BOT "RBRAM_BOT"
26   588: [18,30] LR "BR"
27
28 Tile Type Wires:
29   0 "TOP": CLK0, CLK1, CLK2, CLK3, CLKFB, CLKIN, GCLK0, GCLK1, GCLK2, ..., V6S_BUF3
30   1 "BRAM_TOP": DLL_CLKPAD0, DLL_CLKPAD1, CLKT_CLK2XL, CLKT_CLK2XR, ..., VGCLK3
31   2 "BOT": CLK0, CLK1, CLK2, CLK3, CLKFB, CLKIN, GCLK0, GCLK1, GCLK2, ..., V6N_BUF3
32   :
33   :
34   31 "GCLKT": CLKFB, CLKIN, H6A0, H6A1, H6A2, H6A3, H6A4, H6A5, H6B0, ..., VGCLK3
35   32 "GCLKB": CLKFB, CLKIN, H6A0, H6A1, H6A2, H6A3, H6A4, H6A5, H6B0, ..., VGCLK3
36   33 "GCLKV": BUFLO, BUFL1, BUFL2, BUFL3, BUFR0, BUFR1, BUFR2, BUFR3, ..., GCLK_B3

```

Figure 6.9: HelloWorld program output

```

1 package ADB.Test;
2
3 import ADB.DB;
4 import ADB.Util.GrowableIntArray;
5 import ADB.Util.GrowableLongArray;
6
7 public class HelloSinks {
8     public static void main(String args[]) {
9         try {
10             // open and initialize the database
11             DB db = new DB("ADB/Virtex/XCV50.db");
12
13             // look up a tilewire and display its information
14             int tile_index = db.getTileFromCoordinates((short) 1, (short) 2);
15             int tilewire = db.getTileWireFromWireNameAndTileIndex("OUT0",tile_index);
16             System.out.println("\n" + db.getExtendedWireInfo(tilewire) + "\n");
17
18             // create reusable growable arrays
19             GrowableIntArray wires = new GrowableIntArray();
20             GrowableLongArray sinks = new GrowableLongArray();
21
22             // walk the expanded segment
23             db.ExpandSegmentSinks(wires,tilewire);
24             for(int i = 0;i < wires.size();i++) {
25                 int wireEntry = wires.get(i);
26                 short iwire = (short) (wireEntry >> 16);
27                 int itile = wireEntry & 0xffff;
28                 System.out.println(db.getExtendedWireInfo(wireEntry));
29                 // get information about the connecting sinks
30                 sinks.clear();
31                 db.ExpandWireSinks(sinks,itile,iwire);
32                 for(int j = 0;j < sinks.size();j++) {
33                     long sinkEntry = sinks.get(j);
34                     int bitGroup = (int) (sinkEntry >> 48) & 0xffff;
35                     int bitTile = (int) (sinkEntry >> 32) & 0xffff;
36                     System.out.println("\t" + db.getExtendedWireInfo((int) sinkEntry)
37                         + " [" + bitGroup + "@" + bitTile + "]");
38                 }
39             }
40
41         }
42         catch(Exception exception) {
43             exception.printStackTrace();
44             System.exit(-1);
45         }
46     }
47 }

```

Figure 6.10: ADB HelloSinks program

```
1 Alternate Wire Database 0.5.
2 (C) 2002 Virginia Tech. All Rights Reserved.
3
4 Opening database ADB/Virtex/XCV50.db...
5 Reading Virtex.db...
6     Reading 34 tile types...
7     Reading wire names for 34 tile types...
8     Read 179842 bytes.
9 Reading XCV50.db...
10    Reading tile map for 589 tiles (19 rows x 31 columns)...
11    Reading 589 tiles...
12    Reading 2246 segments...
13    Reading remote arcs...
14    Read 168146 bytes.
15 Reading Virtex classes...
16    Read 1359091 bytes.
17
18 OUT0@[1,2] CENTER "R1C1" (220@33)
19
20 OUT0@[1,2] CENTER "R1C1" (220@33)
21     H6E5@[1,2] CENTER "R1C1" (119@33) [1340@33]
22     H6W4@[1,2] CENTER "R1C1" (142@33) [1341@33]
23     N0@[1,2] CENTER "R1C1" (172@33) [1359@33]
24     N1@[1,2] CENTER "R1C1" (173@33) [1457@33]
25     E2@[1,2] CENTER "R1C1" (20@33) [1338@33]
26     S1@[1,2] CENTER "R1C1" (252@33) [1152@33]
27     S3@[1,2] CENTER "R1C1" (287@33) [1288@33]
28     V6N0@[1,2] CENTER "R1C1" (390@33) [1139@33]
29     V6S0@[1,2] CENTER "R1C1" (402@33) [1140@33]
30     W7@[1,2] CENTER "R1C1" (435@33) [1276@33]
31 OUT_W0@[1,3] CENTER "R1C2" (230@34)
32     S0_F_B2@[1,3] CENTER "R1C2" (238@34) INPUT [1170@34]
33     S0_G_B2@[1,3] CENTER "R1C2" (242@34) INPUT [1105@34]
34     S1_F_B3@[1,3] CENTER "R1C2" (269@34) INPUT [1150@34]
35     S1_G_B3@[1,3] CENTER "R1C2" (273@34) INPUT [1439@34]
```

Figure 6.11: HelloSinks program output

Line 14 looks up the index of the tile at ADB coordinates [1,2]. *These are absolute tile coordinates as the Xilinx Foundation tools would specify them, and not as JBits would specify them.* Line 15 looks up the tilewire for wire OUT0 in the specified tile. The tilewire is composed of the wire index and tile index. *The JBits user is again reminded that wire names used in more than one tile type should not be assumed to have the same wire index in each of those tile types.* And line 16 prints out verbose information for wire OUT0 in tile [1,2].

Lines 19 and 20 create reusable growable arrays to hold the segment and sink information.

Lines 23 through 39 look up all of the wire's sinks and print out verbose information for each one. The `ExpandSegmentSinks()` function in Line 23 expands the segment in the *sinkward* direction. The `ExpandWireSinks()` function in Line 31 then identifies all arcs originating from any of the segment wires.

In addition to fully expanding the given wire's segment, and identifying all of its sink wires, this program also displays the equation group number and tile associated with each sink wire. In order to make a connection, the user would simply have to tell the database object to set the desired arc.

The database object would respond to an arc set request by 1) marking the arc as *set* in its `ArcUsage` object, 2) marking each of the segment wires as *used* in its `WireUsage` object, and 3) marking the equation group in the appropriate tile as *used* in its `GroupUsage` object. These steps effectively allow the database to avoid reusing the segment for another net, and to set the proper bits in the bitstream before it is written to disk.

Figure 6.11 shows the output of the `HelloSinks` program. Lines 4 through 16 show the database initialization transcript. Line 18 shows the extended wire information generated for wire `OUT0@[1,2]`. And lines 20 through 35 show all possible sink wires that the wire can drive. *The reader should note that wire `OUT0@[1,2]` in Line 20, and wire `OUT_W0[1,3]` in Line 31, both belong to the same segment. Therefore the sinks of wire `OUT_W0@[1,3]` in lines 32 through 35 are also sinks of wire `OUT0@[1,2]`, and vice versa.*

6.6.3 DB Class

The `ADB.DB` class owns and manages the selected wire database, and provides user access to higher-level information. The `DB` API is fairly elaborate, but the vast majority of functions and fields have `package`¹² or `protected`¹³ access. The functions presented here provide a subset of the exhaustive API:

¹²Unfortunately Java has no `package` keyword, so functions and fields with unspecified access automatically assume package access.

¹³Java insists that `protected` access also includes package access. Unfortunately this gives the developer no way of reserving functions exclusively for the use of subclasses.

DB(String in_device_file_name): Constructor. Requires the name and path of the desired database, referenced to the current working directory. *e.g.* “*ADB/Virtex/XCV50.db.*” This is admittedly not very portable, and will probably change in the future.

setBitServer(BitServer bit_server): Allows the user to specify a different bit server. *In practice this will probably only be used to provide alternate access to Virtex-E bitstreams, since JBits does not support Virtex-E.*

getRouter(): Returns the current router object.

getTileInfo(Integer tile): Returns the `TileInfo` object for `tile`.

getTileInfo(int tile): Returns the `TileInfo` object for `tile`.

getWireInfo(Integer tilewire): Returns the `WireInfo` object for `tilewire`.

getWireInfo(int tilewire): Returns the `WireInfo` object for `tilewire`.

getWireInfo(int tile, short wire): Returns the `WireInfo` object for `wire` in `tile`.

newExtendedWireInfo(): Returns an `ExtendedWireInfo` object tied to the database.

getExtendedWireInfo(Integer tilewire): Returns an `ExtendedWireInfo` object for `tilewire`.

getExtendedWireInfo(int tilewire): Returns an `ExtendedWireInfo` object for `tilewire`.

getExtendedWireInfo(int tile, short wire): Returns an `ExtendedWireInfo` object for `wire` in `tile`.

getRowCount(): Returns the number of rows in the tile map.

getColCount(): Returns the number of columns in the tile map.

getTileCoordinates(int tile): Returns the [row,col] coordinates for `tile`.

getTileFromCoordinates(short row, short col): Returns the tile index for coordinates [row,col].

getTileType(int tile): Returns the tile type of `tile`.

findWireFromJBitsID(short type, int jbitsID): returns the wire index matching wire ID `jbitsID` in tile type `type`.

getJBitsIDFromWire(short type, short wire): Returns the wire ID for wire index `wire` in tile type `type`.

getTileMap(): Returns the tile map as an array of `TileInfo` objects, indexed by tile index.

getTileTypeNames(): Returns the tile type names as an array of `String` objects, indexed by tile type index.

getTileTypeWireInfo(): Returns the wire information as a two-dimensional array of `WireInfo` objects, indexed first by tile type index, and then by wire index.

ExpandSegmentSources(GrowableIntArray stack, int tilewire): Expands the segment¹⁴ to which `tilewire` belongs, in the *sourceward* direction, and places all resulting tilewires in `stack`.

ExpandWireSources(GrowableLongArray stack, int tile, short wire): Expands the sources of wire in `tile`, and places all resulting sources in `stack`. *Note that this function does not expand segments; the user should first call `ExpandSegmentSources` to expand a segment.*

ExpandSegmentSinks(GrowableIntArray stack, int tilewire): Expands the segment to which `tilewire` belongs, in the *sinkward* direction, and places all resulting tilewires in `stack`.

ExpandWireSinks(GrowableLongArray stack, int tile, short wire): Expands the sinks of wire in `tile`, and places all resulting sinks in `stack`. *Note that this function does not expand segments; the user should first call `ExpandSegmentSinks` to expand a segment.*

ExpandWireSinksWithSources(GrowableLongArray targets, GrowableIntArray sources, GrowableIntArray sinks, Integer tilewire): Special purpose function used by the router. This function behaves in a manner similar to `ExpandSegmentSinks()` and `ExpandWireSinks()`, but also includes arc source and sink information. *For example, `tilewire` can connect to every `tilewire` in `targets`, but there may not be a single arc which connects them directly, since they may reside in different tiles, or may be separated by fake arcs. In order to reach `targets[i]` from `tilewire`, the arc between `sources[i]` and `sinks[i]` must be set.*

ExpandWireSinksWithTies(GrowableLongArray sinks, Integer tilewire): Special purpose function used by the router heuristics. This function behaves in a manner similar to `ExpandSegmentSinks()` and `ExpandWireSinks()`, but also includes fake arc sinks. *Fake arc sinks actually belong to the same expanded segment as `tilewire`, but explicitly listing them makes it easier for a heuristic to determine the segment's proximity to the target tile.*

6.6.4 ExtendedWireInfo Class

The `ADB.DB.ExtendedWireInfo` inner class encapsulates most of the information that may be known about a tilewire. Although `ExtendedWireInfo` objects are normally used for feedback or debugging purposes, it is possible to use them in other situations. `ExtendedWireInfo` objects are reusable, so it is common to create an `ExtendedWireInfo` object for a loop or function or class, and simply change the wire that it describes. Every part of the `ExtendedWireInfo` class is `public`:

¹⁴Expanding the segment is more than simply unpacking the compacted segments as described in Chapter 4. It also involves recursively enumerating the segment wires and descending through any fake arcs which they may drive. In some families it is possible to start at the output of a global clock buffer, and reach nearly any tile in the device, simply by passing through fake arcs.

wire_index: The wire index. *e.g.* 220

wire_name: The wire name. *e.g.* *OUT0*

wire_attributes: The wire visibility, input/output, and remote attributes. *e.g.* *output, hidden*

tile_index: The tile index. *e.g.* 33

tile_name: The tile name. *e.g.* *R1C1*

row: The tile row. *e.g.* 1

col: The tile column. *e.g.* 2

tile_type_index: The tile type index. *e.g.* 22

tile_type_name: The tile type name. *e.g.* *CENTER*

ExtendedWireInfo(): Null constructor; does not provide information for any particular wire.

ExtendedWireInfo(Integer tilewire): Constructs and updates information for *tilewire*.

ExtendedWireInfo(int tilewire): Constructs and updates information for *tilewire*.

ExtendedWireInfo(int tile, short wire): Constructs and updates information for *wire* in *tile*.

setWire(Integer tilewire): Updates information for *tilewire*.

setWire(int tilewire): Updates information for *tilewire*.

setWire(int tile, short wire): Updates information for *wire* in *tile*.

toString(): Returns a standard formatted string with almost all of the available information.

The `toString()` function deserves a little more explanation because it is so convenient to use. A typical return `String` might look like this:

```
OUT0@[1,2] CENTER "R1C1" (220@33)
```

This describes a wire named *OUT0*, at tile row and column [1,2], in a tile of type *CENTER*, with tile name *R1C1*. The wire is internally known by wire index 220 in tile index 33. The only piece of information *not* shown is the tile type. If applicable, the string would have ended with some combination of *HIDDEN*, *INPUT*, *OUTPUT*, *REMOTE*, and *LOCAL* flags.

6.6.5 TileInfo Class

The `ADB.TileInfo` class encapsulates the name, coordinates, and type of a device tile. The internal ADB tile map consists of an array of `TileInfo` objects, one for each tile in the device. The user can gain access to `TileInfo` information through the following functions:

`type_index()`: Returns the tile type index.

`row()`: Returns the tile row.

`col()`: Returns the tile column.

`name()`: Returns the tile name.

6.6.6 WireInfo Class

The `ADB.WireInfo` class encapsulates the name, attributes, connectivity, and dependencies of a tile type wire. One `WireInfo` object exists for each wire in every tile type in the family. Most of the information is hidden from users because of its complexity and dependencies, however the following field and functions are `public`.

name: The wire name.

`isHidden()`: Indicates whether the wire should be omitted from traces.

`isInput()`: Indicates whether the wire is a logic site input.

`isOutput()`: Indicates whether the wire is a logic site output.

`isRemote()`: Indicates whether the wire is a local reference to a wire in a remote tile.

`isLocal()`: Indicates whether the wire is referenced by a remote tile.

6.6.7 Growable Array Classes

ADB is primarily designed for exploring wire connectivity, whether through its internal router, or through external tools. The quantities of data within the database have already been shown in Chapter 4 to be quite large, and by extension the quantities of data required for routing can be quite large. Unfortunately ADB cannot encapsulate its primitive data types inside objects, because of the previously mentioned overhead penalties, so it must use some form of array. But Java arrays cannot be resized, and ADB has little way of knowing ahead of time how many wires it may encounter while expanding segments, and source and sink wires.

The `GrowableViewArray`, `GrowableViewLongArray`, and `GrowableViewObjectArray` classes in the `ADB.Util` package provide convenient and reusable arrays which can grow during processing. These arrays automatically double in size as needed, but never shrink. This leads to less memory processing overhead, and the ability to “clear” the entire array just by setting the internal size variable to zero.

The `ADB.Test.GrowableViewIntArray` class shows that for 10,000,000 elements, `GrowableViewIntArray` is roughly an order of magnitude faster than `ArrayList` with `Integer` objects, with only half of the memory overhead. When the arrays are preallocated for a known number of elements, the performance gains are even greater, though in general ADB is unable to preallocate these arrays. Nevertheless these classes are extremely helpful as reusable caches, and ADB relies heavily on them. The user who wants to employ ADB with maximum efficiency will probably appreciate growable arrays.

GrowableViewArray(): Constructs an array with sixteen entries.

GrowableViewArray(int size): Constructs an array with `size` entries.

GrowableViewArray(GrowableViewArray copy): Constructs an array which clones `copy`.

clear(): Sets the internal size variable to zero.

size(): Returns the internal size variable.

get(int i): Returns the `i`'th entry.

push(int entry): Adds `entry` to the array, after doubling the array size if necessary.

The `GrowableViewLongArray` and `GrowableViewObjectArray` classes have similar interfaces.

6.6.8 ADB.JBits.DB Class

The `ADB.JBits.DB` class is a subclass of `ADB.DB`, suitable for use in conjunction with a `JBits` object. Users may want to instantiate an `ADB.JBits.DB` object instead of an `ADB.DB` object if they intend to convert between ADB and `JBits` coordinates.

DB(String in_device_file_name, JBits jbits): Constructor. Requires the name and path of the desired database, referenced to the current working directory, and a valid subclass of `com.xilinx.JBits.ArchIndependent.JBits`.

getJBits(): Returns the `JBits` object.

newJBitsConverter(): Returns a `JBitsConverter` object which can be used to convert between ADB and `JBits` coordinates.

6.6.9 JBitsConverter Class

The `ADB.JBits.DB.JBitsConverter` inner class is intended to convert between ADB and JBits coordinates. As previously explained, JBits rows are *upside-down* with respect to the Xilinx Foundation tools, and JBits wire IDs remain constant regardless of the tile in which they appear. In addition, JBits defines CLB, IOB, and BRAM coordinates for the convenience of the user. This class allows accurate conversions between the two schemes.

jbits_row: The JBits row.

jbits_col: The JBits column.

jbits_wire: The JBits wire ID.

adb_row: The ADB row.

adb_col: The ADB column. *A semantic convenience, since `jbits_col` and `adb_col` are identical.*

adb_wire: The ADB wire index.

adb_tile: The ADB tile index.

adb_type: The ADB tile type index.

set(int jbits_row, int jbits_col, int jbits_wire): Sets the given JBits coordinates, and updates the ADB coordinates.

set(int adb_tile, short adb_wire): Sets the given ADB coordinates, and updates the remaining ADB and JBits coordinates.

set(short adb_row, short adb_col, short adb_wire): Sets the given ADB coordinates, and updates the remaining ADB and JBits coordinates.

The user should obtain `JBitsConverter` objects from `ADB.JBits.DB.newJBitsConverter()`, which again underscores the fact that no JBits conversion will be possible without the availability of a valid JBits object.

6.7 ADB as an Interactive Browser

ADB includes a crude but useful console-based browser, which facilitates debugging and exploring connectivity. The browser is invoked with the following command line:

```
java ADB.Browser [database_file_name]
```

For example:

```
java ADB.Browser ADB/VirtexE/XCV50E.db
```

Once the database is initialized, the user may select from a variety of functions. *The examples presented here are valid for Virtex-E XCV50E.*

B (Bit values): Prompts for a tile index and for bit coordinates, and lists the bit values. *Only valid if the user has created an ADB.JBits.Browser object and provided a valid bitstream file. e.g. tile index 1,10 and bit coordinates 0,0 1,0 2,0 3,0.*

C (Clear routes): *not presently supported.*

E (tile type Equations): Prompts for a tile type, and lists all equation groups associated with the tile type. *e.g. tile type CENTER or CLKT_4DLL.*

D (wire Detail): Prompts for a wire and tile, and lists most of the private WireInfo data, including arcs and dependencies. *e.g. wire and tile V6N1@400 or 322@45.*

F (Find by tile index): Prompts for a tile index, and identifies the tile name, type, and coordinates. *e.g. tile index 532 or 0.*

G (Go; a.k.a. wire sinks): Prompts for a wire and tile, and lists all sinks for the wire. *e.g. wire and tile index OUT0@1,2 or 16@297 or GCLK0_PW@16. c.f. S which lists sources.*

I (tile type Instances): Prompts for a tile type, and lists all matching tiles in the device. *e.g. tile type MBRAM or CLKB_4DLL.*

Q (Quit): Exits the browser.

R (Route wires): Prompts for source and target wires and tiles, and routes the source to the target. *e.g. wire and tile index S0_X@1,2 to WEA@141, or GCLKPAD1@16 to GCLKBUF0_IN@16, or I2@18,2 to O1@0,30.*

S (wire Sources): Prompts for a wire and tile, and lists all sources for the wire. *e.g. wire and tile index OUT0@1,2 or 16@297 or WEA@141. c.f. G which lists sinks.*

T (tile Types): Lists all tile type indexes and tile type names for the family.

W (tile type Wires): Prompts for a tile type name, and lists all wire indexes and wire names for the specified tile type. *e.g. tile type CENTER or CLKL.*

If the user instead creates an ADB.JBits.Browser object in the following manner, then ADB will first trace through the bitstream and identify all nets whose nodes are not listed as HIDDEN:

```
java ADB.JBits.Browser ADB/Virtex/XCV1000.db TestV1000.bit
```

Whenever ADB traces through a bitstream, it determines which wires and arcs are already in use by existing nets, and updates its wire and arc usage information accordingly. This allows it to avoid contention with existing designs.

6.8 ADB Router

ADB includes a built-in router which may be used in JBits mode or in standalone mode. The JBits router `ADB.JBits.Router` simply extends the standalone router `ADB.Router` by providing support for JBits `Pin` objects (and the necessary coordinate conversions).

This router is based on a directed search, rather than the typical maze algorithm, but unlike the Xilinx Foundation tools router, *it is not timing driven*. In fact the router itself is completely unaware of device or family properties, though associated heuristic classes do encapsulate some domain knowledge. No distinction is made between local and global routing, but the user could provide a heuristic object to segregate the two.

6.8.1 Related Work

References on routers which target Virtex, Virtex-E, or Virtex-II architectures are scarce, because most researchers do not have access to the necessary wiring and connectivity data. Hopefully ADB will help make real-world FPGA wiring data available to interested researchers.

JRoute is a notable exception to the inaccessibility issue, since it was developed internally at Xilinx [30]. JRoute has long supported a variety of routing methods, from template routers to auto-routers. Its auto-router was patterned after an A* search (see Section 6.8.2), which led to further investigation of A* as a possible candidate for the ADB router.¹⁵

A larger number of routing references are available for simpler, simulated FPGA architectures. Swartz describes a fast *directed, depth-first search*, more aggressive than A*.¹⁶ The router includes an interesting *binning* technique, which divides the device into equally-sized bins, to help confine the search to the proper area [38]. This binning technique deserves further investigation for possible future use with ADB. The results obtained by Swartz compare very favorably to *simulated annealing* approaches, which confirms that very high quality routes were obtained. However Swartz uses an FPGA architecture with “unit-length track segments,” which differs significantly from Virtex, Virtex-E, or Virtex-II wiring.¹⁷

A significant number of available routers are traditional maze routers. As Swartz indicates, maze routers perform breadth-first searches, and do indeed guarantee optimal routes in the case of

¹⁵It is not clear that JRoute actually meets the formal requirements for A* set forth by Nilsson.

¹⁶The Swartz router does bear some similarity to the ADB router in its cost evaluation function, but the ADB router was influenced by Nilsson, before the Swartz reference was discovered.

¹⁷Swartz indicates that future work will evaluate the router in segmented architecture contexts.

two-pin nets [38]. But Keller points out that maze routers are generally too slow to facilitate dynamic reconfiguration [30]. Proponents of maze routers provide various optimizations to improve performance. For example, Ludwig penalizes direction changes [39], but it is not clear that a direction-change penalty could provide any benefit to the ADB router.

6.8.2 The GRAPHSEARCH Algorithm

The ADB router is based on the GRAPHSEARCH algorithm presented by Nilsson in Principles of Artificial Intelligence [37], and reproduced here verbatim from pages 64 and 65:

Procedure **GRAPHSEARCH**

- 1 Create a *search graph*, G , consisting solely of the start node, s . Put s on a list called $OPEN$.
- 2 Create a list called $CLOSED$ that is initially empty.
- 3 LOOP: if $OPEN$ is empty, exit with failure.
- 4 Select the first node on $OPEN$, remove it from $OPEN$, and put it on $CLOSED$. Call this node n .
- 5 If n is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from n to s in G . (Pointers are established in step 7.)
- 6 Expand node n , generating the set, M , of its successors and install them as successors of n in G .
- 7 Establish a pointer to n from those members of M that were not already in G (i.e., not already on either $OPEN$ or $CLOSED$. Add these members of M to $OPEN$. For each member of M that was already on $OPEN$ or $CLOSED$, decide whether or not to redirect its pointer to n . (See text.) For each member of M already on $CLOSED$, decide for each of its descendants in G whether or not to redirect its pointer. (See text.)
- 8 Reorder the list $OPEN$, either according to some arbitrary scheme or according to heuristic merit.
- 9 Go LOOP

The embedded references to the “text” appear in the original, and refer to additional explanations provided by Nilsson. This book contains a wealth of well presented information on a variety of topics which can be applied to computer systems. The interested reader is highly encouraged to further investigate this older but very valuable book.

The GRAPHSEARCH algorithm is quite a bit more flexible than it might first appear to be, and that without a large performance penalty. The reordering scheme, referenced in Step 8, can cause the algorithm to function as a depth-first search on one extreme, or as a breadth-first search on the other extreme, or indeed as any number of different schemes in between those two, including the A and A* searches which Nilsson defines. In practice it is convenient to sort the list according to the result of some heuristic evaluation function, where the heuristic function essentially determines how the search will proceed.

The A and A* searches have a number of important properties which Nilsson describes. In particular, A* is said to be *admissible*, which means that if an optimal path exists, the algorithm will always find it. It seems clear then, that admissibility is desirable in a router, however developing the appropriate heuristic evaluation function can become very complex. The ADB router therefore uses somewhat of a homegrown heuristic, believed to provide a reasonable tradeoff between performance and optimality. Further work on the heuristic would most certainly improve the quality of the routes, albeit at the price of increased complexity, and perhaps slower performance.

6.8.3 ADB Implementation of GRAPHSEARCH

The ADB router implements GRAPHSEARCH as described above, but combines the steps into functions. Because the router must be able to route from one source to multiple sinks, where Step 1 talks of placing the start node *s* on *OPEN*, the heuristic may elect to place an entire set of nodes on *OPEN* at the outset.

graphSearch(int source, int sink): Encapsulates GRAPHSEARCH for a single sink.

graphSearch(int source, int sinks[]): Encapsulates GRAPHSEARCH for multiple sinks.

graphSearch(Integer source, Integer sink): Implements steps 1 and 2 for a single sink.

graphSearchExtend(Route route, Integer sink): Implements steps 1 and 2 for multiple sinks.

graphSearchLoop(Integer source, Integer sink): Implements steps 3 through 9, with the exception of Step 7.

graphSearchFilter(Integer tilewire, Node parent, int source, int sink, int user_data, boolean reuse): Implements Step 7.

Although ADB generally avoids relying on Java collection classes and their overhead if possible, their use is entirely appropriate within the context of the router. But even in this situation, the

router chooses `HashMap` over `Hashtable`, and `ArrayList` over `Vector`, because `Hashtable` and `Vector` include additional overhead to support thread-safe methods.

ADB includes default heuristics for each family that it supports, but the interested user should be able to write heuristics for a wide range of additional search strategies. The default heuristics provide some minimal domain knowledge and are described in appendices A and B. The basic functions of the `Heuristic` class are as follows:

`compare(Object o1, Object o2)`: Compares two nodes based on heuristic merit.

`resourceCost(Integer tilewire)`: Returns the *cost* of a physical resource.

`filterExistingRoute(Router.Route route, Router.Route union)`: Selectively adds existing route nodes as starting points for the remaining sinks.

`filterNode(Router.Node node)`: Determine whether to consider a node or skip it entirely.

`nodeCost(Router.Node node)`: Determine the overall node *cost*.

The general approach used by the heuristic is to assign a *cost* based on the scarcity or value of the physical resource, and on the distance from the target node. Attaching a cost to the physical resource is important because Virtex, Virtex-E, and Virtex-II contain many different types of routing resources, and while some of them may indeed quickly lead from one point to another, it is undesirable to use precious resources for trivial routes. The cost of a node is evaluated only when the node is first encountered, and is cached thereafter.

Chapter 7

Conclusion

7.1 Summary

This thesis described ADB, an alternate wire database, which will hopefully support and further the research and development efforts of the Configurable Computing community. The discussion began by addressing background information and the source data from which ADB was derived. It covered the database design, including data representations, both on disk and in memory. It examined the database build process, including the many complex and confusing dependencies, and the equation processing code. And finally, it described the various client interfaces by which the user can extract wiring and connectivity information.

ADB is the culmination of work that began with WireDB, and that in some ways paralleled the development of the JBits 3 internal wire database. It succeeds in reaching its stated objectives, the foremost of which is exhaustive wiring and connectivity coverage for the supported FPGA families, currently Virtex, Virtex-E, and Virtex-II. ADB may in the future be expanded to support other FPGA families, and if those families are similar enough in design, then only the build code will require modifications. The currently supported devices are:

Virtex: XCV50, XCV100, XCV150, XCV200, XCV300, XCV400, XCV600, XCV800, XCV1000

Virtex-E: XCV50E, XCV100E, XCV200E, XCV300E, XCV400E, XCV405E, XCV600E,
XCV812E, XCV1000E, XCV1600E, XCV2000E, XCV2600E, XCV3200E

Virtex-II: XC2V40, XC2V80, XC2V250, XC2V500, XC2V1000, XC2V1500, XC2V2000,
XC2V3000, XC2V4000, XC2V6000, XC2V8000, XC2V10000¹

¹The Virtex-II XC2V10000 database is not yet available because of a version conflict between the *bxtest* tool and the available *x2v10000.bxd* data file. The reader should note that the XC2V10000 has not yet been released, but building the appropriate device database should be a trivial matter since all of the family data is already fully supported.

While providing exhaustive wiring coverage, ADB also succeeds in achieving good initialization and runtime performance, and relatively compact representations on disk and in memory.² Finally, it provides compatibility with JBits and other tools through three different interfaces. It can function as a JBits router, as an external JBits wire database, or as a standalone database usable apart from JBits.

7.2 Results

Table 7.1 provides very rough comparisons of ADB, WireDB, and JBits wire database statistics. Making meaningful comparisons between the three databases is difficult because of their unique behaviors, and this data can be quite misleading if not interpreted correctly. The reader is therefore cautioned as follows:

- Startup times are typical of initial worst cases. *Subsequent invocations benefit from operating system file caching, and can drop by as much as 50%. This is true for all three databases.*
- Startup times are subject to the typical variations found in shared environments, and should be understood to carry some uncertainty. *A larger device appearing to start a little faster than a smaller one, may indicate imprecision in the timing, or may be legitimately caused by differing tile maps.* The only way to obtain more accurate measures is to average the timing over a series of runs, but that introduces operating system caching effects, so these numbers are simply presented as general estimates.
- The JBits wire database figures are specified for families rather than devices, since JBits does not have separate device databases in the same sense as ADB or WireDB.
- The JBits wire database is distributed, so the results presented here are contrived. JBits may very well load only a small percentage of its wire database, and therefore the startup time and memory sizes should not be taken to be representative of typical behavior.
- JBits does not support Virtex-E, and WireDB does not support Virtex-II.
- WireDB is no longer maintained, so it was difficult to collect meaningful data for it. *The most recent efforts to build the Virtex XCV1000 database failed because of insufficient memory on a machine with 1 GB of physical RAM. And finally, the source data for the other Virtex-E devices was unavailable, so those databases could not be built either.*
- The ADB device databases must be used in conjunction with their respective family databases, which require an additional 0.43 MB to 0.75 MB of disk space.
- RAM usage numbers are overhead amounts rather than peak amounts. This is true for all three databases.

²The excellent database build performance and the platform independence of the runtime and build code were not formal objectives of this research, but are pleasing nonetheless.

Table 7.1: Database comparison

Device	ADB			WireDB			JBits Wire Database		
	Startup (ms)	RAM (MB)	Disk (MB)	Startup (ms)	RAM (MB)	Disk (MB)	Startup (ms)	RAM (MB)	Disk (MB)
<i>Vertex</i>									
XCV50	902	6.95	0.04	3,324	12.83	1.28	4,554	4.57	12.26
XCV100	903	7.47	0.04	3,736	15.04	1.57			
XCV800	1,909	17.08	0.17	9,541	57.49	5.55			
XCV1000	1,366	20.35	0.21	-	-	-			
<i>Vertex-E</i>									
XCV50E	1,009	7.97	0.05	-	-	-	-	-	-
XCV100E	993	8.50	0.05	5,162	17.74	1.87			
XCV2600E	2,065	37.02	0.44	-	-	-			
XCV3200E	3,060	45.34	0.56	-	-	-			
<i>Vertex-II</i>									
XC2V40	1,590	10.71	0.07	-	-	-	7,872	6.62	24.55
XC2V80	1,575	11.10	0.08	-	-	-			
XC2V6000	2,518	44.51	0.48	-	-	-			
XC2V8000	3,023	56.26	0.65	-	-	-			

In spite of these numerous differences, some interesting observations can still be made:

- The JBits wire database requires less memory overhead than either ADB or WireDB.
- The ADB databases require less disk space than either WireDB or JBits wire databases.
- The ADB databases start up much faster than either the WireDB or JBits wire databases (assuming that JBits must preload its entire database).
- Because JBits does not have separate device databases, the startup times and memory requirements are constant within a family.

It may then be said that ADB compares favorably with both WireDB and the JBits wire database, *in light of the fact that it provides exhaustive wiring and connectivity coverage*. WireDB also provided exhaustive coverage, but was based on incomplete and therefore unreliable data. And JBits aims for a compromise between good coverage and simplicity.

Tables 7.2, 7.3, and 7.4 show the raw database performance when expanding wire sinks on a 1,400 MHz Pentium III machine. The data for these tables was generated by expanding every single wire in the device, to identify all possible sink wires.³ These numbers indicate that ADB can expand well over one million wires per second, and can also expand well over ten million sinks per second, for all but the smallest devices. In the worst case, for the Virtex-II XC2V40, this corresponds to 127.37 ns to expand a sink, and 753.32 ns to expand a wire, for a CPU period of 0.71 ns. The reader is cautioned that raw database performance is not an indicator of routing performance, but without adequate raw performance it is impossible to achieve good routing performance.

Table 7.2: Virtex raw database performance

<i>Device</i>	<i>Tiles</i>	<i>Wires</i>	<i>Sinks</i>	<i>(ms)</i>	<i>Wires/(s)</i>	<i>Sinks/(s)</i>
XCV50	589	216,070	1,284,452	116	1,862,672	11,072,862
XCV100	851	325,170	2,000,844	184	1,767,228	10,874,152
XCV800	5,605	2,332,806	20,136,508	1,275	1,829,652	15,793,340
XCV1000	7,169	3,020,606	27,600,236	1,722	1,754,127	16,028,012

Table 7.3: Virtex-E raw database performance

<i>Device</i>	<i>Tiles</i>	<i>Wires</i>	<i>Sinks</i>	<i>(ms)</i>	<i>Wires/(s)</i>	<i>Sinks/(s)</i>
XCV50E	627	223,214	1,377,100	119	1,875,748	11,572,269
XCV100E	897	333,942	2,119,144	205	1,628,985	10,337,288
XCV2600E	15,105	6,284,550	72,019,236	4,143	1,516,908	17,383,354
XCV3200E	19,367	8,019,870	99,858,944	5,364	1,495,129	18,616,507

Table 7.4: Virtex-II raw database performance

<i>Device</i>	<i>Tiles</i>	<i>Wires</i>	<i>Sinks</i>	<i>(ms)</i>	<i>Wires/(s)</i>	<i>Sinks/(s)</i>
XC2V40	255	111,506	659,492	84	1,327,452	7,851,095
XC2V80	391	191,590	1,177,188	122	1,570,410	9,649,082
XC2V6000	12,075	7,779,782	92,369,508	4,869	1,597,819	18,970,940
XC2V8000	15,851	10,463,182	133,981,444	6,774	1,544,609	19,778,778

³The reader will note that the number of sinks exceeds the number of physical wires in the device. This is to be expected since multiple wires may belong to a single segment, and multiple source wires may drive a particular sink wire.

It would be very interesting to compare the runtime performance of ADB, WireDB, and the JBits wire database, when faced with meaningful tasks, such as routing a series of common designs. However a number of difficulties preclude such a comparison. WireDB has incomplete arc direction information, so it cannot be trusted to *play fair*. JBits core templates currently have an implicit dependency on the JBits wire database, so there is no way to route core templates with ADB. In addition, JRoute currently fails to route one of its own test cases, which ADB succeeds in, but ADB has no support for the template routing which JRoute executes beautifully. These mismatches and disagreements between ADB and JBits will be resolved in the course of ongoing work.

It is difficult to compare ADB with the Xilinx Foundation tools databases, since they are so much more sophisticated and complete. At present, ADB has no timing knowledge, and little domain knowledge as far as routing is concerned. ADB also happily ignores logic and block RAM configurations. However the flat Xilinx databases are becoming undesirably large, and perhaps in the future certain parts of the data will be converted to hierarchical representations, similar to those used by ADB. The Xilinx Foundation tools *.bfd* and *.nph* files for Virtex, Virtex-E, and Virtex-II currently require 130.06 MB of disk space, while the corresponding ADB family and device *.db* files require 7.97 MB of disk space.

7.3 Lessons Learned

WireDB provided very valuable information by exposing Java's expensive object overhead, both in memory and performance. These overheads affected WireDB both at build time and at runtime. Furthermore, WireDB provided "flat" and fully resolved databases, which did not separate segment data from tile type data.

These observations led to very different design decisions for ADB. Firstly, ADB relies extensively on arrays of primitive data types to avoid the Java's object overheads. Secondly, ADB relies on Perl rather than Java when building the databases. Perl's excellent support for hash tables, for large files, and for regular expressions, made it the right choice for the build code. Thirdly, ADB separated the segment and tile type data, which allowed it to minimize both the device data and the family data, while still maintaining good performance.

Many comments about Java and comparisons were made throughout this thesis, and it remains the author's position that performance, memory overhead, simplicity, and code maintainability could all be improved if ADB were written in C++. Nevertheless, the option of using C++ was unavailable because of the need to interface with JBits, and it is believed that the overhead of a C++ implementation of ADB, communicating with JBits through a Java Native Interface (JNI), would be prohibitively high. But in defense of Java, it must be said that the *write-anywhere*, *compile-anywhere*, *run-anywhere* capability is very appealing, if not strictly necessary.

7.4 Future Work

ADB is in the process of being formally interfaced with JBits 3, and a number of issues discussed in this thesis are being addressed. Beyond that effort, there are other possible areas for expansion, including timing, routing, and support for other FPGA families.

7.4.1 Timing Data

ADB currently does not know anything about timing or wire delays, nor is it prepared to accommodate such information. By comparison, the JBits internal wire database can accommodate resistance and capacitance data, though that data is not yet believed to be in place, and it is unclear how resistance and capacitance alone can yield timing delays. However, ADB will require modifications if it is to include timing data in the future.

7.4.2 Virtex-E Bitstream Support

ADB does not have direct bitstream access, and must instead rely on JBits or an alternate bitstream reader. However, JBits does not support Virtex-E, so ADB cannot rely on JBits to access Virtex-E bitstreams. Instead it may be possible to release an experimental Virtex-E bitstream reader which was derived from JBits Virtex code. Whether this would be included with ADB or with JBits has not yet been determined. Until that time, while ADB users may explore Virtex-E connectivity ad nauseam, they will not be able to read or write Virtex-E bitstreams.

7.4.3 Support for Larger Devices

ADB currently supports all Virtex, Virtex-E, and Virtex-II devices, but rumors and common sense virtually guarantee that larger devices will be introduced in the future. When the number of tiles in a device exceeds 65,536, internal ADB data types will have to be changed, since tile indexes are currently represented as unsigned 16 bit integers. This was a difficult but deliberate design decision, since requiring a 32 bit data type would have essentially doubled the database sizes in memory and on disk. In practice, tile indexes are usually packed with wire indexes into a 32 bit integer, with 16 bits allotted to the tile index, 15 bits allotted to the wire index, and 1 bit lost to Java's mandatory sign bit. However wire indexes in current families only require 11 bits, and so 4 of the allotted 15 bits could be used for the tile index. That would allow for devices with up to 1,048,576 tiles.

7.4.4 Router Enhancements

The ADB router is the subject of ongoing work with Xilinx, for full compatibility with, and possible future inclusion in JBits releases. The JBits and ADB wire IDs will have to be shared, so that both can support precompiled core template classes. The family routing heuristics should be properly tuned against large representative designs. A template routing heuristic could perhaps be designed. And routers which are not based on GRAPHSEARCH could be developed for research purposes or for special situations.

7.4.5 Other Families and Architectures

The newest Xilinx FPGA architecture is the Virtex-II Pro family, which differs from the Virtex-II family mostly in the embedded PowerPC processors that it includes. In theory it shouldn't be too difficult to add Virtex-II Pro support to ADB, but it is difficult not to notice the drastic change in BFD file size. The Virtex, Virtex-E, and Virtex-II compressed BFD files ranged in size from 0.29 MB to 0.55 MB, while the Virtex-II Pro compressed BFD file is a significantly larger 2.06 MB. The reader will recall from chapters 3 and 5 that the BFD file is *the* central part of the build process, and that it contains all of the bitstream equations, which makes the sudden size change a little foreboding.

The Spartan-II family is based on the Virtex architecture, and actually shares that same BFD file as the Virtex family. The fact that ADB can already process the Spartan-II BFD file, suggests that adding Spartan-II support to ADB should be reasonably simple. A similar situation may exist for Spartan-IIE since it is based on the supported Virtex-E family.

The XC4000 architecture includes the XC4000, XC4000E, XC4000EX, XC4000L, XC4000XL, XC4000XLA, XC4000XLV, Spartan, and Spartan-XL families.⁴ Although the XC4000 architecture is older than the Virtex architecture, there may still be some demand for these families. However the BFD dialect used by the XC4000 architecture differs in a number of ways from that of the Virtex architecture, which will require some enhancements and additions to the build code. The database design should be able to accommodate the necessary data, but the build process may be a little more complex.

Adding support for a new family involves obtaining access to the source data, gaining some familiarity with the family, and developing the appropriate Perl family scripts, which encapsulate all of the family exceptions and rules. Experience shows that exceptions are plentiful, and that encoding the exceptions and rules requires a lengthy iterative process, and considerable assistance from Xilinx. However, the rest of the process is automated, and ADB has the potential to support future families as they are introduced.

⁴The XC4000A and XC4000H families have been superceded and discontinued.

7.4.6 Other Manufacturers

The current database architecture is closely tied to the Xilinx source data. It is conceivable, but by no means certain, that devices from other manufacturers could be supported by ADB. Or perhaps some limited modifications could facilitate such support. However the build code would have to change drastically, since no other manufacturers can be expected to share Xilinx's internal and proprietary file formats. Adding support for other manufacturer's devices would only be considered if there were significant interest and demand, particularly since substantial cooperation with the manufacturers would be necessary.

7.5 Conclusion

ADB has achieved its goals, and will hopefully facilitate and encourage future work in the area of Configurable Computing. The database is comprehensive, compact, fast, and compatible with JBits, and can support other FPGA families with some additional effort.

Appendix A

Virtex and Virtex-E Routing Heuristics

The Virtex and Virtex-E routing heuristics are embedded in the family database files, and visible to Java as `ADB.Virtex.Heuristic` and `ADB.VirtexE.Heuristic`. The standard `resourceCost()` function penalizes resources mostly according to their lengths, to prevent trivial routes from unnecessarily using long lines, for instance. Table A.1 shows the costs associated with different types of resources.

Table A.1: Virtex and Virtex-E resource costs

<i>Wire Class</i>	<i>Regex Pattern</i>	<i>Cost</i>
Long line	<code>^L [HV]</code>	1,800
Hex line	<code>^ [HV] 6</code>	600
Tri-state buffer line	<code>^TBUF</code>	400
<i>default</i>	<code>^.*\$</code>	100

The Virtex and Virtex-E heuristics must also pay special attention to *single* lines. The problem is that singles connect to each other through pass transistors instead of buffers, and when too many of them are connected together, the capacitance of the extended net may be such that rise and fall times exceed the timing requirements. The unofficial estimate is that ten or more singles, tied together in *any* configuration without intervening buffers, may create problems.

The Virtex and Virtex-E heuristics keep track of how many singles are tied together, and once ten or more are connected, the cost to connect other singles is made prohibitively high. In order to implement this behavior, the heuristics encode a little extra information along with each node, and verifies every group of singles in order to determine if a connection may still be safely made. Fortunately, the performance penalty appears to be minimal.

Appendix B

Virtex-II Routing Heuristics

The Virtex-II routing heuristics are embedded in the family database file, and visible to Java as `ADB.Virtex2.Heuristic`. The standard `resourceCost()` function penalizes resources mostly according to their lengths, to prevent trivial routes from unnecessarily using long lines, for instance. Table B.1 shows the costs associated with different types of resources.

Table B.1: Virtex-II resource costs

<i>Wire Class</i>	<i>RegEx Pattern</i>	<i>Cost</i>
Long line	<code>^L [HV] .* <i>and not</i> _TESTWIRE\$</code>	1,800
Hex line	<code>^ [EWNS] 6</code>	600
Double line	<code>^ [EWNS] 2</code>	200
Tri-state buffer line	<code>^TBUF</code>	400
Output mux	<code>^OMUX</code>	0
<i>default</i>	<code>^.*\$</code>	100

These costs are similar to those for Virtex and Virtex-E, with two extra entries for *double* lines and *output muxes*. The use of *output muxes* is encouraged with lower than average costs, to help expose long lines and a variety of local routing resources. And finally, all Virtex-II connections are buffered, so there is no pass transistor issue as there was for Virtex and Virtex-E.

Bibliography

- [1] G. Alexander *et al.*, *FPGA Openness*. comp.arch.fpga, March 2000. The thread is archived at http://www.fpga-faq.com/archives/threads_2000_03.html#21405, and was initiated by Greg Alexander on 3/22/2000. Graham Seaman provides a helpful synopsis at <http://www.opencollector.org/news/Bitstream/>.
- [2] P. Athanas, “Resources, Connections, and Speed.” Presented at internal weekly JBits teleconference between Xilinx and Virginia Tech Loki teams, August 2000.
- [3] S. A. Leon, “A Self-Reconfiguring Platform for Embedded Systems,” Master’s thesis, Virginia Tech, August 2001. Describes JBits running on Linux inside an FPGA.
- [4] R. Fong, *title not yet available*, Master’s thesis, Virginia Tech, 2002. Describes FPGA self-reconfiguration on a Virtex-II.
- [5] C. Patterson, “High Performance DES Encryption in Virtex FPGAs Using JBits,” in *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2000*, pp. 113–121, April 2000.
- [6] IBM, East Fishkill, New York, *IBM and Xilinx Shake up Art of Chip Design With New Custom Product*. Press Release, June 24, 2002.
- [7] Triscend, San Jose, California, *Triscend Announces the Industry’s First Configurable Processor*. Press Release, October 15, 1998.
- [8] Xilinx, San Jose, California, *Xilinx Introduces Breakthrough Virtex-II Pro FPGAs to Enable New Era of Programmable System Design*. Press Release, March 4, 2002.
- [9] Actel, Sunnyvale, California, *Actels Antifuse FPGAs: Programmable ASIC Solutions*, 2000. Brochure.
- [10] QuickLogic, Sunnyvale, California, *QuickLogic Products*, 2001. Brochure.
- [11] M. J. S. Smith, *Application-Specific Integrated Circuits*. Reading, Massachusetts: Addison-Wesley, 1997.
- [12] Xilinx, San Jose, California, *The Programmable Logic Data Book*, 1994.

- [13] Xilinx, San Jose, California, *The Programmable Logic Data Book*, 1996.
- [14] Xilinx, San Jose, California, *The Programmable Logic Data Book*, 2000.
- [15] Xilinx, San Jose, California, *Xilinx Virtex Series Reaches One Billion Dollars In Cumulative Revenue, In Record Time*. Press Release, May 17, 2002.
- [16] Xilinx, San Jose, California, *Virtex-E Extended Memory Family*, 2000. Brochure.
- [17] Xilinx, San Jose, California, *Virtex-II Platform FPGA Handbook*, 2001.
- [18] Xilinx, San Jose, California, *Virtex-II*, 2001. Brochure.
- [19] Xilinx, San Jose, California, *Virtex-II Pro Platform FPGA*, 2002. Databook—Advance Product Specification.
- [20] Xilinx, San Jose, California, *XC4000XLA/XV Field Programmable Gate Arrays*, 1999. Datasheet.
- [21] Xilinx, San Jose, California, *Spartan and Spartan-XL Families Field Programmable Gate Arrays*, 2002. Datasheet.
- [22] Xilinx, San Jose, California, *Spartan-II 2.5V FPGA Family*, 2001. Datasheet.
- [23] Xilinx, San Jose, California, *Spartan-II 1.8V FPGA Family*, 2002. Datasheet.
- [24] 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.
- [25] E. Lechner and S. A. Guccione, “The Java Environment for Reconfigurable Computing,” in *Field-Programmable Logic and Applications, Lecture Notes in Computer Science* (W. Luk and P. Y. K. Cheung, eds.), vol. 1304, (Berlin), pp. 284–293, Springer-Verlag, September 1997.
- [26] J. Ballagh, “An FPGA-based Run-time Reconfigurable 2-D Discrete Wavelet Transform Core,” Master’s thesis, Virginia Tech, June 2001.
- [27] S. Lopez-Buedo, J. Garrido, and E. Boemo, “Measurement of FPGA Die Temperature Using Run-time Reconfiguration,” in *Proceedings of the 7th International Workshop on Thermal Investigations of ICs and Systems (THERMINIC 2001)*, (Paris), September 2001.
- [28] S. McMillan and S. A. Guccione, “Partial Run-Time Reconfiguration Using JRTR,” in *Field-Programmable Logic and Applications, Lecture Notes in Computer Science* (R. W. Hartenstein and H. Grunbacher, eds.), vol. 1896, (Berlin), pp. 352–360, Springer-Verlag, August 2000.
- [29] Xilinx, San Jose, California, *Virtex Series Configuration Architecture User Guide*, September 2000. Xilinx Application Note XAPP151, Version 1.5.

- [30] E. Keller, “JRoute: A Run-Time Routing API for FPGA Hardware,” in *Parallel and Distributed Processing, Lecture Notes in Computer Science* (J. Rolim *et al.*, eds.), vol. 1800, (Berlin), pp. 874–881, Springer-Verlag, May 2000.
- [31] Xilinx, San Jose, California, *Xilinx Integrated Software Environment 4.2i Documentation*. Xilinx Design Language (XDL) usage and BNF syntax for design translation are described in `$XILINX/help/data/xdl/xdl.html`. Usage and BNF for device reports do not appear to be available. The `xdl` tool can be found at `$XILINX/bin/{platform}/xdl.exe`.
- [32] D. Flanagan, *Java in a Nutshell*. Sebastopol, California: O’Reilly & Associates, 1999.
- [33] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Reading, Massachusetts: Addison-Wesley, 1999.
- [34] J. Bloch, *Effective Java Programming Language Guide*. Boston, Massachusetts: Addison-Wesley, 2001.
- [35] C. Larman and R. Guthrie, *Java 2 Performance and Idiom Guide*. Upper Saddle River, New Jersey: Prentice Hall PTR, 2000.
- [36] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*. Reading, Massachusetts: Addison-Wesley, 1990.
- [37] N. J. Nilsson, *Principles of Artificial Intelligence*. Palo Alto, California: Tioga Publishing Company, 1980.
- [38] J. S. Swartz, V. Betz, and J. Rose, “A fast routability-driven router for FPGAs,” in *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, pp. 140–149, ACM Press, 1998.
- [39] S. H.-M. Ludwig, *Hades - Fast Hardware Synthesis Tools and a Reconfigurable Coprocessor*. Ph.D. thesis, ETH Zurich, Institute of Computer Systems, Nov. 1997.

Vita

Neil Steiner was born in Monaco, in 1969, where his parents were missionaries with Trans World Radio. He attended the Lycée Albert 1er through the equivalent of Junior High, and then went to boarding school at the Black Forest Academy in Kandern, Germany. In 1998, after many years of part- and full-time work and study, Neil earned a B.A. from Wheaton College, and a B.S.E.E. with honors from Illinois Institute of Technology. During those 10 years he had the pleasure of writing very cool C++ code on the Mac for The DesignSoft Company. But after earning his degrees, he headed toward a warmer climate and a hardware position with Raytheon Systems Company in Dallas, where he worked on a variety of projects, including a few black projects which he still knows nothing about.

Through various circumstances, Neil's yearning for academic punishment resurfaced, and he enrolled as a graduate student at Virginia Tech in 2000, with the express desire of joining the Configurable Computing Lab. As a result, and in continuation of his research, he was offered a summer internship with Xilinx in 2002. Neil hopes to begin working on a Ph.D. in Spring 2003.

Neil has enjoyed visiting 18 countries, including much of Western Europe and some of Eastern Europe. He has lived in Monaco, Germany, Chicago, Miami, Dallas, Blacksburg, VA, and Longmont, CO, and as a result, his friends are spread across more countries and states than he cares to remember. Neil speaks fluent, if somewhat rusty French, but his German is lousy, and his Italian always comes out German. A thorough analysis is still pending. Many of his hobbies have fallen prey to graduate school life, but he still loves playing drums, given the right music.