2014-12-01

Tincr: Integrating Custom CAD Tool Frameworks with the Xilinx Vivado Design Suite

Brad S. White
Brigham Young University - Provo

Tincr: Integrating Custom CAD Tool Frameworks with the Xilinx Vivado Design Suite

Brad Selian White

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Brent E. Nelson, Chair
Brad L. Hutchings
Michael J. Wirthlin

Department of Electrical and Computer Engineering

Brigham Young University

December 2014

ABSTRACT

Tincr: Integrating Custom CAD Tool Frameworks with the Xilinx Vivado Design Suite

Brad Selian White
Department of Electrical and Computer Engineering, BYU
Master of Science

The field programmable gate array (FPGA) is appealing as a computational platform because of its ability to be repurposed for a number of different applications and its relatively low design cost. Traditionally, FPGA vendors provide a set of electronic design automation (EDA) tools to assist customers with the implementation of their designs. These tools are necessarily general purpose, and the resulting tool flow does not provide the user much in the way of customization.

Frameworks such as RapidSmith and Torc allow for the creation of custom CAD tools that are able to target actual Xilinx FPGA devices. However, they are built on the Xilinx Design Language (XDL), which was discontinued with the introduction of Xilinx's new tool suite Vivado. Instead, Vivado provides direct access to its data structures through a Tcl interface, as well as EDIF and Xilinx Design Constraint (XDC) files.

This thesis discusses Vivado's ability to support a custom CAD tool framework similar to RapidSmith and Torc. It provides a detailed description of the CAD-related aspects of Vivado's Tcl API and shows how its command set can be used to integrate a custom CAD tool framework. This is demonstrated through the introduction of Tincr, a suite of two Tcl-based libraries that each encapsulate a separate method for implementing such a framework. The first is the TincrCAD library, a high-level CAD tool framework built within Vivado's Tcl environment. The second is TincrIO, a set of Tcl commands that comprise a file-based interface into Vivado, similar to XDL. These libraries are offered up as evidence that the Vivado Design Suite can provide a foundation for the implementation of custom CAD tools that operate on Xilinx FPGAs for the foreseeable future.

Keywords: FPGA, Vivado, Tcl, CAD tool, Tincr, RapidSmith, Xilinx

ACKNOWLEDGMENTS

I would like to dedicate this work to my wife, Ashley, whose love and support gave me the motivation I needed to complete this work, even after I had began full-time employment. The completion of this thesis represented the culmination of many nights and weekends during which she cared for our newborn daughter Olivia with an unwavering amount of patience and love. I am also thankful to my parents for instilling in me a love for education, and providing me with countless opportunities to learn and grow throughout my lifetime.

I would like to thank Dr. Brent Nelson for hiring me into BYU's Configurable Computing Lab and mentoring me over the last four years of my life. His guidance and direction were invaluable to my growth as a graduate student. I am grateful for the time and patience he offered as this thesis took shape, despite the many ups and downs we encountered along the way.

Thanks to Dr. Brad Hutchings who provided valuable feedback on this project throughout its development. I am grateful for his example and the advice he offered up throughout my tenure as a graduate student.

I would also like to thank Dr. Mike Wirthlin, whose courses on digital design instilled in me a love for FPGAs. His leadership in CHREC has had a positive influence over many students, and provided me and others with opportunities for research and close interaction with industry.

I owe my thanks to several students, whose examples and assistance contributed to my growth as a graduate student: Chris Lavin, Jaren Lamprecht, Josh Monson, Travis Haroldsen, Danny Savory, Michael Gardner, Wyatt Hepler, Thomas Townsend, Alex Harding, and all of the other students in the Configurable Computing Lab.

This research was supported by the I/UCRC Program of the National Science Foundation under Grant Numbers 0801876 and 1265957 through the NSF Center for High-Performance Reconfigurable Computing (CHREC).

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Field programmable gate arrays (FPGAs) have gained popularity in recent years as computational platforms [1]. FPGAs pose an attractive alternative to application specific integrated circuits (ASICs) because of their ability to be re-programmed as many times as the hardware designer wishes. The re-programmable aspect of FPGAs not only allows the device to be re-purposed, but also eliminates the lengthy fabrication process inherent in ASIC design during each iteration of the design cycle. Consequently, the number of design-debug iterations that a hardware designer may complete on a project in a single day is increased, thus increasing that designer's productivity.

The design flow for FPGAs is a fully automated process. FPGA vendors often offer a set of electronic design automation (EDA) tools to help consumers map designs to their devices. These EDA suites provide the necessary tools for a hardware designer to implement a design on an FPGA, which includes computer aided design (CAD) applications for synthesizing, technology mapping, placement, and routing. They encapsulate every step of the design process, from synthesizing HDL to uploading a bitstream to the FPGA. However, these CAD tools are necessarily general purpose and do not give the user with much control over how they operate on a design.

The lack of fine-grained control over vendor-issued tools has created a demand in the FPGA community for custom CAD tools. This need for custom CAD applications (such as placers and routers) is evidenced by the development of open source CAD tool frameworks such as Versatile Place and Route (VPR) [2]. VPR is among the most popular of these frameworks and has served as the proving ground for many great applications such

as [3], AAPack, and T-VPack [4]. Unfortunately, VPR was developed to target hypothetical architectures, and as of yet is unable to operate on actual commercial FPGAs on its own.

In recent years, other frameworks have emerged that are able to target Xilinx FPGAs using the Xilinx Design Language (XDL), including RapidSmith [5], Torc [6], and FAT [7]. These tools use Xilinx's Integrated Software Environment (ISE) EDA suite to output information regarding a design or device into a human-readable format. From that point, designs can be manipulated or mapped, and then read back into ISE and converted into a bitstream that could program an FPGA. These frameworks led to the development of many practical tools such as HMFlow [8] and TFlow [9].

In summer 2012 Xilinx released Vivado, its next EDA suite for future FPGA architectures [10]. As ISE's replacement, Vivado brings a number of changes including the discontinuance of XDL. Without XDL, XDL-based CAD tools and frameworks will become obsolete with future generations of Xilinx FPGAs. The goal of this work is to investigate the facilities provided by Vivado and their ability to support a CAD tool framework. Specifically, it answers the question of how Vivado can be used to implement custom CAD tools for Xilinx FPGAs going forward.

1.2 Preview of Approach

As previously mentioned, the goal of this work is to show how Vivado can support custom CAD tools that can target future generations of Xilinx FPGAs. The particular approach of this work may be broken up into two parts. First, an in-depth study of Vivado and its capabilities was required to construct a comprehensive knowledge base of the tool. As Vivado is a relatively new addition to the FPGA community, the only available publications on the matter consisted mainly of official documentation and user guides, which are cited throughout this work. While these offered a good starting point, they are not necessarily an exhaustive source of information. When additional information was required, the official Xilinx forums were consulted, where employees offered up useful information on a variety of topics. In some cases, there were questions that neither source of information could provide an answer to. This ultimately led to cases of trial and error to establish certain capabilities and functionality within Vivado.

Upon establishing said knowledge base, the second part of this study focused on identifying potential avenues for custom CAD tool frameworks. This involved careful study of Vivado's existing mechanisms and determining the best way to utilize them so that the resulting framework met a set of requirements. These requirements dictated that the resulting framework have a similar range of functionality as existing frameworks, and that it be able to perform its functions in a reasonable amount of time. Several methods for implementing such a framework were considered and ultimately two were decided upon: a Tcl-based framework integrated within the Vivado Tcl environment itself, referred to as TincrCAD, and an external CAD tool framework interface called TincrIO. These were then implemented and tested against the aforementioned set of requirements.

1.3 Contributions of this Work

As Vivado is still relatively unexplored by the academic world, much of the information presented in this thesis is novel to its field. While custom CAD tool frameworks is not a new subject in the realm of FPGA research, this work is unique in that it proves and also provides a means by which CAD tool frameworks may be implemented for the foreseen future of Xilinx products. Specifically, the main contributions of this thesis are listed as follows:

- It provides unique insight into the capabilities of Vivado from a CAD tool perspective and documents this information in a clear and precise way.
- It introduces TincrCAD, a Tcl-based CAD tool framework for Vivado. Using Vivado's included tool command language (Tcl) interpreter, TincrCAD is a set of functions built on top of Vivado's native command set to provide access to Vivado's internal data structures at a higher level of abstraction. It can be used to implement custom tools such as placers and routers.
- It presents TincrIO, an external CAD framework interface into Vivado. Following the XDL-based CAD tool framework model, this is an ASCII file-based interface into Vivado's internal database for external frameworks such as RapidSmith to interface with.

- It offers performance evaluations on each of the candidate framework methods, including measurements of Vivado's Tcl interpreter and its relevant I/O commands.

An overview of FPGA architecture, XDL, and existing CAD tool frameworks is provided in chapter 2, and chapter 3 provides CAD-relevant documentation on the Vivado Design Suite. Chapter 4 introduces TincrCAD as a CAD tool framework for operating within Vivado, while chapter 5 presents TincrIO as a means by which external CAD tool frameworks to interface with Vivado. Finally, chapter 6 reviews the findings of this thesis and discusses the potential impact of the contributions made in this work.

Unless otherwise stated, tests and experiments in this work were performed on a machine running Windows 7 64-bit with a Core i7-4770 (@ 3.4GHz) processor and 32GB of RAM. Data was gathered from the 2014.2 version of Vivado and the 14.7 version of ISE where applicable.

Chapter 2

Background and Related Work

The objective of this chapter is to provide the reader with a basic understanding of the structure of Xilinx FPGAs and the tools available for operating on them. This chapter begins by providing a brief explanation of Xilinx's FPGA architecture and terminology. It then describes established tools that target Xilinx FPGAs, such as ISE and RapidSmith. Any discussion of Vivado is avoided, as it is addressed in the following chapter.

2.1 Xilinx 7 Series FPGA Architecture

This paper refers to concepts and terminology relating to CAD tools that are able to target actual Xilinx FPGAs. Consequently, the discussion that ensues requires a basic understanding of Xilinx's FPGA architecture and nomenclature. This sections seeks to provide readers with a brief overview of the individual components and resources available within Xilinx's 7 Series FPGA architecture.

2.1.1 Logical Resources

The logic resources on a Xilinx FPGA are organized into a hierarchy of device elements. This complements the uniform nature of an FPGA as elements of the same type will typically be identical in their composition. A Xilinx FPGA's hierarchy consist of the Part, Tiles, Sites, and BELs, as shown in figure 2.1. This section will provide a rudimentary background on each of these levels, as well as their relationships towards one another.

Part

At the highest level of the hierarchy is the FPGA **part**. A part refers to a single FPGA chip and may be described using a number of identifiers such as architecture, family,

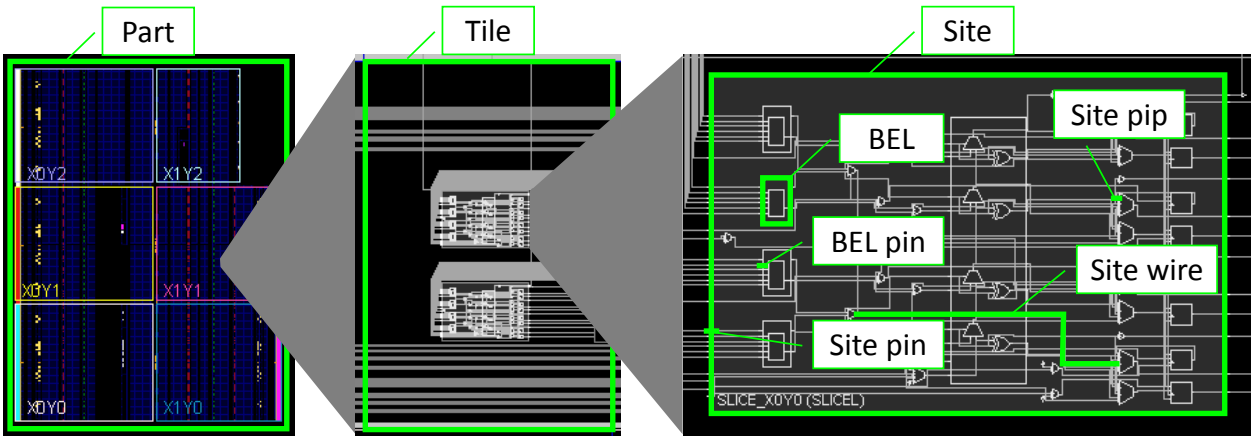

Figure 2.1: Diagram of the component hierarchy in 7 Series FPGAs

device, package, and speed grade. For instance, the xq7vx330trf1157-1I part comes from the virtex7 architecture and is a member of the qvirtex7 family. It is the xq7vx330t device implemented with the rf1157 package and has a speed grade of -1I. In any case, all of these identifiers may be inferred from the device's unique name.

Tile

The fabric of an FPGA part is divided into a two dimensional grid of **tiles**. A tile is a rectangular division that has sites, wires, and PIPs. Ultimately, wires in a tile provide the connectivity between sites, while a tile's PIPs provide the connectivity between wires. Each tile has a name, type, and a coordinate detailing its position on the device's tile grid starting from the bottom left corner of the device.

The reconfigurable nature of an FPGA compels a uniform distribution of tile types across its fabric. Tiles of the same type are generally identical in their composition, possibly with minor changes in the routing structure. For instance, a tile on the edge of the device may have fewer wires than other tiles of the same type, since there is not an adjoining tile on one of its sides.

Site

At the next layer of abstraction are **sites**. A site exists within a tile and has a name, type, and in most cases, a coordinate. The coordinate system for sites bears little to no

resemblance to the tile coordinate system and is strictly based on the grid of sites *of the same type*, with the origin at the lower left corner of the device.

The interface to a site is made up of **site pins**, each of which has a name and direction. These pins mark the boundary between the parent tile's routing network and the site's internal routing structure. Within a site, its composition mimics that of tiles, consisting of a network of interconnected BELs. Unlike a tile, all routing resources are contained within the site. **Site wires** provide the connectivity between site pins and/or BEL pins, while **site PIPs** provide the connectivity between site wires.

Basic Element (BEL)

The most atomic component in an FPGA is the basic element, or **BEL**, examples of which include flip-flops (FFs), look-up tables (LUTs), block RAMs (BRAMs), and input/output buffers (IOBs). BELs have a name, type, and a set of **BEL pins**, each of which has its own name and direction (input or output). BELs of the same type are always identical in every way except their names which are unique. A BEL is contained within a site, and in cases when there are no other components involved, the site will simply serve as a wrapper for the BEL, passing the signals from its site pins directly through to the respective BEL pins.

2.1.2 Routing Resources

Unlike ASICs, FPGAs do not have fixed traces of wire connecting the various components on a device together, so the programmable routing resources on an FPGA need to be plentiful in order to accommodate a wide variety of possible routing configurations. Complex calculations are involved in determining the optimal quantity of routing resources an FPGA provides, which lie outside the scope of this work. What is relevant are the types of resources available and a general idea of their quantities in a modern FPGA.

Node

The **node** is the highest level of abstraction within an FPGA's routing structure. A node may be thought of as a continuous piece of metal that terminates in site pins and/or

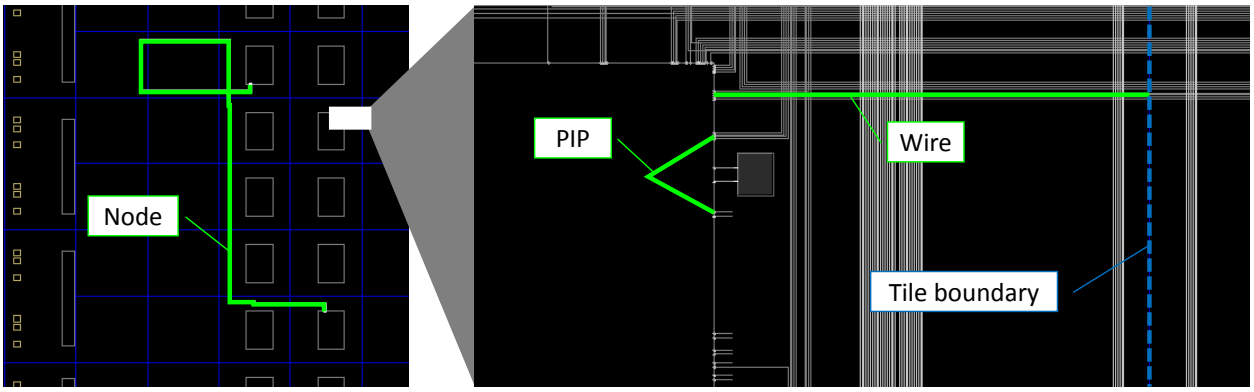

Figure 2.2: Diagram of the routing resources in 7 Series FPGAs

PIPs. As can be seen in figure 2.2, nodes may span multiple tiles, with endpoints in two or more tiles. In most cases, nodes are directional, meaning that when a design is mapped to a device, used nodes will be driven by a single distinct source, which may take the form of either a site pin or PIP. Additionally, the node will drive one or more sinks, which will also be either a site pin or PIP. There is no programmable aspect to a node, only an inherent connection between the node's source and all of its sinks, meaning that if its source is driving the node high, then all of its sinks are driven high as well.

Wire

Nodes are composed of one or more adjacent **wires**. Like nodes, wires refer to the channels of unprogrammable metal running throughout an FPGA. Unlike nodes, wires are divided by tile boundaries, meaning that the same wire cannot exist in two separate tiles. Simply put, wires are a subdivision of nodes that are sourced and sinked by not only site pins and PIPs, but possibly other wires as well. For example, a node that crosses no tile boundaries has only one wire, whereas a node that traverses many tile boundaries has just as many wires. On average, a node consists of about three wires.

Programmable Interconnect Point (PIP)

A programmable interconnect point (**PIP**) is simply a programmable connection between two nodes (or, consequently, two wires). The routing network within an FPGA is controlled almost entirely by PIPs, with a few exceptions including BELs that double

as pseudo-PIPs called route-throughs (see section 3.3.2). When an FPGA is programmed, the bitstream is effectively turning on and off PIPs to create a path of nodes connecting components across the device. A single node may drive many enabled PIPs, but only one enabled PIP may drive a node.

PIPs are almost always collected within routing matrices called switchboxes. Generally, entire tiles are dedicated to housing these switchboxes and are called interconnect tiles. These tiles are strategically placed throughout the part to promote the greatest possible connectivity between components.

Site Wire

Within a site, a similar but different routing structure exists. Components inside of a site are connected by traces of metal called **site wires**. Site wires have a name and direction. They begin and terminate in either a site pin, BEL pin, or site PIP. Like nodes and wires at the tile level, there is no programmable aspect to a site wire, it simply provides connectivity.

Site PIP

The routing within a site is controlled by **site PIPs**. Site PIPs are similar to PIPs in that they can be turned on and off when the FPGA is programmed. However, site PIPs are normally organized into multiplexors, meaning that a set of site PIPs associated with the same mux are mutually exclusive of one another because they all offer a connection to the same outgoing site wire. An example of a site PIP and site wire is shown in Figure 2.1.

2.2 XDL: The Xilinx Design Language

As do most major FPGA vendors, Xilinx offers an electronic design automation (EDA) tool suite for mapping designs (usually written in some form of HDL) to their devices. Until recently, Xilinx packaged all of these proprietary CAD tools into their Integrated Software Environment (ISE) suite.

Figure 2.3 shows the basic flow of ISE's CAD tools from a design written in HDL to a placed and routed bitstream file (.bit). Each of the red boxes represents a separate run of a specific CAD tool. For instance, `map` loads a .ngd netlist and maps its elements to the

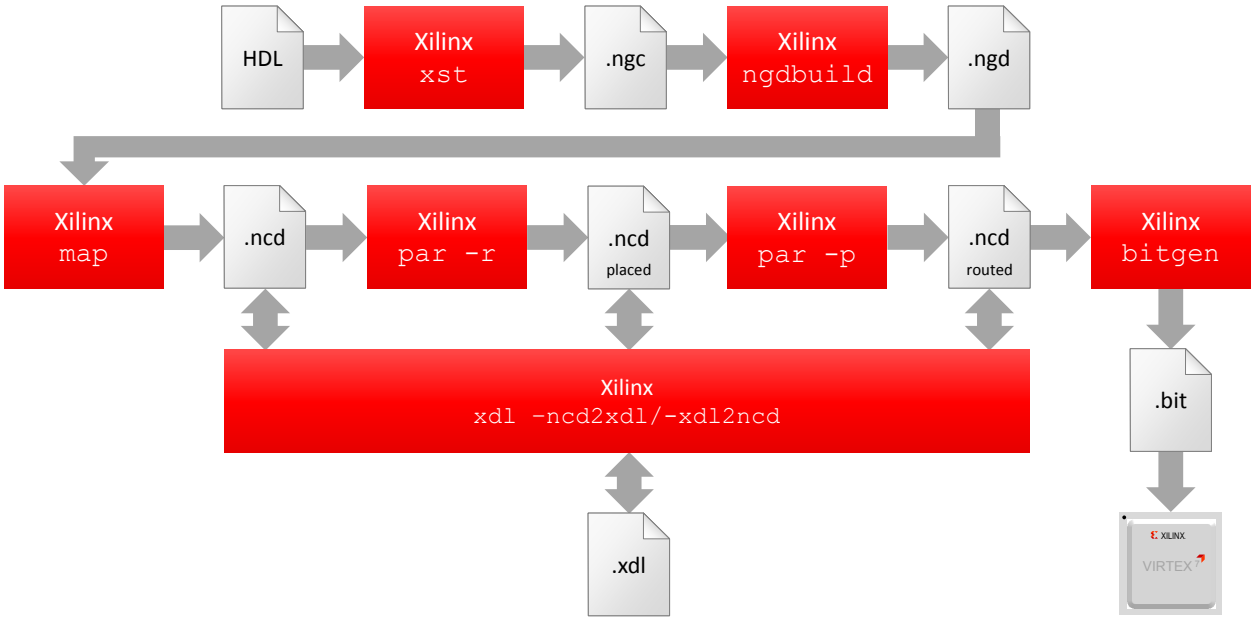

Figure 2.3: Depiction of ISE's CAD tool flow with XDL I/O

available types of components on the FPGA, also known as technology mapping[1]. The `par` tool then places (`-r`) this technology mapped netlist and routes (`-p`) it to the FPGA. Finally, the `bitgen` tool loads the placed and routed netlist and uses it to generate a bitstream that can be downloaded to the FPGA.

Perhaps the biggest challenge facing custom CAD tool frameworks that target actual FPGAs is gaining access to enough information to build a suitable representation of all the resources on the device. At some point during ISE's lifetime, Xilinx introduced the Xilinx Design Language (XDL), a grammar capable of completely representing a Xilinx design or device in a human-readable, ASCII format. The `xdl` executable was added to ISE's toolbox, allowing users to convert designs between Xilinx's proprietary NCD format and XDL, as well as generate complete device resource descriptions in an XDL report (XDLRC) file.

2.2.1 XDL Design Files

At any point after a netlist has been technology mapped, ISE allows the user to extract the netlist into a Xilinx Design Language description that is written to file using

[1]For architectures other than Spartan-3 and Virtex-4, `map` will also do some preliminary placement. For simplicity, this example and its associated figure will assume a Virtex-4 part is being targeted.

its `xdl` executable, as shown in figure 2.3. Specifically, `xdl` converts Xilinx's native circuit description (NCD) files into human-readable XDL files or vice versa using the switches `-ncd2xdl` or `-xdl2ncd`. XDL has most, if not all of the same capabilities as NCD [11]. These XDL files may then be parsed and used to populate an external data structure. Hence, the `xdl` executable effectively creates an interface into Xilinx's ISE tool flow that is well-suited for an external CAD tool framework. The following sections very briefly describe the elements that make up an XDL file.

Instances and Placement

An **instance** is the primitive logic element in XDL. It has a name, type, and a set of attributes. Instances map directly to sites on the FPGA, and so the type of an instance will actually be a site type that the instance is compatible with. When an instance is placed, its placement is represented in the instance header by the keyword "placed" followed by the name of the site it is placed on. BELs themselves are not represented, but rather are programmed by the attributes of the instance placed on their parent site.

Nets and Routing

The **net** construct in XDL represents a logical connection between two or more instances. A net has a name and a list of pins on instances that represent the connections the net makes. Nets may also contain routing information, which is provided by a list of PIPs that must be turned on in order to complete the net's route across the FPGA, starting from its source pin and ending at its sink pins.

2.2.2 XDL Resource Report Files

XDL resource report (XDLRC) files use a syntax very similar to XDL to describe the physical resources on a Xilinx FPGA. These files are self-documenting and provide a very detailed description of the routing and logical structures of the FPGA down to the BEL level (without timing data). Consequently, XDLRC files are used by external CAD frameworks such as RapidSmith and Torc to populate their device data structures. These

device representations may then be accessed through the framework's API to obtain vital resource information for CAD tools such as placers and routers.

XDLRC reports are generated by ISE's `xdl` tool using the following command: `xdl -report [-pips] [-all_conns] <part> [<filename>]`. When the `-pips` and `-all_conns` switches are used, `xdl` prints out all PIP and wire connection descriptions respectively, which are both needed to extract a complete device description.

XDLRC files contain individual entries for each tile on the part, making the generated file rather large in size. The files for the largest 7 Series devices are well over 70 GB in size. This is largely due to the verboseness of the files and lack of optimizations in the device description. For example, all sites of the same type have the exact same pinout, yet a XDLRC file still lists every pin for every single site on the device.

An XDLRC file may be divided into two parts, the device description and the primitive definitions. This division parallels the site-based nature of XDL as the device description covers logical and routing components down to the site level, while the primitive definitions section provides sub-site information for all different types of sites that may exist on the device. A more in depth description of either portion is provided in the ensuing sections for the reader's convenience.

Device Description

Listing 2.1: Structure of the device description section in an XDLRC report file

```
1 (xdl_resource_report v0.2
2 xc7k70tfbg484-3 kintex7 (tiles 209 117
3     ...
4     (tile 119 10 CLBLL_L_X2Y85 CLBLL_L 2
5       (primitive_site SLICE_X1Y85 SLICEL internal 45
6         (pinwire A1 input CLBLL_L_A1)
7         ...
8       )
9       ...
10      (wire CLBLL_NE4BEG0 11
11        (conn INT_R_X1Y85 NE6BEG0)
12        ...
13      )
14      ...
15      (pip CLBLL_L_X2Y85 CLBLL_IMUX6 -> CLBLL_L_A1)
16      ...
17      (tile_summary CLBLL_L_X2Y85 CLBLL_L 90 298 146)
```

```
18      )
19      ...
20    )
21    (summary tiles=24453 sites=20665 sitedefs=86 numpins=90 numpips=146)
22 )
```

The device description portion of an XDLRC file roughly follows the device element hierarchy outlined in section 2.1.1. At the top level, is the `xdl_resource_report` element, which provides basic information on the FPGA part, including its name and family. Within this element, is a set of `tile` elements, one for every tile on the device. A `tile` element provides the tile's name, its coordinates, and type. Within the tile element are three lists: a list of `primitive_site` elements, a list of `wire` elements, and a list of `pip` elements. Together, these lists describe the logical and routing structures within the tile. A `primitive_site` element provides a site's name, type, and contains a listing of `pinwire` elements. Each `pinwire` represents a site pin and provides its name, direction, and the name of the wire outside of the site that it connects to. A `wire` element provides the wire's name and a list of its connections in tile/wire name pairs. A `pip` element simply describes a single programmable connection between two wires in the same tile. This information offers a complete description of a device down to the site level.

Primitive Definitions

As can be observed from the previous section, the device description does not provide any information below the site level. However, section 2.1 makes it very clear that BELs and a network of intrasite routing exists within a site's hierarchy. The primitive definitions component of an XDLRC file provides a complete description of the resources within a site.

Listing 2.2: Structure of the primitive definitions section in an XDLRC report file

```
1 (primitive_defs 86
2   ...
3   (primitive_def SLICEL 45 140
4     (pin A1 A1 input)
5     ...
6     (element D6LUT 7 # BEL
7       (pin A1 input)
8       ...
9       (cfg <eqn>)
```

```
10          (conn D6LUT O6 ==> DUSED 0)
11          ...
12        )
13        ...
14    )
15    ...
16 )
```

At the top of the hierarchy is the `primitive_defs` element which simply contains a number of `primitive_def` elements. Each `primitive_def` element represents a different site type and has a name (site type) and two lists: a list of `pin` entries and a list of `element` entries. A `pin` represents a site pin and contains an internal and external name, as well as a direction. An `element` is somewhat complicated as it can represent a number of things, including BELs, site pins, site PIP muxes, site route-throughs, and configuration properties. In any case, an `element` will have a name and may have any combination of these three lists: a list of `pin` entries, a list of `conn` entries, and/or a list of `cfg` entries. In this context, a `pin` will have just a name and direction, and refers to either a BEL pin, site PIP, or site pin, depending on what the parent `element` represents. A `conn` represents a connection between two `element`/`pin` pairs. It should be noted that `conn` elements completely describe the sub-site routing structure. Finally, the `cfg` element contains a list of configuration strings.

2.3 FPGA CAD Tool Frameworks

This section provides some examples of CAD tool frameworks that currently exist because of the items discussed thus far in this chapter. Specifically, these CAD frameworks target Xilinx FPGAs and are built on top of the XDL interface, and therefore are compatible with the ISE design suite. First, RapidSmith will be discussed in depth as it is the topic of further discussion later on in this work. Other frameworks will then be briefly mentioned. Ultimately, these are offered as an example of the benefits of custom CAD tool frameworks that can target actual Xilinx FPGAs.

2.3.1 RapidSmith

RapidSmith was developed at Brigham Young University in conjunction with a productivity tool called HMFlow [12]. It provides a Java-based API that leverages XDL for the rapid creation of custom FPGA CAD tools. Through the `xdl` executable, RapidSmith can be used to modify designs at any point in ISE's design flow, or even create one from scratch. RapidSmith also provides functionality for mapping a design to a device, and provides high-level functions to accommodate the creation of placers and routers.

RapidSmith was specifically designed to run quickly and efficiently, and so a number of optimizations have been applied to its structures to allow CAD tools to run quickly and efficiently. As a result, in some cases efficiency was chosen over clarity when implementing its data structures. That being said, a secondary priority was ease of use. The Java programming language was chosen with this in mind because it offers an easy-to-use API and automated garbage collection.

RapidSmith consists of a number of Java packages offering a wide range of functionality, all of which are based on the `design` and `device` packages. These packages help to alleviate the challenges of using XDL and establish an API to interface with the ISE tool flow through the XDL interface. The following subsections provide additional information on these packages to establish the platform with which current and future iterations of the Xilinx EDA tool flow will need to interface with in order to support custom CAD tool frameworks.

Design Package

As mentioned previously, the `xdl` executable may be used to extract design descriptions in the XDL format. RapidSmith parses these XDL files and uses them to populate a design database in memory. This database may then be operated on via the RapidSmith API, written to XDL, and inserted back into the ISE tool flow.

Naturally, the major classes in the `design` package mirror the declarations found in an XDL file, namely `Design`, `Instance`, and `Net`. A `Design` object will have a name, an FPGA part, a set of `Instance` objects, and a set of `Net` objects. An `Instance` object has a name, type, site (if placed), and a map of attributes. A `Net` object has a name, source pin, a

set of sink pins, and a set of PIPs (if routed). Aside from these main classes, there a several supporting classes that make traversing the netlist easier such as `Pin`, `Port`, and `PIP`.

Currently, like XDL, RapidSmith's netlist representation is site-based, meaning the most atomic component represented in its netlist is the site. Hence, `Instance` objects map directly to sites rather than BELs, while BELs within a site are programmed by the set of attribute strings stored in an `Instance`. This could present some difficulties when implementing a technology mapper or placer in RapidSmith for a gate-level netlist, as some sort of a packing tool would be required.

Device Package

The `device` package is closely involved with the `design` package as it comprises the database of resources on an FPGA. The major classes in the `device` package include `Device`, `Tile`, `PrimitiveSite`, and `WireEnumerator`. RapidSmith's device representation loosely follows that of an XDLRC file, with a few exceptions where it deviates to apply performance optimizations to the data structure. This being the case, this section will not reiterate the information that is already available in section 2.2.2. Instead, it will outline any major differences in the device representation.

The first difference has to do with the site-based nature of RapidSmith. Since the RapidSmith leaves all sub-site programming up to the user (through attribute strings), it has no need for the primitive definitions section of an XDLRC file. Therefore, this information is discarded by RapidSmith.

The next difference deals with limitations of XDLRC files. Certain types of instances may be placed on more than one type of site. A common example is placing a `SLICEL` instance on either a `SLICEL` or `SLICEM` site. This sort of information is not provided by XDLRC files, since they are only concerned with physical resource information, not logical to physical mappings. To remedy this, RapidSmith is hard-coded with this information, which is invaluable when writing a placer.

The final difference involves various optimizations that were made to increase performance and efficiency. Considering the enormity of XDLRC files, it is impractical for

an FPGA CAD framework to reference these files each time the tool is run. RapidSmith employs three major strategies for decreasing database size and load time:

1. *Wire and Object Reuse*: The majority of an XDLRC file's size may be attributed to wire declarations, many of which are superfluous. Given the regular nature of FPGAs, the same wires often appear in different tiles all over the part [5]. In fact, a wire that is present in a tile of some type is more than likely to appear in other tiles of the same type. Therefore, rather than instance a separate wire object for the duplicate wires in every tile, RapidSmith creates an enumeration for one unique wire that all identical wires reference. This is done in the `WireEnumerator` class by storing a wire's name and mapping it to an integer. Connections between wires are represented by mapping the source wire's enumeration to the sink wire's enumeration and its tile offset, thus removing the reference to any one tile on the chip.

2. *Wire Graph Pruning*: For every single node in a device, an XDLRC file reports every single wire that node traverses. This includes wires whose purpose is to simply carry the node across a tile, but otherwise provide no relevant routing information (i.e. they provide no connections to the node). For instance, on the xc7k70t part there is a node called `INT_L_X16Y84/WW4BEG2` that has a source in tile `INT_L_X16Y84` and a single sink in tile `INT_L_X12Y84`. The node begins and ends in wires `WW4BEG2` and `WW4END2` respectively, but in the process traverses nine other wires (across nine tiles), all of which are included in the XDLRC file. During the process of generating its device representation from XDLRC, RapidSmith removes these extra wires, greatly reducing the memory footprint of its device representation.

3. *Serialization and Compression*: Rather than parse XDLRC files every time it is loaded, RapidSmith uses custom serialization and compression routines to save its device database to a compact device file, eliminating the need to generate and parse a device's XDLRC every time the tools are run.

With the combination of these strategies, RapidSmith enjoys very fast load times (approximately 3 seconds for the largest Virtex 7 part) and significant file compression (on the order of 10,000X compression), making it an idea framework for CAD tool implementation [11].

2.3.2 Other Examples

Tools for Open Reconfigurable Computing (Torc)

Tools for Open Reconfigurable Computing (Torc) [6] is a C++ based open source CAD tool framework developed by the Information Services Institute at the University of Southern California (USC-ISI). Like RapidSmith, TORC uses XDL and XDLRC to populate its own data structures and provides an API that allows users to manipulate designs. Unlike RapidSmith, Torc provides a more comprehensive tool set which provides for technology mapping and packing. More specifically, Torc is able to unpack XDL instances into a representation at the BEL level. It is currently implementing a feature to pack this representation back into an XDL instance. RapidSmith, on the other hand, only provides support down to the XDL instance level and expects users to handle all sub-site manipulations via attributes passed to the instance.

FPGA Analysis Tool (FAT)

The FPGA Analysis Tool (FAT) [7] is a Python-based framework for analyzing Xilinx designs and devices. FAT uses XDLRC to populate its device representations and XDL for its design data structure. Functionality is defined using a number of scripts called "recipes." A recipe utilizes any number of low-level FAT modules to implement high-level algorithms. Using the information gleaned from XDL and documentation provided by FPGA vendors, FAT is able to offer a low-level bitstream API. This provides additional functionality for processing bitstream packets and decoding bitstream headers and frames.

Chapter 3

The Vivado Design Suite

Xilinx has recently introduced Vivado as its new EDA suite of FPGA implementation tools [10]. With Vivado came many changes, one of the most notable being the discontinuance of the Xilinx Design Language (XDL). Without XDL, tools such as RapidSmith, HMFlow, and TORC will not be compatible with Vivado and future generations of Xilinx FPGAs. This chapter presents Vivado with an emphasis on functionality directly related to implementing CAD tools within its environment. It seeks to answer the question: *"Does Vivado provide the basic functionality needed to support a custom CAD tool framework similar to RapidSmith or Torc?"*

Many of the features presented in this chapter are covered by Xilinx in Vivado's official documentation. These items will merely be presented in conjunction with the specific CAD tool application they supplement, and the reader is encouraged to review the appropriate documentation. There is, however, additional functionality that is (at the time of this writing) undocumented by Vivado and is only available through a thorough study of the Vivado tool. These resources will be covered in depth by this chapter, in enough detail to ensure that the reader is able to utilize their functionality. Ultimately, the goal of this chapter is to offer a cache of information that is pertinent to the implementation of a custom CAD tool framework for the future of Xilinx FPGAs.

3.1 Overview

At the time of this writing, Vivado 2014.2 provides support for 7 Series and Ultrascale Xilinx FPGAs. The introduction of the Ultrascale line sees several devices that reach well into the millions of logic cells. This is a significant increase over devices supported by ISE[1], which

[1]Virtex 4 offered up to 200,000 logic cells, Virtex 5 up to 330,000 logic cells, and Virtex 6 offered up to 750,000 logic cells

has led Xilinx to redesign its EDA tool suite. Xilinx reports that the Vivado design suite was built from the ground up to provide support for the next generation of FPGA architectures and to accelerate design productivity [10]. As a result there are many differences from its successor, ISE. The primary components of Vivado that are worth mentioning include its interface, its standard flow for compiling designs, its checkpoint capability for saving and loading designs, and the various productivity mechanisms available in Vivado.

3.1.1 Interface

Vivado allows users to interact with its IDE through a variety of modes: GUI, Tcl, or batch. The user may indicate which mode is used when starting Vivado using the `-mode` option. This allows the user to interface with the tool in a number of ways, including through scripts and a command line interface.

GUI Mode

Vivado defaults to GUI mode, which loads the design environment skinned in a fully featured GUI through which the user may interact with the tools and design space. This GUI is rather reminiscent of ISE's PlanAhead in its layout, though there are some differences. One such difference is that Vivado's device browser provides a much more detailed view of the physical FPGA than PlanAhead. PlanAhead's view of the FPGA excludes the routing network, whereas Vivado's device browser depicts all objects from its data structure that are accessible to the user and more, similar to FPGA editor. To clarify, wires and nodes are not visible in PlanAhead, but are in Vivado. Vivado will also display objects that are not accessible to the user through its Tcl interface, such as the routing resources inside of a site. This is useful when the user wishes to view the innards of a site without needing to look up the official documentation.

Tcl Mode

Tcl mode opens a command line interface (CLI) that is built on top of a Tool Command Language (Tcl) interpreter (more information on this is provided in section 3.2). This Tcl interpreter is augmented with a set of Tcl commands that may be used to control Vivado

and its tool flow. Commands called in Tcl mode run can run anywhere up to 5 times faster than when called in GUI mode. In fact, most if not all operations in GUI mode are actually executing commands in Vivado's Tcl Interpreter, which may be found in a window at the bottom of the GUI. The user could copy whatever commands are recorded there while in GUI mode, restart Vivado in Tcl mode, paste those commands into the CLI, and achieve exactly the same functionality. That being said, most, if not all, Vivado-related procedures referenced in this work were done in Vivado's Tcl mode, bypassing the GUI altogether.

Batch Mode

Batch mode runs whatever script it is handed and then quits. This script must be composed in Tcl, drawing from native Tcl or Vivado's own set of Tcl commands. A script may be executed by starting Vivado with the following parameters: `vivado -mode batch -source <script>.tcl`. Test show that commands in scripts run via batch mode are as fast as commands run in Tcl mode.

3.1.2 Compilation Flow

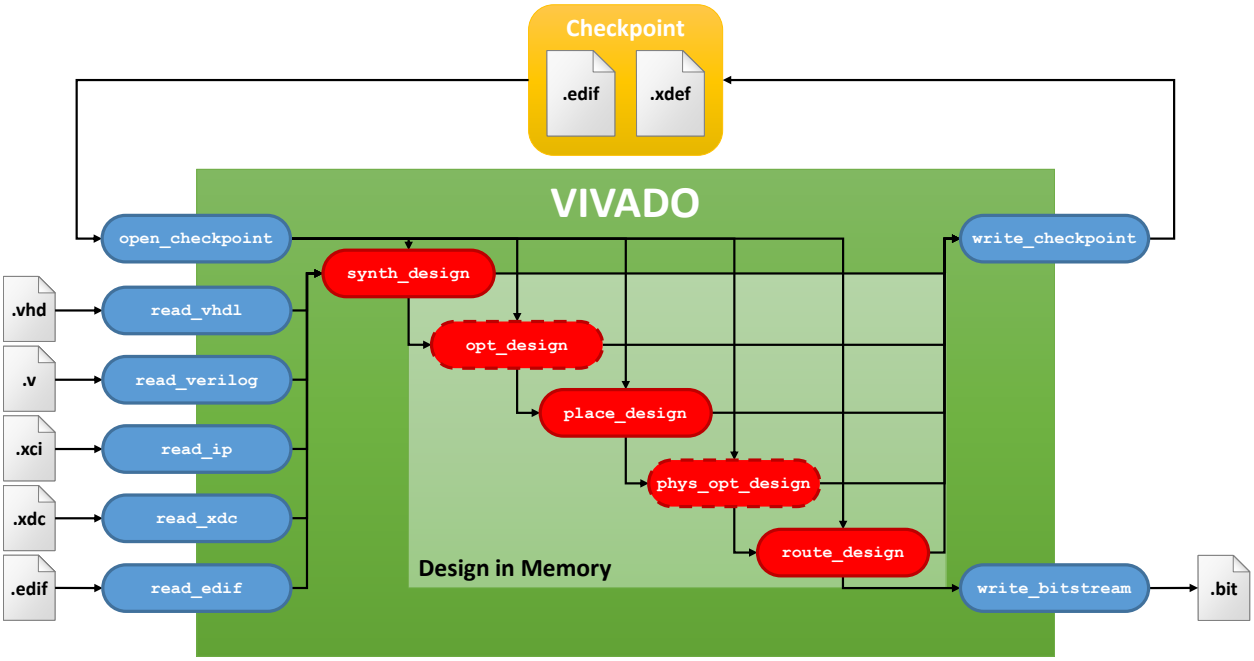

Figure 3.1: Depiction of Vivado's CAD tool flow with file I/O

Vivado's compilation flow is not a file-to-file process like ISE, rather the Vivado environment encapsulates every CAD tool and operates on the design representation in-memory. Generally it involves parsing a design file, synthesizing it into a working netlist, and then implementing it on a given FPGA part. Figure 3.1 diagrams Vivado's compilation flow, including its primary file input/output mechanisms.

As can be seen Figure 3.1, Vivado has several commands for reading in various design files such as RTL (VHDL and Verilog), IP (Xilinx XCI files), constraint files (Xilinx Design Constraint or XDC format), or third party netlist files (in the EDIF format). Once these files have been read into Vivado, if they comprise a valid design, Vivado can then synthesize them into a netlist and/or implement them onto a part using the commands `synth_design`, `place_design`, and `route_design`. Optimizations to the design may be made using the commands `opt_design` and `phys_opt_design`. Each of these commands is a Tcl command that serves as a wrapper for one of Xilinx's proprietary CAD tools. Once the design has been implemented, a bitstream may be written using the `write_bitstream` command and then downloaded to the board.

3.1.3 Design Checkpoints

As figure 3.1 shows, at any point along the compilation flow of a design, the user may write out to disk what is referred to as a "design checkpoint" (.dcp) file using the `write_checkpoint` command. Similarly, a design checkpoint may be read into Vivado using the `open_checkpoint` command, effectively restoring the state of the design to where it was when the checkpoint was written. In other words, design checkpoints are able to save the state of a design at any point along the design flow:

1. Linked Design: A part has been loaded into memory. No netlist exists.
2. Post-synthesis: RTL has been parsed and synthesized into a netlist.
3. Post-placement: The netlist has been optimized and/or placed on the FPGA fabric.
4. Post-routing: Timing driven optimizations have been applied and/or the netlist has been routed.

Checkpoints themselves do not add any features to Vivado, but provide a convenient means of saving and restoring entire designs to Vivado with a single file. Checkpoint files may be thought of as snapshots of a design as it exists in memory at the time it is written.

Checkpoints are of particular interest to the development of CAD tools for Vivado because they offer a fast way of transporting designs in and out of the Vivado environment. Tests have shown that Vivado design checkpoints can be written from and read into Vivado in linear time, which is significantly better than current alternative methods (see section 5.2 for more details).

Composition

A checkpoint file is actually just an archive containing a collection of files that describe the design's netlist, constraints, placement, and routing information. These files can be extracted using typical file archiving software such as the open source 7-Zip tool. A checkpoint archive includes the following files:

- `dcp.xml`: Checkpoint information. This file contains information on the checkpoint, including the design name, part name, whether or not it is out-of-context (see section 3.1.4), and a listing of every file in the checkpoint along with various properties.

- `<design>.edf`: The design's netlist. This file may be encrypted if any of its source files are encrypted (i.e. IP). Otherwise, this file is equivalent to the file output by the `write_edif` command with a few differences: 1) Some (but not all) ports, instances, and/or views have an additional property called `XLNX_LINE_FILE`, 2) some nets are renamed, 3) the design element has an additional property called `FILE0`, and 4) some lines are reordered.

- `<design>.xdc`: Design constraints. This file contains all the constraints applied to the design. The file is written in the Xilinx Design Constraint (XDC) format, which is essentially a Tcl script that sets various netlist object properties. This file is identical to the file output by `write_xdc` except for a couple differences: 1) it begins with a command to set the design's `SRC_FILE_INFO` property to the absolute and relative file paths of the file(s) that contain any user defined constraints, and 2) every single line

(including comments) is preceded by a command that describes where that line came from (i.e. which file and line number) via the design's `src_info` property.

- `<design>.xdef`: Placement and routing information. Xilinx Design Exchange Files (XDEF) are Xilinx's proprietary format for representing the placement and routing information of an implemented design. This file is encrypted and undocumented by Xilinx, although it is mentioned briefly in various locations throughout official documentation and the Vivado tool itself. XDEFs can be written from and read into Vivado using the undocumented Tcl commands `write_xdef` and `read_xdef` respectively. These generated files are identical to the files found in design checkpoints.

- `<design>_stub.v`: Port declarations. This file contains a Verilog module that simply declares all the top-level ports in the design. This is referenced when Vivado needs to infer a black box for IP.

- `<design>_stub.vhdl`: Port declarations. Same as the preceding file, but written in VHDL instead.

- `<design>.incr`: Contains timing data for Vivado's incremental flow feature (see section 3.1.4).

- `<design>.psr`: In all cases studied, this file has been blank.

- `<design>.wdf`: This file begins with a "version" tag and ends with an "eof" tag. In between are several lines consisting of six hexadecimal numbers separated by colons. Its function is currently unknown.

3.1.4 Productivity Mechanisms

Vivado provides a number of mechanisms in its tool flow for increasing a designer's productivity. In this context, "productivity" refers to the amount of work a hardware designer can accomplish in a given amount of time. This corresponds to the performance of a tool flow and the tools it consists of. Since the performance of CAD tools falls under the purview of a CAD tool framework, they will be briefly introduced in this section. References to their respective documentation are provided for the reader's convenience.

Incremental Compile

Incremental Compile (also known as Incremental Implementation) uses design checkpoints saved at previous states of implementation to guide subsequent implementations. Its function is similar to the SmartGuide feature from ISE. More can be learned on this mechanism in [13].

Out-of-Context (OOC) Checkpoints

Vivado offers a hierarchical design flow that employs the use of out-of-context (OOC) modules. OOC modules are sub-designs that have been synthesized and possibly implemented at a fixed location on an FPGA. These modules can then be imported into a top-level design (as long as none of their resources overlap) and stitched together using Vivado's EDA tools. This way, when a change needs to be made to the combined design, only the affected module need be re-compiled. This process is similar to ISE's partition flow. For more information, see [14] and [15].

Checkpoints are used to store OOC modules in Vivado. When a checkpoint is saved as OOC, a flag is set in its `dcp.xml` file and a variety of special measures are taken to prepare the checkpoint to be imported into a design (such as not inserting IO buffers). An OOC checkpoint may be read into a cell in a design using the Vivado command `read_checkpoint -cell <cell_name> <file>`, which will import the OOC module described in `file` into the cell `cell_name`.

Relatively Placed Macros

Vivado still provides support for relatively placed macros (RPMs). These are sets of cells with a number of relative placement locations defined between them. When the macro is placed, the cells in the macro are placed according to the relative locations outlined for them. Macros are similar to OOC modules in that they may be used to define placement information for group of cells, but unlike OOC modules, these groups of cells may be moved anywhere on the device that supports the placement constraints of the macro.

Vivado also introduces what are called XDC macros. These operate in much the same way as RPMs except for a few differences, the most notable being that RPMs are defined

in HDL, while XDC macros are defined by an XDC constraint. For more information on macros, see [16].

3.2 Tcl Interpreter

Perhaps Vivado's most important feature when considering a CAD tool framework is its Tcl shell. Xilinx has incorporated a fully featured Tcl 8.5 interpreter with every version of Vivado, allowing users to manipulate designs through either a command prompt or through sourcing scripts written in Tcl. The Tcl shell can be accessed through any interface to the IDE. In GUI mode, the bottom pane has a "Tcl Console" that displays the shell's standard out, as well as offers a text box that pipes to its standard in. In Tcl mode, the shell is loaded into the terminal. In batch mode, the provided script is sourced into Vivado's Tcl shell.

3.2.1 Tcl Command Set

The Tool Command Language (or Tcl) is a very capable dynamic programming language that was developed for controlling and extending applications. It has no fixed grammar and is defined by a set of substitution rules and an interpreter that parses individual commands. Tcl offers familiar control constructs such as `if` and `for`, however these features among others (such as control flow, procedures, and expressions) are not understood by the interpreter, but are implemented as commands [17]. For instance, new commands may be defined at any time using the command `proc <name> <args> <body>`, where `proc` is the Tcl command for defining a new command, `<name>` is the name of the new command, `<args>` is a list of arguments being passed into the new command, and `<body>` is the body of the procedure.

Tcl Standard Library

In addition to the basic set of packages included in Tcl 8.5, Vivado also comes installed with the Tcl Standard Library. The Tcl Standard Library (or TCLLIB) offers a number of packages that extend Tcl's native functionality by adding a wide variety of features related to programming, mathematics, data structures, text processing, encryption, and more. Of the most relevance to a CAD tool framework are the `stooop` and `struct` packages. The Simple

Tcl Only Object Oriented Programming (`stooop`) package is a light weight object-oriented extension that follows the C++ syntax while still adhering to the Tcl philosophy. It allows users to create classes in Tcl, with their own data members and member functions. Features such as inheritance are also supported. To use `stooop` in Vivado, the user simply need to include the package before using any of its commands. This can be done using the command `package require ::stooop`.

The `struct` package provides a number of data structures that have been implemented in Tcl and may be included using the `package require ::struct` command. It adds the following data structures to the Vivado Tcl environment:

- `list`
- `set`
- `stack`
- `queue`
- `prioqueue` (priority queue)
- `skiplist` (similar to binary trees)
- `tree`
- `disjointset` (merge-find)
- `graph`
- `record` (similar to a `struct` in C)
- `matrix`
- `pool`

Many of these data structures are implemented using native Tcl constructs such as `list`. The advantage to this is that commands from the `struct` package may be used to operate on these native constructs. For example, if the user wishes to find the intersection of two lists `a` and `b`, they can use the command `struct::set intersect $a $b`.

3.2.2 Vivado Command Set

In addition to the command set provided by Tcl 8.5 and the TCLLIB packages, Xilinx has incorporated its own set of commands into Vivado's Tcl Interpreter to provide unprecedented access to Vivado's IDE, CAD tools, and data structures. These low-level commands combined with rather detailed design and device databases present a rich foundation upon which to construct a custom CAD tool framework for present and future Xilinx FPGAs.

Vivado 2014.2 has a total of 539 documented commands in its global namespace that are spread across 32 categories [18]. A full description of any command can be obtained in Vivado using the `-help` switch. Four categories are significant when considering a CAD tool framework: Object, Netlist, Tools, and FileIO.

Object

As section 3.3 discusses, a subset of objects from Vivado's internal data structures have been exposed through its Tcl interface. These components are wrapped in Tcl object classes (built using TCLLIB's `::stooop` package) and may be accessed using a number of commands from this category. Vivado's Object commands deal with general access and manipulation of Tcl objects and properties.

A number of "getter" functions are listed under this category to allow the user to get pointers to these objects. These are typically prefixed by `get_` and followed by the name of the class of the objects you wish to obtain, such as `get_cells`, `get_bels`, and `get_designs`. By default, a Tcl list of all objects of that class are returned, though a pattern may be applied to retrieve objects with certain names. A powerful feature of these getter functions is the `-filter` switch, which allows the user to filter out the objects returned based on a search pattern that is applied to their properties. For example, if one wishes to obtain all unused BELs that are LUTs in a device, they can use the command `get_bels -filter {!IS_USED && TYPE=~*LUT*}`.

There are also a set of relationships that exist between objects in Vivado. Figure 3.2 depicts the relationships for the most common objects in Vivado. In this figure, an arrow represents that a "has-a" relationship exists between the head and tail of the arrow — in other words, $\mathbf{A} \rightarrow \mathbf{B}$ implies that "A has a B." These relationships can be accessed using the `-of_objects` switch with the appropriate getter function. For example, figure 3.2 shows a relationship exists between `cell` objects and `bel` objects, or that "a cell has BELs." For a cell stored in the variable `cell`, its BELs can be gotten using the command `get_bels -of $cell` (where `-of` is an abbreviation for `-of_objects`). There are some problems with the relationships in Vivado, as can be seen in figure 3.2. Namely, there are some incomplete object relationships. For example, it is possible to obtain the list of BEL pins of a BEL

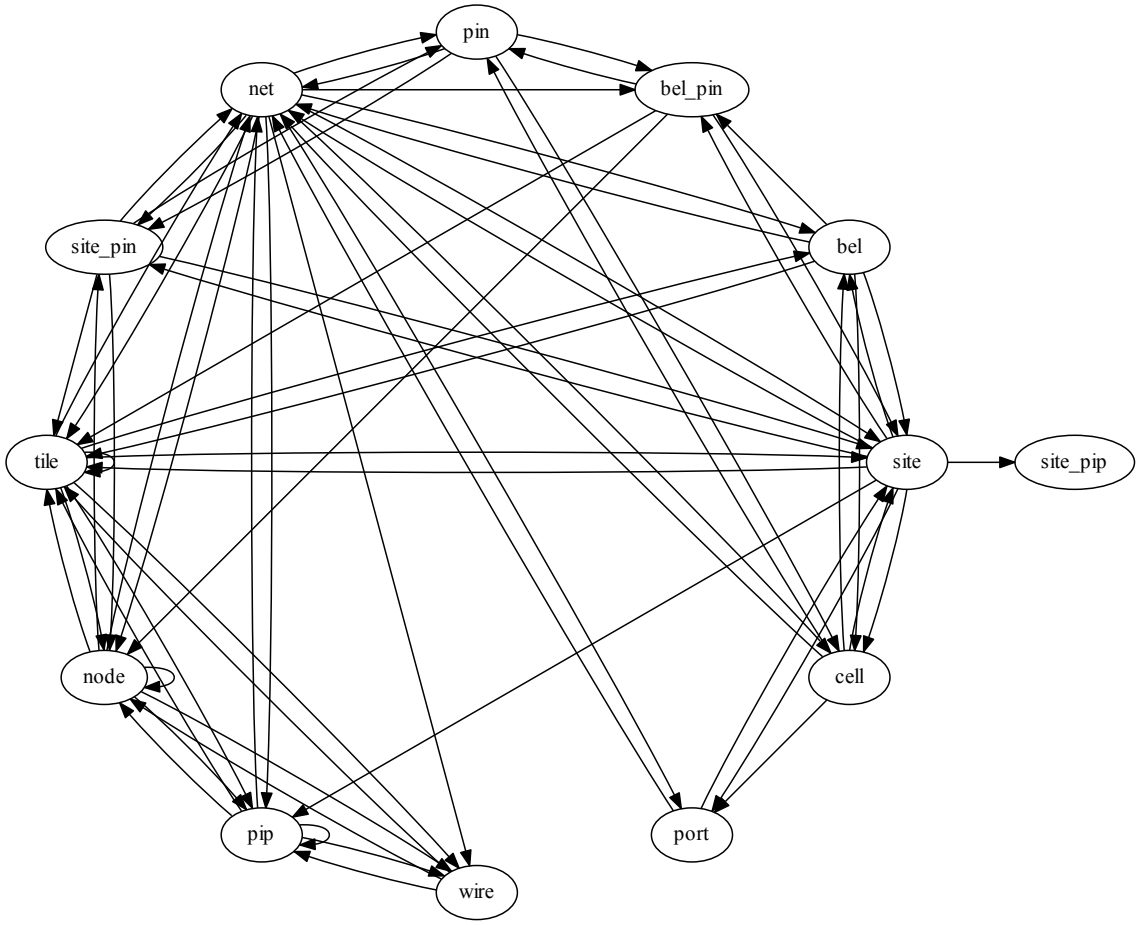

Figure 3.2: Depiction of some common object relationships in Vivado

(`get_bel_pins -of $bel`), but it is not possible to get the BEL of a BEL pin using similar commands.

Each Tcl object in Vivado has a set of properties, which can be accessed using the `get_property` command. The `set_property` command allows an object's property to be changed, provided it is not read-only. Generally, device-related objects have all read-only properties, though there are a few exceptions. Properties are the means by which Vivado represents any information about an object, other than its relationships. This includes placement, routing, or even LUT equations. A useful command is `report_property -all`,

which will print out a report of all the properties of an object, including their data type and whether or not they are read-only.

Netlist

The Netlist category encompasses all commands used to make real-time changes to Vivado's netlist representation. This includes commands for creating netlist objects (including `create_cell`, `create_pin`, `create_port`, and `create_net`), as well as removing them (`remove_cell`, `remove_pin`, etc.). It also includes commands for manipulating the connectivity of the netlist, such as `connect_net` and `disconnect_net`. Using these commands, the user can create a Vivado netlist in its Tcl interpreter from scratch, a process that discussed further in section 3.4.1.

There are some subtle nuances that the CAD tool designer need be aware of when manipulating a netlist in Vivado. These mostly have to do with how to add cells and pins to existing cell objects. First of all, adding children cells to a hierarchical cell is done using the `create_cell` command, however when naming the new cell, rather than passing the command the cell's relative name, the user must offer the cell's hierarchical name. For example, to add a child cell called `and2` to a parent cell called `full_adder`, the user must use the command `create_cell -ref LUT2 full_adder/and2`. Adding pins to cells must be done in the same manner. To add an input pin called `addr3` to a cell called `dma_controller`, the user must use the command `create_pin -dir IN dma_controller/addr3`.

Tools

Commands in the Tools category are simply Tcl wrappers for Vivado's proprietary EDA tools. Commands include `synth_design`, `place_design`, and `route_design`. Each of these Tcl commands are richly featured, and can be tweaked to the user's discretion using a number of switches and options for them. For example, if the user wanted Vivado's router to route all the nets whose name began with the letter 'h' as quickly as possible, he or she could use the command `route_design -directive Quick -nets [get_nets h*]`.

In the context of CAD tool design, these tools are useful for completing operations along the standard tool flow that any custom CAD tools neglect to handle. For instance,

the user may wish to manually place most of a design, and have Vivado's placer finish off the rest of the design by calling `place_design`.

FileIO

The FileIO category covers all Tcl commands in Vivado that deal with reading and writing files to and from Vivado's IDE. As can be seen in Figure 3.1, Vivado accepts a number of file formats into its design flow. Of these formats, a couple are worth discussing in more detail. The `write_edif` and `read_edif` commands allow an EDIF file to be generated from and read into Vivado, respectively. When an EDIF file is read into Vivado, it is able to completely populate Vivado's netlist structure with `cell`, `pin`, `port`, and `net` objects. In fact, an EDIF written out by Vivado contains all information needed to completely restore the design's netlist in Vivado using just the `read_edif` command.

The second pair of commands is the `write_xdc` and `read_xdc` commands. The `write_xdc` command will generate a file of Tcl commands that apply constraints on a design, typically dealing with IO and timing. What's interesting is that `read_xdc` has the ability to execute a wider range of functionality than files generated by `write_xdc` would suggest. For example, any property on any Vivado object may be set by an XDC file using the `set_property` command. The `read_xdc` command is also able to execute the commands in an XDC file faster than if they were entered into the Tcl interface manually. This will be discussed in further detail in section 3.4.1.

3.2.3 Additional Commands

In addition to the command sets described above, Vivado offers a number of commands across several undocumented namespaces. It is also possible to extend Vivado with custom defined namespaces and commands.

Undocumented Vivado Namespaces

The commands mentioned above are well documented by Xilinx in [18] and exist in the Tcl global namespace. In addition to these commands, there are thousands of other commands that exist in a dozen other namespaces that Vivado has added to its shell. These

can be discovered by calling the Tcl command `namespace children`. Due to the sheer number of these undocumented commands, they will remain largely undiscussed in this work unless they have been found to serve a purpose directly related to implementing a CAD tool framework in Vivado. Their namespaces are listed here for the reader's convenience and a brief description (if known) is provided.

`cli` Add/hide commands, check arguments

`cmdline` Usage statements and options

`config` Load/print configs

`debug` Debug versions of commands

`device` Getters for device objects

`dir`

`ipx` Returns IP cores in the design

`pkg` Package creation

`rdi` Controls aspects of the Tcl shell

`rt` Very low-level Tcl commands

`rtdc` Manipulate attributes and properties

`sdc`

`sdcrt`

`tcl` Various native Tcl commands

`tclapp` Namespace for Tcl store packages

`utils`

In general, the user should use commands from these namespaces with caution, as Xilinx may not provide long term support for them. In fact, over the various releases of Vivado thus far, several of these commands, and even entire namespaces, have been removed altogether. One such instance worth mentioning is the `internal` namespace, which was removed from Vivado in its 2013.3 iteration. The `internal` namespace provided detailed information about Vivado's underlying device representation, allowing users to obtain handles to "template" objects that defined the structure of types of tiles, sites, and BELs. The information from the `internal` namespace could be used to infer routing information within a site, which is otherwise unobtainable from Vivado. Unfortunately, Vivado 2013.3 and later has had this information removed or hidden from the Tcl interpreter. Due to the uncertainty of the permanence of these undocumented commands, they will be avoided for the remainder of this work.

Extending Vivado's Command Set

Packages of Tcl commands and variables may be loaded into the interpreter at any time using the `package require <package name>` command. The path to a package must be defined in the file `pkgIndex.tcl`, which is automatically run at the start of the Tcl interpreter. In Vivado, this file should be in the directory `<Xilinx install path>/Vivado/<version>/tps/tcl/tcl8.5`.

3.3 Data Structure

Elements from Vivado's internal data structure are wrapped Tcl objects that are accessible through Vivado's Tcl interface. Every one of these Tcl objects has a set of properties that describe their unique attributes. Each property has a name, value, data type, and is either read/write or read-only. These properties may be queried and manipulated using the Tcl commands `get_property` and `set_property`, respectively. Additionally, each object has relationships with other objects that may be accessed by using "getter" commands in Vivado's Tcl interpreter. For instance, a cell object (stored in the variable `cell`) has a set of pin objects, which may be accessed using the command `get_pins -of_objects $cell` (see section 3.2.2).

Objects in Vivado's data structure may be divided into two categories, design objects and device objects. The following sections provide a brief overview of objects in each of these classifications and their relationships amongst one another.

3.3.1 Design Objects

Vivado's netlist data structure is loosely based on the EDIF design language. Simply put, Vivado's design data structure consists of a set of cells interconnected by nets. While there are several classes involved in describing a design in Vivado (see table 3.1 for a full listing), this section will briefly describe the major components, which includes cells, pins, ports, nets, and their library counterparts.

Table 3.1: Listing of all design objects in Vivado

Logical Component	Vivado Class
Cell	`cell`
Clock	`clock`
Design	`design`
IO Interface	`interface`
Intellectual Property	`ip`
Library	`lib`
Library Cell	`lib_cell`
Library Pin	`lib_pin`
Macro	`macro`
Net	`net`
Partition Block	`pblock`
Pin	`pin`
Port	`port`

Cell

A **cell** in Vivado is represented by the `cell` class. Every cell references a library cell when it is created. Like EDIF, a library cell is a template from which a cell instance inherits its properties and pins. When a cell instances a library cell, its property map is populated with the library cell's list of properties and their default values. Among these properties is the cell's name, which is passed in as a parameter when the cell is created. The cell is also given a pointer back to the library cell, which can be accessed using the `get_lib_cells -of $cell` command. Library objects are discussed in greater detail later in this section. For simplicity, the remainder of this work will refer to the library cell referenced by a given cell as the "type" of that cell.

Cells are a hierarchical construct, and may contain other cells. By default, only top-level cells are returned by the `get_cells` command. To get a list of all cells, including those buried within levels of hierarchy, the `-hierarchical` switch needs to be used with the `get_cells` command. Alternatively, whenever the `get_cells` command is passed a name pattern that includes the hierarchy separator character, such as `get_cells "*[get_hierarchy_separator]GND"`, all children cells are searched for the pattern.

Oddly enough, the list of children cells of a cell cannot be obtained using the `get_cells -of $cell` command in Vivado Tcl. Instead, children cells are given a hierarchical name,

consisting of the parent cell's name and the cell's unique name divided by the hierarchy separator character. To get all the children cells of a cell, all cells must be filtered by their names like so: `get_cells "[get_property NAME $cell][get_hierarchy_separator]*"`.

Cells at the lowest level of hierarchy are referred to as leaf cells. These cells cannot be broken down into smaller cells and must reference a primitive ("leaf") library cell. A leaf cell maps directly to a BEL during the placement process and contains the set of properties (if any) for programming that BEL. Internally, Vivado has a mapping of primitive library cells to the list of BELs they are compatible with, but does not share any of this information with the user. The only information Vivado provides the user concerning cell/BEL compatibility is the error it throws when a cell is placed on an incompatible BEL. For example, the `LUT2` library cell belongs to the `UNISIM` library and has 3 library pins, `LUT2/I0`, `LUT2/I1`, and `LUT2/O`. A cell may be created that references this library cell using the command `create_cell -ref [get_lib_cells LUT2] xor`. A cell named `xor` is created that has the pins `xor/I0`, `xor/I1`, and `xor/O`. When the user attempts to place `xor` on a flip-flop, Vivado throws an error stating it is illegal to place `xor` at that location. If instead the user tries to place `xor` on compatible BEL such as a LUT6, the placement is made and Vivado offers no feedback.

Pin

The cell's set of **pins** are instanced according to the library cell's set of library pins. Each pin inherits its properties (such as name and direction) and initial values from, and is assigned a pointer back to, its respective library pin. Pins connect directly to nets and comprise the interface to a cell.

Port

A **port** in Vivado represents a top-level IO pin and shares the same characteristics as both a cell and a pin. For instance, a port can be placed on a BEL and can also be directly connected to a net. Cells that are connected to ports are sometimes treated as a special cases during the manual placement of a port, which is discussed in more detail in section 3.4.2.

Net

A **net** represents a logical connection between ports and/or cells in a netlist and may cross hierarchical boundaries. If a net traverses multiple layers of hierarchy, a separate `net` object is created within each level of hierarchy. Nets have a set of properties, including the net's name and type. A net's connections are represented by a set of cell pins and ports.

Like cells, nets are also hierarchical. The `get_nets` command only returns top-level nets, unless the `hierarchical` switch is used. The same naming scheme as cells is used, so children nets may be obtained by filtering the names like so: `get_nets "[get_property NAME $net][get_hierarchy_separator]*"`. Also, like cells, you cannot get the nets of a net (i.e. `get_nets -of $net` is illegal).

Library Objects

As mentioned previously, a number of "library objects" are referenced during the creation of a netlist. Cell instances must reference a library cell, and each of their pins must reference a library pin. Library cells are represented in Vivado by `lib_cell` objects, and are organized into libraries, or `lib` objects. Each library cell has a set of input and/or output library pins, or `lib_pin` objects. For example, every Vivado design references the `UNISIM` library, which contains a set of all the primitive library cells that can be instanced on the device, such as `XOR6`, `IBUF`, and `FDCE`. The `LUT2` library cell has three library pins, two inputs (`LUT2/I0` and `LUT2/I1`) and one output (`LUT2/O`).

As of yet, Vivado provides no commands for creating or removing library objects. It seems the only way to add a user-defined library object to Vivado is through the design's source files (i.e. VHDL, Verilog, or EDIF).

3.3.2 Device Objects

Vivado's physical data structure is closely modeled after an FPGA's actual structure for obvious reasons. This being the case, most of Vivado's physical data structure can be understood from extending the material discussed in section 2.1. That being said, there are some special cases that directly affect CAD tool design that will be discussed more in depth in this section. Nevertheless, all first class physical objects in Vivado are listed in table 3.2.

Table 3.2: Listing of all device objects in Vivado

Physical Component	Vivado Class
BEL	`bel`
BEL Pin	`bel_pin`
Clock Region	`clock_region`
IO Bank	`iobank`
Node	`node`
Package Pin	`package_pin`
Part	`part`
PIP	`pip`
Site	`site`
Site Pin	`site_pin`
Site PIP	`site_pip`
Super Logic Region	`slr`
Tile	`tile`
Wire	`wire`

Objects in Vivado's physical data structure are created in memory when the user loads a part using commands such as `open_checkpoint`, `synth_design`, or `link_design`. This information is read in from a number of encrypted files located in Vivado's install directory (typically under `<install path>/data/parts`). As this data structure represents an actual FPGA's fabric, it is mostly a static representation, though there are some exceptions and special cases. Suffice it to say that, while the objects in Vivado's device data structure can neither be created nor destroyed by the user, their composition can be changed under certain circumstances. The following sections will discuss this among other caveats in greater detail.

Alternate Site Types

Vivado's definition of a site differs slightly from the traditional concept. While a site in Vivado still represents a physical division in component hierarchy on the device, the structure inside of a site in Vivado is not necessarily representative of the circuitry inside an actual site on an FPGA. This is due to the fact that site objects in Vivado may be instanced by multiple site types, making them a dynamic construct while an actual site on an FPGA is obviously static. Every site in Vivado has a property called `ALTERNATE_SITE_TYPES`. When this property is empty, the site has no alternate types and is static. If this property is not

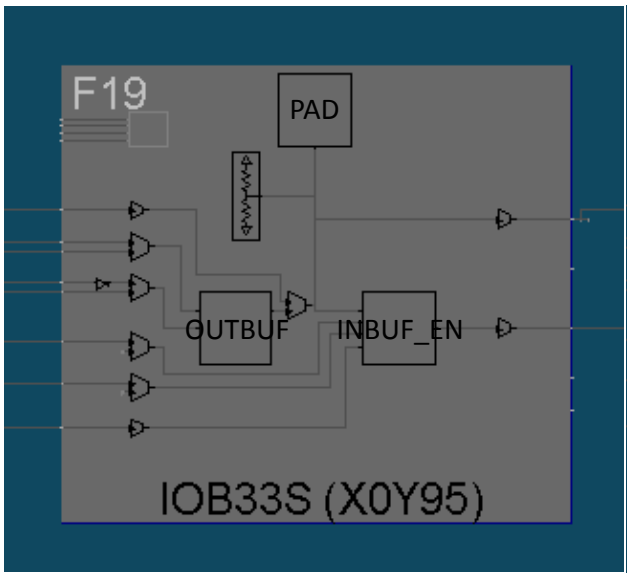

Figure 3.3: Site IOB_X0Y95 instanced by the IOB33S site type.

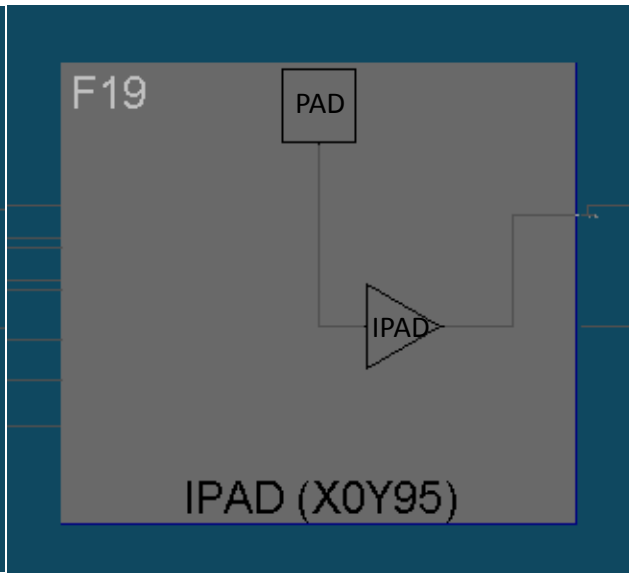

Figure 3.4: Site IOB_X0Y95 instanced by the IPAD site type.

empty, however, it will contain a list of other types the site may instance *besides its original type*. The user may change the type of such a site by setting the `MANUAL_ROUTE` property to whatever site type they wish to instance. When a site's type changes in Vivado, the actual composition of the site changes with it, including BELs and the intrasite routing structure. This is because each site type contains information on how the site on the actual FPGA site should be programmed.

For example, on a xc7k70tfbg484 part, the site `IOB_X0Y95` is initially instanced by the `IOB33S` site type. This defaults to eight BELs in the site, including `PAD`, `INBUF_EN`, and `OUTBUF`, as shown in figure 3.3. Site `IOB_X0Y95` actually has two alternative site types: `IOB33` and `IPAD`. When the site is changed to the type `IPAD`, then it only contains two BELs, `PAD` and `IPAD`, as seen in figure 3.4. The change in BELs necessitates a change in the site's routing structure, as can be seen by comparing figures 3.3 and 3.4.

Alternate BEL Types

Like a site, a BEL's type can be changed. This can be done in one of two ways. First, when a site is instanced by a type, the BELs within that site are instanced by their respective types. Second, there are some cases in which a BEL of one type may change its

type when a cell is placed on it. For example, this happens on the xc7k70tfbg484 part with the BEL within sites that are instanced by the `RAMBFIFO36E1` type, such as `RAMB36_X1Y28`. By default, the BEL inside (there is only one) has the name `RAMB36_X1Y28/RAMBFIFO36E1`, and is of type `RAMBFIFO36E1_RAMBFIFO36E1`. However, if a cell that references the library cell `RAMB36E1` is placed on the BEL named `RAMB36_X1Y28/RAMB36E1`, the BEL inside of the site changes to the BEL that was named, which is of type `RAMB36E1_RAMB36E1`. The parent site's type remains.

This is important to know when implementing a placer, as Vivado will allow the user to place cells on BELs of an incompatible type if those BELs have an alternate BEL type that is compatible. In the example mentioned above, it is technically legal to place the cell of type `RAMB36E1` on a BEL of type `RAMBFIFO36E1_RAMBFIFO36E1`, even though none of the cell's pins are compatible any of the BEL's pins. The only reason that placement is legal is because the BEL has an alternate type that is compatible with the cell.

Unfortunately, there is no property that lists a BEL's alternate types. This must be deduced by exploring the site's alternate types and noting the different types that a BEL may take on during this process. This can be tricky as there is no definitive way to know which BELs in a site type instance refer to the same actual BEL on an FPGA. Thus, this may become a tedious and error-prone process.

Intrasite Routing

While Vivado's Tcl interface provides the user with a wealth of information regarding its device database, there is still a good deal of information that is not visible. Unfortunately, this includes the routing within a site, or in other words, the intrasite routing structure. This information is vital for a CAD tool framework, as it is a necessity for any routing tool. Without intrasite routing information, it is not possible to determine how BELs connect to either the routing structure outside of a site or to each other within a site.

While the intrasite routing structure is not available through Vivado's Tcl interface, interestingly it is possible to view through the GUI. Vivado's device browser gives the user the option to display routing resources, and this includes intrasite routing. At the very least,

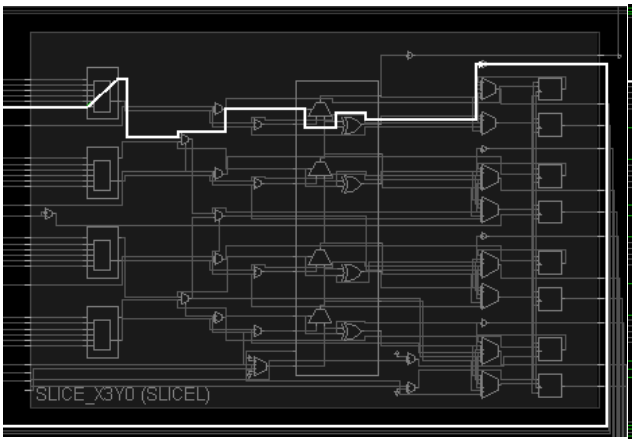

Figure 3.5: A site route-through

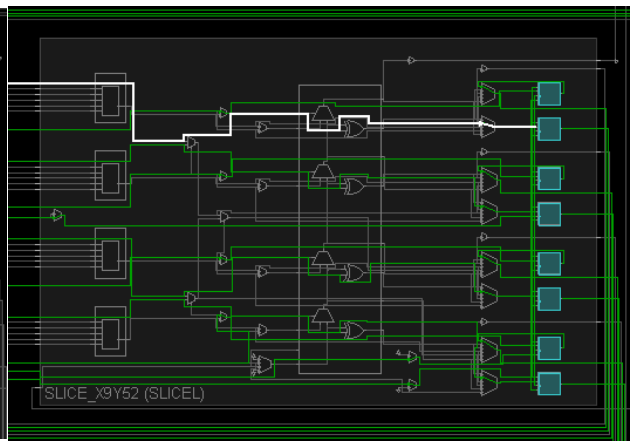

Figure 3.6: A BEL route-through

the user may obtain this information through visual inspection of the GUI and then hard code it into their router.

Route-throughs

Xilinx allows the use of what it calls "route-throughs" to supplement the routing of a design on an FPGA. Simply put, a route-through is when a logical resource is utilized as a routing resource. In Vivado, there are two types of route-throughs which may be referred to as site route-throughs and BEL route-throughs.

In a site route-through, a net is passed in through one of a site's input pins, routed through the site's internal routing structure, and passed out through one of the site's output pins. Once a single site route-through has been made, the entire site is classified as a routing resource, making it illegal to place any cells on it. Site route-throughs are included as part of the device's routing structure, and are represented by PIP objects. To distinguish them from regular PIPs, their `IS_PSEUDO` property is set to 1. A `SLICEL` site is a common example of a site with route-throughs. Calling the `get_pips -of $site -filter IS_PSEUDO` command on the `SLICE_X3Y0` site on the `xc7a35tcpg236` part returns 50 such site route-through PIPs. Figure 3.5 shows a net (depicted by a heavy white line) that is routed through `SLICE_X3Y0`'s `CLBLM_R_X3Y0/CLBLM_R.CLBLM_L_D1->>CLBLM_L_D` route-through PIP.

A BEL route-through allows a net to be passed in through one of a BEL's input pins and passed out through one of its output pins. This is exclusively a function of LUTs in

Xilinx devices, and is essentially the equivalent of instancing the LUT with an equation that assigns the output to the value of the input. BEL route-throughs exists at the sub-site level, and are considered a part of the intrasite routing network. They are represented by site PIP objects, one for each input on the LUT. It should be noted that a site route-through may use a BEL route-through as a part of its route-through (see figure 3.5), though this is not always the case. This is because a site route-though is allowed to use any available sub-site routing resources to make the route-through, including BEL route-throughs. In figure 3.6, the `D6LUT` is acting as a BEL route-through for the net depicted by the heavy white line. This net is routed through the `SLICE_X9Y52/D6LUT:A6` site PIP to source the `DFF` flip-flop's `D` pin on the right side of the site.

BEL route-throughs present a number of problems for CAD tool designers. First, while a site PIP is created for each possible route-through, there are no properties that distinguish them from normal site PIPs (unlike a site route-through). Additionally, normal site PIPs have the `IS_USED` property asserted when they are traversed by a net, but BEL route-throughs do not. This is a problem because it becomes very difficult to know whether or not a BEL route-through is actually being used, requiring the user to check for a myriad of special cases. Finally, Vivado does not provide any way to constrain BEL route-throughs. Vivado automatically chooses which BEL pins are used according to some predefined algorithm, which unfortunately has the tendency to break under certain circumstances. As BEL route-throughs cannot be avoided in some cases, a means of handling or replacing them must be implemented by any candidate CAD tool framework.

Routing Directionality

Routing resources can be difficult to navigate in Vivado, due to a distinct lack of directionality in the relationships between Tcl objects. This is primarily due to the organization of related Tcl objects into sets. For example, if the user would like to get a list of the PIPs that connect to a node, they would use the command `get_pips -of $node`. Unfortunately, this list provides no information about the directionality of the PIPs returned — for all the user knows, these PIPs could all source the node or all be sourced by the node. To help remedy this situation, Vivado provides the switches `-uphill` and `-downhill` for the

commands `get_nodes`, `get_pips`, and `get_wires`. These switches provide the directionality information regarding routing resources that a CAD tool needs to successfully navigate Vivado's routing network.

3.4 Database Manipulation

Perhaps of the most interest to a CAD tool designer is the ability to manipulate Vivado's object databases for the purpose of writing a technology mapper, placer, or router. This section discusses the various methods available in Vivado for manipulating a design's netlist, placement, and routing information using a number of high- and low-level commands from Vivado's Tcl interface.

To help explain these manipulation methods, an example circuit for a full adder will be constructed, placed, and routed using the commands described in this section. The part used is the `xc7a35tcpg236`, and any VHDL, EDIF, or XDC files are listed in appendix A.

3.4.1 Netlist

As mentioned previously, a Vivado design is represented across several netlist objects in memory and can be manipulated through a set a Tcl commands. The netlist structure can be populated by a number of methods, each of which has its own advantages and disadvantages. In any case, Vivado allows a relatively wide variety of means by which a user can create a design in memory, whether it be through synthesis of an RTL language, by importing the netlist from an EDIF file, or even by manual creation through Tcl commands. This section aims to provide some more insight on each of these methods.

Synthesis

The most obvious way of populating a netlist in Vivado is through its proprietary synthesis EDA tool. This is done by dumping any number of HDL and constraint files into Vivado's memory and then calling the `synth_design` command, as shown in listing 3.1. From there, Vivado's synthesis tool will map the design into a netlist of primitive cells and populate it's data structures accordingly. While this is the most efficient way to populate Vivado's design data structure from an RTL representation, it does not provide much control

over how the netlist is synthesized, which is to be expected from a general purpose EDA tool. More information on Vivado's synthesis tool can be found in [19].

Listing 3.1: Populating the full adder netlist using Vivado's synthesis tool

```
1 read_vhdl -library fullAdder ./fullAdder.vhd
2 read_xdc ./fullAdder.xdc
3 synth_design -part xc7a35tcpg236
```

EDIF File

The EDIF format is capable of completely describing a Vivado netlist, and is the best method for populating entire designs in Vivado. Using the `read_edif` command, an EDIF netlist is read directly into Vivado's file database. From that point, the file can be linked into a design by calling the `link_design` command, as shown in listing 3.2. This creates a completely new design, and elements from any previous designs are not included. This method is able to populate the design database quickly, and is significantly faster at populating an entire netlist from scratch than any other method. However, it does require that the EDIF file be written with an intimate knowledge of the device being targeted and the primitive cells that are available. Bearing that in mind, Vivado also provides the `write_edif` command, which outputs an EDIF representation of the design currently in memory. All netlist object properties are correctly preserved. More information on the `read_edif` and `write_edif` commands can be found in [18].

Listing 3.2: Populating the full adder netlist using `read_edif`

```
1 read_edif ./fullAdder.edf
2 read_xdc ./fullAdder.xdc
3 link_design -part xc7a35tcpg236
```

Manual Netlist Creation

The final and most primitive way of populating a netlist in Vivado is through a set of netlist manipulation commands. Most design objects in Vivado may be created or destroyed using Tcl commands prefixed by `create_` and `remove_`, respectively. For example, a cell can be created using the `create_cell` command, or destroyed using the `remove_cell` command. Connecting the netlist together can be done using the `connect_net` command, or disconnected using `disconnect_net`. Listing 3.3 shows how the netlist for a full adder can be created using these commands. While this method offers the most control and flexibility over how the netlist is populated in memory, it is also the most time consuming. Even when a netlist's creation is scripted in this manner, the Tcl commands being used are slower than if the netlist were populated using the `read_edif` command.

Listing 3.3: Manually populate the full adder netlist using Tcl commands

```
# Load the part
link_design -part xc7a35tcpg236

# NETLIST CREATION
# ================
# Create cells
set and1_lut [create_cell -reference AND2 and1_lut]
set and2_lut [create_cell -reference AND2 and2_lut]
set or_lut [create_cell -reference OR2 or_lut]
set xor1_lut [create_cell -reference XOR2 xor1_lut]
set xor2_lut [create_cell -reference XOR2 xor2_lut]

# Create ports
set a_port [create_port -direction IN a_port]
set b_port [create_port -direction IN b_port]
set cin_port [create_port -direction IN cin_port]
set cout_port [create_port -direction OUT cout_port]
set sum_port [create_port -direction OUT sum_port]

# Create IO buffers
set a_ibuf [create_cell -reference IBUF a_ibuf]
set b_ibuf [create_cell -reference IBUF b_ibuf]
set cin_ibuf [create_cell -reference IBUF cin_ibuf]
set cout_obuf [create_cell -reference OBUF cout_obuf]
set sum_obuf [create_cell -reference OBUF sum_obuf]

# Create the nets
set a_net [create_net a_net]
set a_ibuf_net [create_net a_ibuf_net]
```

```
30 set b_net [create_net b_net]
31 set b_ibuf_net [create_net b_ibuf_net]
32 set cin_net [create_net cin_net]
33 set cin_ibuf_net [create_net cin_ibuf_net]
34 set cout_obuf_net [create_net cout_obuf_net]
35 set sum_obuf_net [create_net sum_obuf_net]
36 set and1_net [create_net and1_net]
37 set and2_net [create_net and2_net]
38 set or_net [create_net or_net]
39 set xor1_net [create_net xor1_net]
40 set xor2_net [create_net xor2_net]
41
42 # Connect the nets
43 connect_net -net $a_net -objects "$a_port [get_pins a_ibuf/I]"
44 connect_net -net $a_ibuf_net -objects "[get_pins a_ibuf/O] [get_pins
     xor1_lut/I0] [get_pins and1_lut/I0]"
45 connect_net -net $b_net -objects "$b_port [get_pins b_ibuf/I]"
46 connect_net -net $b_ibuf_net -objects "[get_pins b_ibuf/O] [get_pins
     xor1_lut/I1] [get_pins and1_lut/I1]"
47 connect_net -net $cin_net -objects "$cin_port [get_pins cin_ibuf/I]"
48 connect_net -net $cin_ibuf_net -objects "[get_pins cin_ibuf/O] [get_pins
     xor2_lut/I1] [get_pins and2_lut/I1]"
49 connect_net -net $cout_obuf_net -objects "[get_pins cout_obuf/O] $cout_port"
50 connect_net -net $sum_obuf_net -objects "[get_pins sum_obuf/O] $sum_port"
51 connect_net -net $and1_net -objects "[get_pins and1_lut/O] [get_pins or_lut/
     I1]"
52 connect_net -net $and2_net -objects "[get_pins and2_lut/O] [get_pins or_lut/
     I0]"
53 connect_net -net $or_net -objects "[get_pins or_lut/O] [get_pins cout_obuf/I
     ]"
54 connect_net -net $xor1_net -objects "[get_pins xor1_lut/O] [get_pins
     xor2_lut/I0] [get_pins and2_lut/I0]"
55 connect_net -net $xor2_net -objects "[get_pins xor2_lut/O] [get_pins
     sum_obuf/I]"
56
57 # Set an IOSTANDARD for inputs and outputs (avoids a DRC error)
58 set_property IOSTANDARD LVCMOS18 [all_inputs]
59 set_property IOSTANDARD LVCMOS18 [all_outputs]
```

3.4.2 Placement

Placement is the mapping of logical cells to physical components on a device. The act of mapping a cell to a BEL is called "placing" (i.e. cells are placed on BELs). Typically, an FPGA manufacturer will provide the consumer with a CAD tool called a placer. Given a netlist and an FPGA, a placer maps logical cells to physical BELs on the FPGA. Vivado

provides commands both for accessing its placer and for placing individual cells, allowing the user to make either large or minute changes to the placement map of a design.

Place the Entire Design Using Vivado's Placer

For placing an entire design, the `place_design` command calls Xilinx's proprietary placer tool. `place_design` offers a number of options so that the user can adjust the placer's settings according to their own requirements. For instance, the `-directive` option allows the user to choose from a number of different placement strategies. One particularly useful strategy is "Quick," in which the placer performs the minimum required placement for a legal design, yielding the fastest possible runtime. The `-unplace` switch unplaces all non-locked cells in a design. To place the full adder design using Vivado's built-in placer, one may simply call the `place_design` command. Alternatively, Vivado also has the `place_ports` command which uses Vivado's placer tool to place just the top-level ports in a design.

Manual Cell Placement

Placing individual cells may be done in one of two ways. The first, and perhaps most primitive way, involves setting the properties of individual cells. Specifically, each cell has two writable properties, `BEL` and `LOC`, that will constrain the cell to a specific location on the device. The `LOC` property will *always* display the name of the current site the cell is placed on. When `LOC` is empty, the cell is not placed. The `BEL` property constrains the cell to a BEL within the site it is either already placed in, or will be placed in. The `BEL` property can be set to a local BEL name relative to the parent site's type, such as `SLICEL.D6LUT`. Additionally, the `BEL` property may be set to any compatible BEL name or the empty string. If the given string is the name of an incompatible BEL or not a BEL name at all, it will be ignored. If the `BEL` property's value is given the empty string, the `LOC` property may be set to the name of any site that has at least one available compatible BEL on it, otherwise an error is thrown. Vivado will automatically choose one of the compatible BELs in the site and set the `BEL` property to its local name. If the `BEL` property's value is non-empty, the cell will be constrained to that BEL inside of the site as long as it isn't already being used. If it is being used or if the given site has no such BEL, Vivado throws an error. To unplace

a cell, one may simply set these properties to the empty string (recommend `LOC` first, then `BEL`). Listing 3.4 places the full adder circuit by setting the `LOC` and `BEL` properties.

Listing 3.4: Manually place each cell by setting its `LOC` and `BEL` properties

```
# PLACEMENT
# =========
# Place the cells
set_property LOC SLICE_X0Y1 $and1_lut
set_property BEL SLICEL.A6LUT $and1_lut
set_property LOC SLICE_X0Y1 $and2_lut
set_property BEL SLICEL.B6LUT $and2_lut
set_property LOC SLICE_X0Y1 $or_lut
set_property BEL SLICEL.C6LUT $or_lut
set_property LOC SLICE_X1Y1 $xor1_lut
set_property BEL SLICEL.A6LUT $xor1_lut
set_property LOC SLICE_X0Y1 $xor2_lut
set_property BEL SLICEL.D6LUT $xor2_lut
```

The second command for placing individual cells is the `place_cell` command. This command will take as input an interleaved list of the names of cells to be placed and the corresponding names of the BELs they are to be placed on. The behavior of this command is identical to the act of setting the `BEL` and `LOC` properties and suggests that `place_cell` is merely a wrapper command. The antithesis of this command is `unplace_cell`, which, given a list of cell names, will unplace all of them.

Manual Pin Placement

When a cell is placed, typically there is a one-to-one mapping between cell pins and BEL pins. However, in some cases, a cell pin is compatible with more than one BEL pin. In such situations, Vivado arbitrarily assigns pins to BEL pins. This can cause problems for the CAD tool programmer when he or she needs to ensure that a specific BEL pin is being used. The cell's `LOCK_PINS` property allows the user to specify which cell pins are placed on which BEL pins for cells of type `LUT`, `INV`, and `BUF`. Listing 3.5 shows how to constrain the pins on the `LUT`-based cells in the full adder design.

Listing 3.5: Manually place each pin by setting its `LOCK_PINS` property

```
# Set the input pins for LUTs
set_property LOCK_PINS {I0:A3 I1:A5} $and1_lut
set_property LOCK_PINS {I0:A3 I1:A6} $and2_lut
set_property LOCK_PINS {I0:A6 I1:A1} $or_lut
set_property LOCK_PINS {I0:A3 I1:A5} $xor1_lut
set_property LOCK_PINS {I0:A2 I1:A6} $xor2_lut
```

Manual Port Placement

Similar to cells, ports have a `LOC` property that the user may set. Unlike cells, ports do not have a `BEL` property, but do have a property called `PACKAGE_PIN` that can be set to assign the port to a specific package pin on the device. A package pin is a BEL of type `PAD` on an IO site and represents a physical pin coming off of the device. Setting the `PACKAGE_PIN` property also sets its `LOC` property to the site the package pin is located in. Curiously, if the port is connected to an IO buffer, Vivado will automatically place the buffer cell in the same site where the port is placed. Listing 3.6 displays the Tcl commands for manually placing the ports in the full adder design. It may be noted that listing 3.4 does not place any of the IO buffer cells — that is because they are placed with the ports in this step.

Listing 3.6: Manually place each port by setting its `LOC` property

```
# Place the ports
set_property LOC IOB_X0Y0 $a_port
set_property LOC IOB_X0Y1 $b_port
set_property LOC IOB_X0Y2 $cin_port
set_property LOC IOB_X0Y3 $cout_port
set_property LOC IOB_X0Y4 $sum_port
```

The Placed Flag

Unfortunately, Vivado will not recognize a design as "placed" until after `place_design` has been called at least once, even if all cells, pins, and ports in the design have been placed. In the context of a custom placement tool, this means that even if every cell in the design has been manually placed, the user must still call Vivado's placer to throw the placed flag

before calling any of the EDA tools that occur later in Vivado's design flow (such as the router or bitstream generation).

3.4.3 Routing

Routing a design in Vivado involves assigning a sequence of nodes to a net. When a bitstream is written, the PIPs between subsequent nodes are turned on to complete a connection between the two nodes. Similar to placements, there are two ways to route a design in Vivado: by using Vivado's EDA routing tool or by manually assigning routes to every net in the design.

Route the Entire Design Using Vivado's Router

The `route_design` command executes Vivado's proprietary router on the design in memory. The router has a number of options and switches that allow the user to tweak its operation to their liking, all of which are outlined in [18]. To route the full adder example using Vivado's router, one need only call `route_design`.

Manually Route an Individual Net

Among a net's properties is the `ROUTE` property. This property is read/write and contains what Xilinx refers to as a "directed routing" string [13]. A directed routing string describes the route of the net in terms of the names of the nodes it traverses from its source to all of its sinks. This is different from XDL, which described a net's route in terms of PIPs, the components *between* nodes. When the value of the `ROUTE` property is set, the net is constrained to the route defined by the directed routing string. If it is set to the empty string, the net is unrouted. Nets also have a `FIXED_ROUTE` property, which has the same basic function as the `ROUTE` property, but also asserts the net's `IS_FIXED` property when set, preventing any of the Vivado tools from altering the route.

Directed routing strings use a very simple grammar for describing a route. This grammar borrows from Tcl's list structure, and can be easily parsed by Tcl as a list of nested lists. Specifically, a directed routing string is a nested list of node names which can either be absolute (`CLBLL_L_X2Y0/CLBLL_LL_DX`) or relative (`CLBLL_LL_DX`). Relative directed routing

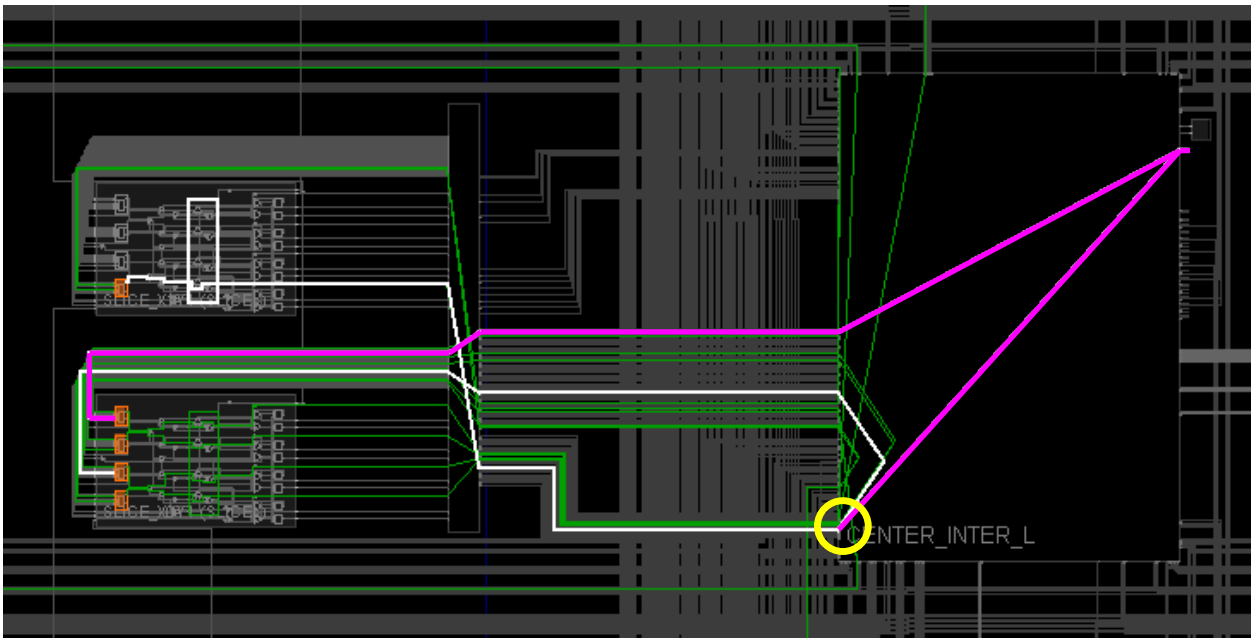

Figure 3.7: A depiction of a branched route

strings are particularly useful when importing routing for relatively placed macros. A pair of curly brackets define a branch in the route (or a nested list in Tcl). For example, figure 3.7 shows a switchbox and the net that is routed through it. The net is highlighted with heavy white and magenta lines. This net branches at the PIP junction marked by the yellow circle. This net's directed routing string is constructed like so: { { `CLBLL_L_A CLBLL_LOGIC_OUTS8` { `NL1BEG_N3 IMUX_L45 CLBLL_LL_D2` } `IMUX_L17 CLBLL_LL_B3` } }. Notice the nested list beginning with node `NL1BEG_N3` and ending with node `CLBLL_LL_D2`. This list represents a branch in the route and is shown in magenta in figure 3.7. The branch begins right after node `CLBLL_LOGIC_OUTS8` when the net enters the switchbox, and terminates at the sink sourced by node `CLBLL_LL_D2` (the `D6LUT`'s `A2` pin).

Listing 3.7 shows the code to completely route the full adder example circuit. The `ROUTE` property of each net has been set to the relative node names the net is to take. Special care must be taken to ensure that the pins a net connects to line up with the nodes the net's route is constrained to. It is good practice to set the `LOCK_PINS` property on any cells that instance LUTs (see section 3.4.2), as Vivado may arbitrarily shuffle their input pins around without notice.

Listing 3.7: Manually route each net by setting its `ROUTE` property

```
# ROUTING
# =======
# Route the nets
set_property ROUTE { { IOB_IBUF0 LIOI_I0 LIOI_ILOGIC0_D IOI_ILOGIC0_O
    IOI_LOGIC_OUTS18_0 INT_INTERFACE_LOGIC_OUTS_L18 EE2BEG0 NR1BEG0 {
    IMUX_L1 CLBLL_LL_A3 } IMUX_L0 CLBLL_L_A3 } } $a_ibuf_net
set_property ROUTE { { IOB_IBUF1 LIOI_I1 LIOI_ILOGIC1_D IOI_ILOGIC1_O
    IOI_LOGIC_OUTS18_0 INT_INTERFACE_LOGIC_OUTS_L18 EE2BEG0 { IMUX_L8
    CLBLL_LL_A5 } IMUX_L9 CLBLL_L_A5 } } $b_ibuf_net
set_property ROUTE { { IOB_IBUF0 LIOI_I0 LIOI_ILOGIC0_D IOI_ILOGIC0_O
    IOI_LOGIC_OUTS18_1 INT_INTERFACE_LOGIC_OUTS_L18 EL1BEG_N3 EL1BEG2 {
    IMUX_L43 CLBLL_LL_D6 } IMUX_L12 CLBLL_LL_B6 } } $cin_ibuf_net
set_property ROUTE { { CLBLL_L_A CLBLL_LOGIC_OUTS8 { NL1BEG_N3 IMUX_L45
    CLBLL_LL_D2 } IMUX_L17 CLBLL_LL_B3 } } $xor1_net
set_property ROUTE { { CLBLL_LL_D CLBLL_LOGIC_OUTS15 NW6BEG3 SR1BEG3
    SR1BEG_S0 IMUX_L34 IOI_OLOGIC0_D1 LIOI_OLOGIC0_OQ LIOI_O0 } } $xor2_net
set_property ROUTE { { CLBLL_LL_A CLBLL_LOGIC_OUTS12 IMUX_L32 CLBLL_LL_C1 }
    } $and1_net
set_property ROUTE { { CLBLL_LL_B CLBLL_LOGIC_OUTS13 IMUX_L35 CLBLL_LL_C6 }
    } $and2_net
set_property ROUTE { { CLBLL_LL_C CLBLL_LOGIC_OUTS14 WW2BEG2 NL1BEG2 NL1BEG1
    IMUX_L34 IOI_OLOGIC1_D1 LIOI_OLOGIC1_OQ LIOI_O1 } } $or_net
```

Manually Route a Site

The previous section discusses how to constrain routing outside of a site, a function that was available in XDL. Also available in XDL was the ability to constrain the routing within a site by turning on various routing muxes. Vivado also allows the user to manually route a site using a method that is completely undocumented by Xilinx.

All sites on a device have two properties called `MANUAL_ROUTING` and `SITE_PIPS`, both are strings and allow read/write access. Section 3.3.2 discusses the `MANUAL_ROUTING` property and points out that it is used to change the type of a site. In this context, the `MANUAL_ROUTING` property is used to turn on or off a site's manual routing feature. To manually route a site, the `MANUAL_ROUTING` property must be set to the site type you wish to route. For example, to manually route a `SLICEL` site, the user must first call the command `set_property MANUAL_ROUTING SLICEL $site`.

Once the `MANUAL_ROUTING` property has been set, the user may then begin routing the site. This is done by "turning on" a set of site PIPs within the site that connect the site

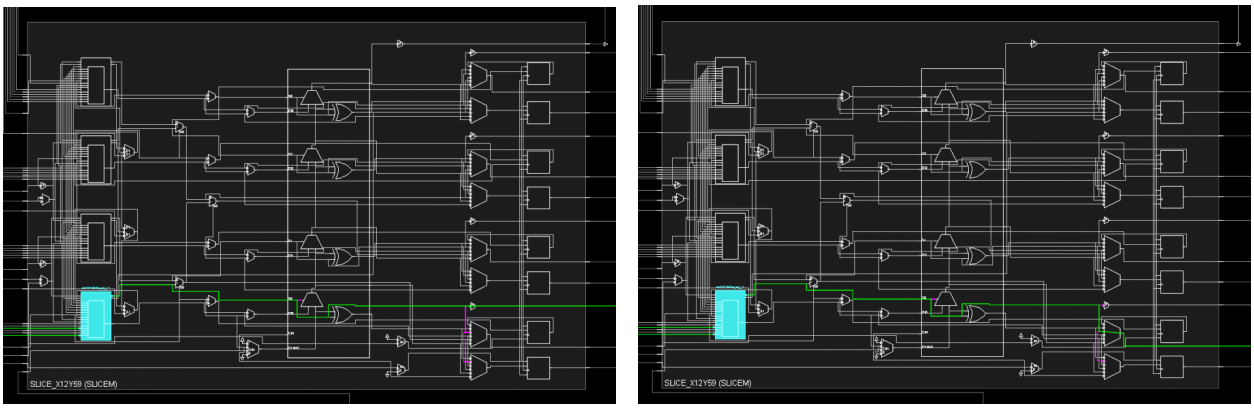

Figure 3.8: Results of manually routing a site. On the left is the original route performed by Vivado, on the right the route has been manually redirected.

wires the route will take. To turn on a set of site PIPs, a space-separated list of their names must be written to the site's `SITE_PIPS` property. For example, figure 3.8 shows two different routing configurations for the same site. The first one (on the left) is the default routing assigned by Vivado. The second one is the result of calling the commands in listing 3.8.

Listing 3.8: Manually route the site using the `MANUAL_ROUTING` and `SITE_PIPS` properties

```
1 set site [get_sites SLICE_X12Y59]
2 set_property MANUAL_ROUTING SLICEM $site
3 set_property SITE_PIPS {SLICE_X12Y59/AOUTMUX:O6} $site
```

Global Logic Nets

In a Vivado netlist, global logic `GND` and `VCC` nets are represented by two nets named `<const0>` and `<const1>` respectively. These nets are a unique case in Vivado, and require special handling. This is largely due to the fact that *all* global logic signals are represented by just two nets. Thus, each these nets are very large in size and can span thousands of nodes. While global logic nets can be routed using directed routing strings, their sheer size may cause Vivado to grind to a halt. As the CAD tool designer is rarely (if ever) concerned with the routing of global logic nets, the task of routing them is better suited for Vivado's router. This can be done using the command `route_design -physical_nets`.

3.5 Conclusion

The beginning of this chapter stated that its ultimate goal was to discover whether or not Vivado provided the basic functionality to support a custom CAD tool framework similar to RapidSmith or Torc. It is the opinion of this thesis that Vivado is able to support such a framework, based on these qualifying features:

- Vivado provides complete access to and control over its design database through a set of Tcl commands.
- Vivado provides access to a majority of its device database.
- Vivado has methods for constraining placement and routing.
- Vivado has methods for exporting and importing a netlist to and from an EDIF file.
- Vivado is able to apply constraints to a design through the importation of XDC files.

As pointed out in this chapter, Vivado withholds some information which is vital to the implementation of some CAD tools, such as the routing within a site. However, most, if not all, of this information can be discovered through alternative means, such as visual inspection of the GUI or brute force algorithms. The following chapters in this work will seek to show how this information can be supplemented by outside sources to provide the required functionality within the Vivado design environment.

Chapter 4

TincrCAD: A Tcl-Based CAD Tool Framework for Vivado

Chapter 3 gives a thorough explanation of Vivado, and offers evidence that Vivado is able to support a custom CAD tool framework, provided that certain elements are supplemented from external sources. This chapter proposes one of two possible methods for implementing a CAD tool framework for Vivado, a library of Tcl commands called TincrCAD.

4.1 Motivation

The Vivado Design Suite provides a richly featured command set for low-level circuit manipulations. Chapter 3 provides a detailed description of the capabilities of the Vivado Tcl interface from a CAD tool perspective. Based on the functionality described there, one could conceivably implement a number of FPGA CAD tools within the Vivado Tcl environment. There are, however, some problems with Vivado's Tcl interpreter as it now stands.

First, as section 3.2.2 indicates, the Vivado Tcl command set is so primitive that trying to implement higher-level algorithms can be a tedious and error-prone process that often leads to rather verbose and convoluted code. This is because most of the functionality provided by Vivado for querying and manipulating its data structures consists solely of a set of getter commands for obtaining pointers to various objects, and a set of setter commands for manipulating the properties of these objects. An example is the command for unplacing all of the primitive children cells of a hierarchical cell (stored in the variable `cell`):

```
1 unplace_cell [get_cells -hierarchical -filter "(PRIMITIVE_LEVEL == LEAF ||
    PRIMITIVE_LEVEL == INTERNAL) && NAME =~ [get_property NAME $cell][
    get_hierarchy_separator]*"]
```

Additionally, while Vivado does provide adequate documentation for its set of Tcl commands, it neglects to document the object properties that these commands operate on.

Chapter 3 is riddled with explanations of various object properties that are otherwise undocumented, and the only way of understanding their effects is through exhaustive trial and error. Examples include a cell's `NAME`, `LOC`, and `BEL` properties, or a site's `ALTERNATE_SITE_TYPES`, `MANUAL_ROUTING`, and `SITE_PIPS` properties.

Furthermore, Vivado has omitted large amounts of information from its Tcl interface that is vital to the implementation of a number of CAD tools within its interpreter. This includes cell to BEL compatibility mappings and intrasite routing information, without which writing a placer or router in Vivado is all but impossible.

These reasons, among others, have led to the development of **Tincr** (pronounced "tinker"), a suite of Tcl libraries that may be installed alongside Vivado's existing Tcl shell. Tincr augments Vivado's native Tcl interface with a set of high-level commands that simplify a variety of functions common in the implementation of CAD tools, allowing the user to operate at a higher level of abstraction. By wrapping sequences of complex low-level Tcl commands into single cohesive functions, Tincr is able to reduce both design and verification time for researchers and designers. On average, Tincr saves the user 50 lines of code per command. In addition, Tincr adds a significant amount of information that is otherwise unavailable in Vivado, allowing the implementation of a greater variety of CAD tools.

The Tincr suite is split into two libraries, called **TincrCAD** and **TincrIO**. TincrCAD offers a complete Tcl-based CAD tool framework for Vivado's Tcl interpreter and will be the topic of discussion in this chapter. TincrIO is a library of commands that together provide an interface into Vivado similar to XDL, and is discussed in Chapter 5. For the sake of cohesion, aspects of TincrIO's organization and performance will be mentioned in this chapter in conjunction with similar topics from TincrCAD.

4.2 An Overview of Tincr

A number of design decisions were made while implementing Tincr to make it a feasible alternative to previous frameworks such as RapidSmith. Typically, the primary metrics used to gauge a CAD tool framework's quality include performance and ease-of-use. Like RapidSmith, Tincr's primary priority is performance with its secondary priority being organization. This section discusses the mechanisms used to achieve gains in performance

and efficiency, as well as Tincr's organizational structure, which was intended to make its API intuitive and simple to use.

4.2.1 Performance Mechanisms

A number of optimizations were applied to the Tincr suite to obtain gains in performance. In this context, performance refers to the amount of time it takes a command to execute. A more performant command executes in a shorter amount of time. Often the gains Tincr achieves are realized at the expense of memory, but as low memory usage is not considered a priority in Tincr, such expenses were disregarded. This subsection will discuss the key optimizations that were made and offer measurements of performance.

Caching

The greatest performance improvements are obtained through a number of caches that are kept throughout the framework. Tincr operates on a large amount of data in order to provide the additional functionality that it does. For example, to provide the ability to get the BEL of a BEL pin, Tincr stores a set of BEL pin to BEL mappings in an associative array. This is opposed to parsing the name of the BEL from the BEL pin's name, and then using `get_bels` to obtain a pointer to the BEL, all of which is slower by about 0.5 milliseconds.

For the most part, caching occurs with device objects. This is because the device database is normally the most traversed by CAD tools, since an algorithm is often searching for available resources on the device. Caching these resources allows look-ups to occur quickly during the most critical portions of a CAD tool's algorithm. Most notably, are the BEL and site classes, which employ a number of caches that keep track of their various configurations. Sites in Vivado are dynamic and can be instanced by multiple site types, each of which is cached in a variable called `site2sitetypes`. Another cache is kept that lists every site/site-type pair on a device, called `sitesitetypepairs`. This particular cache allows the user to quickly iterate through all the site/type combinations on a device, and is what Tincr's `sites foreach` command uses in its main looping construct.

Since BELs may be added or removed from a device when a site's type is changed, Tincr caches all possible BELs also. This is incredibly useful when trying to obtain a list of BELs that are compatible with a particular cell type. It also saves a lot of time. Rather than iterate through all sites and their respective site type configurations to create a list of all possible BELs of a specific type, the `beltype2bels` cache maps BEL types to a list of possible BELs of that type. Retrieving this information off of a warm cache takes only 0.5 milliseconds, while obtaining it through Vivado commands takes upwards of 10 seconds.

The only issue with this system of caching is that it currently takes several minutes to "warm up" and every time Vivado is restarted, Tincr has to re-generate all of its information from scratch. This is, however, the worst case scenario as no provisions have yet been taken to save the cached information between successive runs of Vivado. Thus, every time Vivado starts, the caches are re-generated from scratch, which takes approximately 5 minutes for a moderate-sized part. Future work could include methods that write the cached information to disk, possibly using the serialization packages available in TCLLIB.

Parallelization

The other method Tincr uses to achieve performance gains is parallelization of large tasks. As modern machines are able to handle more and more concurrent processes, Tincr has been outfitted with a set of multi-process commands to allow the user to utilize a greater amount of the resources available on the host machine. One such function is the `spawn_vivado_run`, which accepts as input a Tcl script and increments a process counter semaphore. Once the Vivado run completes, the process counter is decremented. Using this command, the user can execute concurrent runs of Vivado, each running a different set of commands.

This capability is best suited for tasks that take an inordinate amount of time and/or use a severe amount of memory (due to one of Vivado's memory leaks). More importantly, the task must lend itself to concurrency. A good example is the `write_xdlrc` function from TincrIO's `device` package. As this particular function is simply reading the device database and dumping information to file, the script can easily be parallelized on a tile by tile basis. `write_xdlrc` does this by having each spawned process dump the XDLRC information for

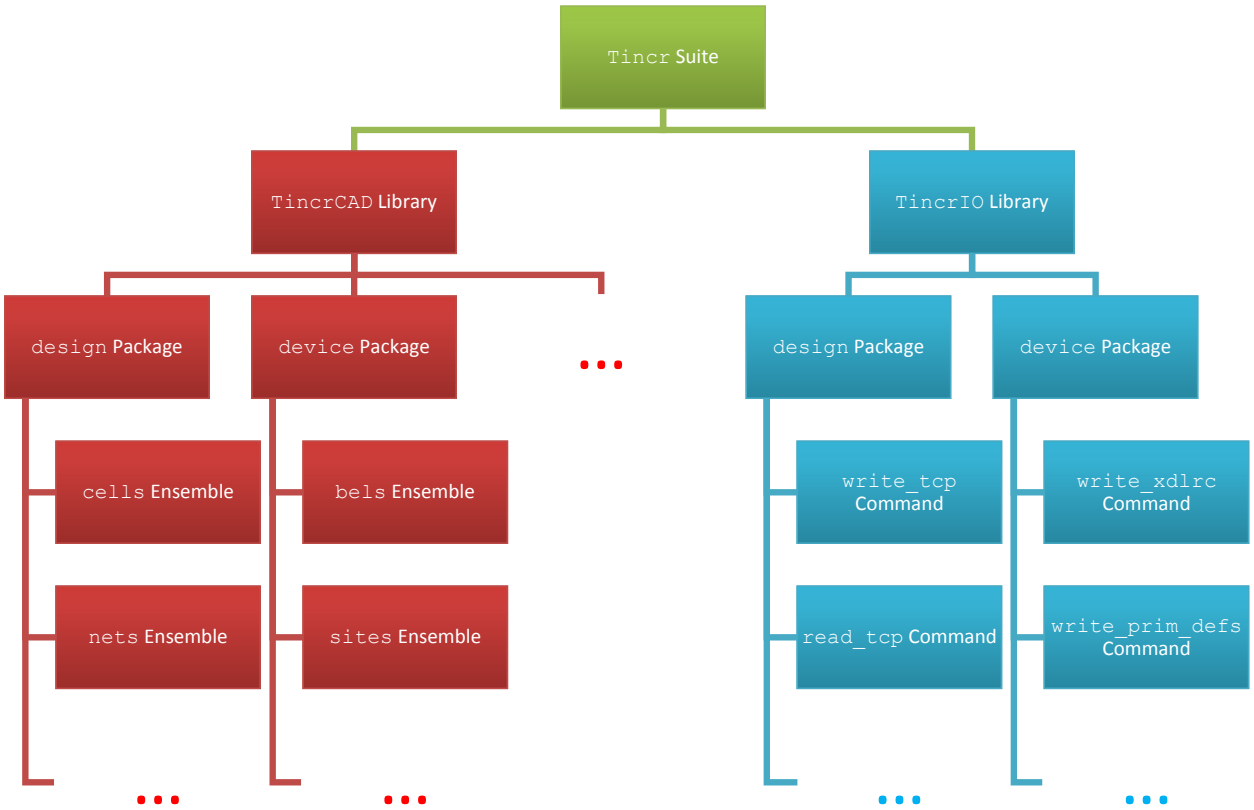

Figure 4.1: A graph describing Tincr's organizational hierarchy

some number of tiles out to a temporary file, and then concatenating them all together at the end. To control the number of processes spawned, `write_xdlrc` accepts as input a maximum number of processes the user wishes to be running at once. Using the process counter semaphore mentioned above, `write_xdlrc` spawns new processes as old ones finish, all while making sure that the number of concurrent processes never exceeds the maximum amount specified by the user. Tests have shown that using this method, the `write_xdlrc` function achieves a speed up equivalent to the maximum number of processes specified by the user.

4.2.2 Organizational Structure

At the time of this writing, Tincr consists of more than 170 commands spread across 9,000 lines of code written in Tcl. Tincr uses a number of organizational techniques to provide the user with a simple and intuitive API. Each of the methods used to organize

Tincr are outlined in Tcl's official specification and are available in all versions of Tcl 8.5 and later. A depiction of Tincr's fundamental hierarchy is shown in figure 4.1. This section discusses the organization of Tincr and explains why certain design decisions were made.

Namespaces

All of Tincr's commands exist within the `::tincr::` namespace, which makes it easy to call commands from all of Tincr's libraries while still avoiding any symbol conflicts with Vivado's global namespace. All of the commands in Tincr's namespace can be imported into the global namespace using the Tcl command `namespace import ::tincr::*`. As of Vivado 2014.2 and Tincr 2.0, there are no symbol conflicts between the two command sets.

Ensembles

New to Tcl 8.5 is the namespace ensemble construct. An ensemble is a set of subcommands that are all organized under the same parent command, and takes the form of `<command> <subcommand> [args]`. An example of an ensemble is the `dict` command, for manipulating dictionary structures in Tcl. The `dict` ensemble has many subcommands, one of which is the `dict get <dictionary> <key>` command, which returns the value in `dictionary` associated with `key`. Ensembles are useful for organizing a group of related functions.

Vivado's command set adheres well to the ensemble structure, because commands often operate on one or more objects of the same class. Yet for some reason, ensembles are not used to organize commands in Vivado. As a result, these commands require longer and more descriptive names, such as `remove_cells_from_pblock`. Vivado's command set is also lacking any fundamental organization or hierarchy, and dumps all of its commands into the global namespace. Thus, there is no natural division between commands such as the `get_cells` and `link_design`, as opposed to an object oriented programming language in which these commands would be encapsulated by the class they operated on.

TincrCAD solves these problems by encapsulating groups of related commands into ensembles named for the class they operate on, as can be observed in figure 4.1. For example, all commands that query BELs are contained within the `bels` ensemble, including `get`,

Table 4.1: Listing of all packages in Tincr

	Package	# Commands	Description
TincrCAD	`design`	47	Design-related operations
	`device`	41	Device-related operations
	`placer`	3	Placer implementations
	`extract`	12	Extract various data from Vivado
	`util`	39	Extends the functionality of Tcl
	`regression`	3	A complete regression suite
	`maintenance`	5	Commands for maintaining Tincr
TincrIO	`design`	4	Export or import Tincr Checkpoints
	`device`	4	Commands for generating device files

`foreach`, and `unique`. Using ensembles prevents both the need for overly descriptive names and allows objects to be operated on in a manner similar to an OOP language.

Packages

TincrCAD and TincrIO are each divided across a number of packages, as shown in figure 4.1. Each package contains a set of commands that belong to some category of functionality. For example, all commands that operate on device objects may be found in TincrCAD's `device` class. This gives the user the option to choose which types of commands are added to Vivado without needing to import all of Tincr's command set. Table 4.1 lists all of the packages available in the TincrCAD and TincrIO libraries.

At the core of TincrCAD are the `design` and `device` packages. These two packages offer the primary functionality that is required by many of the other packages in Tincr and will be discussed in the following section.

4.3 Functionality

TincrCAD is a fully-featured CAD tool framework for Tcl-based FPGA applications. TincrCAD is able to increase the productivity of a CAD tool designer by eliminating the need to understand the fine-grained mechanics of Vivado, instead allowing them to code their applications through a high-level API. TincrCAD abstracts away many of the low-

Table 4.2: Listing of all ensembles available in TincrCAD's `design` package

Ensemble	# Subcommands	# Lines
`designs`	9	441
`cells`	10	212
`pins`	2	41
`ports`	1	16
`nets`	22	588
`clocks`	1	16
`libs`	1	16
`lib_cells`	1	16
`lib_pins`	1	16
`macros`	1	16
`pblocks`	1	16

level details inherent in Vivado's command set, replacing them with a rich set of commands tailored to the implementation of FPGA CAD tools.

This section introduces the main functionality available in TincrCAD. It focuses on the two core packages in TincrCAD, `design` and `device`. A brief overview of the remaining packages in the TincrCAD library is given, and the section concludes in an example illustrating the amount of effort that is avoided by using an average command in TincrCAD.

4.3.1 Design Package

TincrCAD's `design` package provides methods for querying and modifying a design at a higher level of abstraction. As Vivado's low-level routines provide all the required functionality for manipulating a netlist in Vivado, the `design` package is primarily focused on making complex operations easier to implement. The package itself is made up of a number of ensembles, one for each major design-related class in Vivado. Table 4.2 lists all ensembles found in the `design` package, as well as the number of subcommands in the ensemble and the number of lines spanned by these commands. Of these, a few are worth discussing in more detail, namely the `designs`, `cells`, and `nets` ensembles.

The `designs` Ensemble

The `designs` ensemble provides a number of subcommands for operating on entire designs. For example, the `designs diff` command performs a comparison of two separate designs and reports any differences between them. The `designs clear` command clears the netlist of a design. The `designs edif` command serves as a wrapper for Vivado's `write_edif` command, and dumps the netlist of a design to whatever file the user specifies in the EDIF format.

The `cells` Ensemble

All operations related to creating, removing, manipulating, querying, and placing cell objects are encapsulated in the `cells` ensemble. While the `cells` ensemble seems to recycle stock commands such as `get_cells` and `place_cell`, it actually augments these commands with additional functionality. For example, the `cells get` command is based on Vivado's `get_cells` command, but also allows the user to get the cells of a cell (hierarchical), something that is illegal for `get_cells`. Additionally, the `cells place` command mimics the functionality of Vivado's `place_cell` command by setting the `BEL` and `LOC` properties of a cell, but also performs a complex site type negotiation algorithm to find a configuration that has all the BELs needed by cells on that site, a necessary operation considering the dynamic nature of some sites and something `place_cell` does not do.

Additionally, the `cells` ensemble adds completely new commands such as `cells compatible_with` and `cells insert`. The `cells compatible_with` function accepts as input a BEL or site and returns a list of cells in the current netlist that can be placed on it. Section 4.3.4 provides a detailed description of `cells insert` as an example of benefits of TincrCAD.

The `nets` Ensemble

The `nets` ensemble consists of a set of subcommands that operate on the nets and connectivity of a design. Among its greatest contributions to TincrCAD are its getter functions, which allow the user to obtain a much better representation of a net's composition than Vivado offers. For instance, Vivado lacks a function for obtaining an ordered list of the

nodes in a net. The `nets list_nodes` command returns a list of the node objects of a net, in the order they are traversed from source to sinks. A command like this is invaluable for any CAD tool that deals with routing, as the alternative Vivado method would require the user to parse the net's routing string (which is written using the relative names of nodes) and recursively follow the path of nodes from the source to each sink until the desired node was found. In the end, `nets list_nodes` saves the CAD tool designer 119 lines of code. Other getter commands include `nets get_source`, `nets get_sinks`, `nets get_pips`, `nets get_routethroughs`, and `nets get_root` which returns the top-level net for any hierarchical net.

TincrCAD's `nets` ensemble also provides commands for easily manipulating a net's routing and connectivity. The `nets add_node` command allows the user to add a node to a net. The command automatically searches the net for the correct location to add the node, either by creating a new branch or appending it to an existing one, and then generates and assigns a new routing string to the net. The `nets add_pip` does the same thing by adding the node at the other end of the PIP to the net's route. The `nets float` command quickly disconnects the net from all of its input and output pins, effectively making it a "floating" net in the design.

The `nets` ensemble includes a set of commands that are useful for cost functions in applications such as routers. The `nets manhattan_distance` reports a net's manhattan distance in terms of tiles, and `llength [nets get_branches]` can be used to count the number of branches in a route.

4.3.2 Device Package

The primary goal of the `device` package is to provide the user with a reliable and complete view of the physical FPGA device being targeted. A secondary goal is to provide this information in a clear and intuitive manner. Unfortunately, as chapter 3 points out, Vivado's Tcl interpreter is missing a large amount of key information from its device representation. As this information is often closely involved with operations that affect the placement and routing of a design, TincrCAD's `device` package fills in these gaps and provides a seamless interface through a number of ensemble command structures. These are

Table 4.3: Listing of all ensembles available in TincrCAD's `device` package

Ensemble	# Subcommands	# Lines
`parts`	1	16
`tiles`	7	125
`sites`	13	403
`site_pins`	2	25
`site_pips`	1	16
`bels`	11	224
`bel_pins`	2	37
`package_pins`	1	16
`nodes`	7	145
`wires`	2	26
`pips`	6	56

listed in table 4.3. The most notable among these are the `sites`, `bels`, `nodes`, and `pips` ensembles.

The `sites` Ensemble

The `sites` ensemble extends Vivado's native site interface in a number of ways. First, Vivado provides minimal handling for alternate site types (see section 3.3.2), that is, it provides no methods for navigating and controlling alternate site types other than a couple of undocumented properties. TincrCAD solves this problem by incorporating a rich API for manipulating site objects in Vivado, which is invaluable when implementing a placer in Vivado (see section 4.4). Second, Vivado lacks all intrasite routing information which is necessary for the implementation of a router. The `sites` ensemble provides a preliminary framework for obtaining this information from a database that is built outside of Vivado.

The `sites` ensemble implements a number of commands that enable a user to easily navigate and manipulate sites with alternate types. For example, the `foreach` subcommand is a control flow operation that iterates through every site and its alternate types while still supporting Tcl's `break` and `continue` commands. It does so by actually changing each site's configuration to the corresponding type and restoring its previous configuration once the control flow leaves the site. Since placed cells prevent a site from changing types (as BELs could potentially change), these sites are only represented in their current type. The `sites`

`foreach` command has been optimized to run quickly and efficiently, while still preserving the device's configuration. Additional subcommands associated with alternate site types include `get_types`, `set_type`, and `has_alternate_types`. All of these operate off of cached information and thus run very quickly.

For obtaining connectivity information of a site, the `sites` ensemble provides an interface that supplements the missing intrasite routing information that is discussed in section 3.3.2. The `get_site_wire_sinks` accepts as input the source of a site wire (either a site pin, BEL pin, or site PIP) and returns all of the site pins, BEL pins, and site PIPs that are sinks of that site wire. This information is obtained by parsing the primitive definition section of the corresponding part's XDLRC file. If no XDLRC file can be obtained for a given part through ISE's `xdl` executable, RapidSmith has a tool that can be use to generate a primitive definition for a site type by drawing the circuit in a GUI. This can be used alongside Vivado, which (despite not offering site wire information through the Tcl interface) shows all intrasite routing in its device browser view, and may be used as a reference. Future work will involve moving this information out of the `sites` ensemble, as discussed in section 6.3.

The `bels` Ensemble

When a site changes types, the BELs inside may also change. Unfortunately, Vivado does not account for these changes in its API. In fact, the only way the user can even learn of them is through manually changing a site's type and getting the list of BELs inside. This is a costly operation and is necessary for placed cells of certain types. TincrCAD's `bels` ensemble addresses this issue by caching a list of all the possible BELs on a device and the site types they correspond with. This allows the user to easily search a list of all BELs available for placement on a device and also establish mutual exclusivity between BELs that exist in conflicting site types.

There are several very useful subcommands that the `bels` ensemble is able to provide due to these caches. First, the `get` subcommand extends Vivado's native `get_bels` command with the ability to return all "possible" BELs using the `-all` switch. `get_site_types` returns a list of the different site types the BEL is instanced in. The `foreach` subcommand allows the user to iterate through all the possible BELs on the device. Finally, if the `-all` switch

is used with the `bel_pins` ensemble's `get -of $site` command, the information from the `bels` ensemble is used to populate the list that is returned.

Another useful function is `bels compatible_with` which, given a cell, will return a list of BELs that cell can be legally placed on (it should be noted that this information is otherwise unavailable in Vivado and is generated by Tincr).

The `nodes` and `pips` Ensembles

TincrCAD's `nodes` and `pips` ensembles are solely concerned with making it easier to access the routing structure of a part. In Vivado, nodes and PIPs are static constructs and do not require any special handling, so when a user makes a call to either ensemble, it is because they are only concerned with learning about a part's routing network. For example, it is sometimes useful to determine whether or not two nodes have a PIP between them (and thus may be interconnected to form a longer wire). To accomplish this, TincrCAD provides the `pips between_nodes` function, which will return such a PIP object if one exists, or an empty string if it does not. To do this outside of TincrCAD, the user would have to generate the sets of PIPs associated with each of the nodes and then take the intersection of the two sets, all while taking PIP directionality into account. TincrCAD also provides both ensembles with the `hops` subcommand which returns a set of nodes or PIPs that are a certain number of hops away from the specified node or PIP respectively.

4.3.3 Other Packages

There are several other packages in Tincr that build off of the commands provided by the `design` and `device` packages. The `extract` package consists of several methods for extracting large quantities of data from Vivado. One example is the `object_relationships` method, which will output a .DOT graph file of the "has-a" relationships between Vivado objects (i.e. a pin has-a cell). Other functions have been implemented to print out available Tcl commands, namespaces, and variables. The `util` package contains several general but useful Tcl procedures. This includes commands for parsing procedure arguments, reporting a procedure's runtime, or spawning a parallel Vivado instance as mentioned in section 4.2.1. This package also includes several methods to augment existing Tcl data structures and

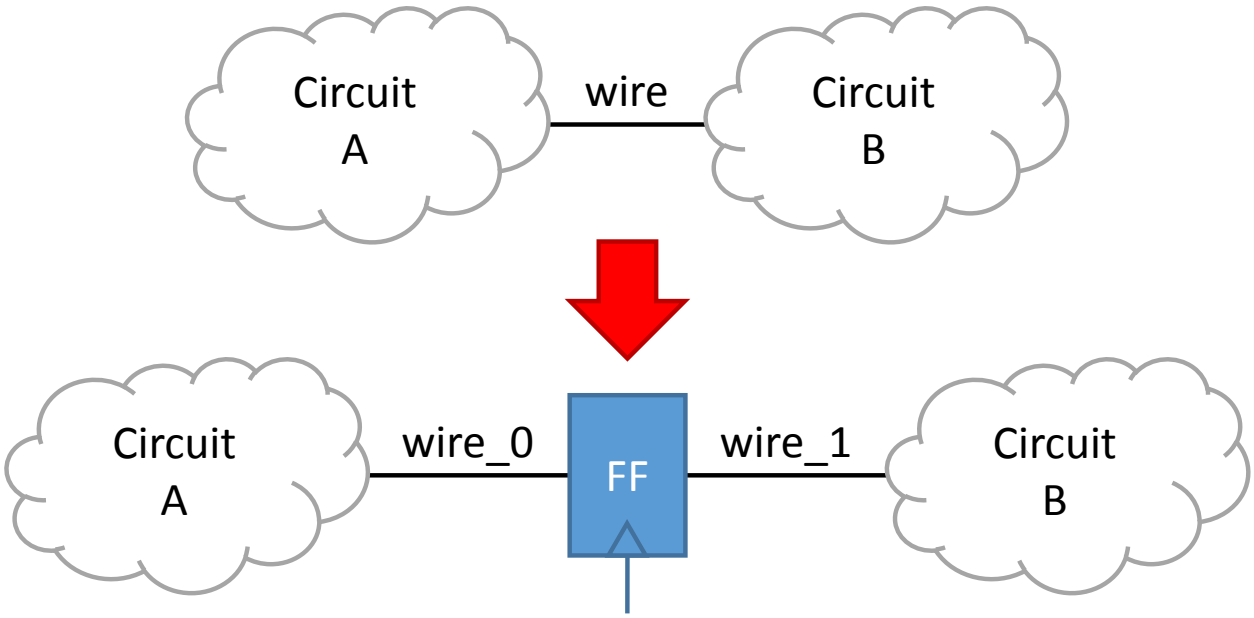

Figure 4.2: Example operation of Tincr's `cells insert` routine

functions, such as a binary search method for lists, string operations, and an `lappend` extension of Tcl's `dict` construct. Finally, Tincr also has a `regression` package that contains a fully-featured regression suite for testing Tincr against current and future releases of Vivado. For a full listing of all packages currently available in TincrCAD, see table 4.1.

4.3.4 Example: Insert a Cell into a Net

To help illustrate the advantages of TincrCAD, this section walks through the implementation of one of its functions. In FPGA design or research, a designer might face a situation that requires a cell be inserted in the middle of a net as shown in Figure 4.2. Examples of the use of this include inserting pipeline registers or NOT-ing an input to combinational logic. Such an operation could be accomplished using the following algorithm:

1. Record the original net's connectivity
2. Remove the original net from the design
3. Create the new cell

4. Create the preceding and succeeding nets and wire the new cell into the design using them

5. Return the cell

In the `design` package, the `cells insert` function serves such a purpose. Its prototype is defined below and it takes two arguments — a net object called `net` and a list of arguments stored in the variable `args`. The latter is a keyword reserved for passing an indeterminate number of arguments into a function in Tcl. In this case, it is used to describe a number of parameters for the cell that will be created.

```
1 proc ::tincr::cells::insert { net args } {
```

In the main body of the routine, the first task is to initialize default values for the input parameters as shown below. Note that the default operation is to insert a buffer cell. The call to `::tincr::parse_options` then parses the list `args`, setting any parameters that were provided in the function call in the following format: `-<parameter> <value> ...`

```
2   # The default parameter values
3   set type BUF
4   set name ${net}_$type
5   set inpin I
6   set outpin O
7   set net0_name ${net}_0
8   set net1_name ${net}_1
9   ::tincr::parse_options $args
```

The next step is to obtain pointers to the original net's sources and sinks. Using Vivado's `get_pins` command, line 12 assigns the source pin of the original net (if there is one) to the variable `in_pin`. This is done by getting all the pins of (`-of_objects`) `net` and then filtering out any whose `DIRECTION` property is set to `OUT`. The `-quiet` option is used to suppress warnings if no such pins are found. Line 13 checks to see if `net` is driven by a top-level port using the `get_ports` command, which one would expect if no source pins were found. Note that the direction of a port is described from the entire design's perspective,

unlike a pin which is described from the perspective of the cell it resides in. Similarly, lines 15 and 16 get the sinks of `net`, whether they be pins or ports, and assign them to the `out_pins` and `out_ports` lists, respectively. Since a net may have multiple sinks, these two variables may be lists.

```
12  set in_pin [get_pins -quiet -filter {DIRECTION==OUT} -of_objects $net]
13  set in_port [get_ports -quiet -filter {DIRECTION==IN} -of_objects $net]
14
15  set out_pins [get_pins -quiet -filter {DIRECTION!=OUT} -of_objects $net]
16  set out_ports [get_ports -quiet -filter {DIRECTION!=IN} -of_objects $net]
```

The connectivity of `net` is completely described by the four variables set in lines 12–16. This being the case, `net` may now be removed from the design. First, however, `net` must be disconnected from any of its sources and sinks as is done in line 18. Then, `net` is removed from the design using the `remove_net` command in line 19.

```
18  disconnect_net -quiet -net $net -objects   "$in_pin $in_port $out_pins
       $out_ports"
19  remove_net $net
```

The next step is to create the new cell in line 21 using the `create_cell` Vivado command. The new cell is created with the name stored in the `name` parameter which was defined earlier. To create a cell, one must provide a reference cell from which the new cell derives its pins and default properties. This can either be a user-defined cell or a primitive cell from the list of library cells that the `get_lib_cells` command returns. In either case, the `type` parameter is used to convey the cell's type. The resulting cell is then assigned to the variable `cell`. Finally, in a manner similar to that used to obtain `net`'s sources and sinks, lines 22 and 23 get pointers to the new cell's pins identified by `inpin` and `outpin` and store them in `cell_in` and `cell_out`, respectively.

```
20  # create the cell
21  set cell [create_cell $name -reference $type]
22  set cell_in  [get_pins -filter "REF_PIN_NAME==$inpin"  -of_objects $cell]
23  set cell_out [get_pins -filter "REF_PIN_NAME==$outpin" -of_objects $cell]
```

Lines 25 and 26 then instance two new nets using Vivado's `create_net` command. By default, these nets are given names based off `net`'s name with a number appended. They are assigned to the variables `net0` and `net1` and added to the design.

```
24  # create the nets
25  set net0 [create_net $net0_name]
26  set net1 [create_net $net1_name]
```

At this point, the only thing left to do is wire together the netlist. The `connect_net` command in Vivado allows one to connect a net specified by the `-net` switch to be connected to one or more pins or ports, specified by the `-objects` switch. Thus, `net`'s source, `in_pin` or `in_port`, and `cell`'s input pin, `cell_in`, are added to `net0` in line 28. Likewise, `net1` is connected to `cell`'s output pin, `cell_out`, and all of `net`'s sinks, `out_pins` and `out_ports` in line 29. In general, it should be noted that if more than one driver is assigned to the same net, then Vivado will throw an error and the operation will not complete. Also, when calling the `connect_net` command, the net must be unrouted and all of the objects it is being connected to must be unplaced. Finally, once all this has completed, the function returns a pointer to the created `cell` in line 31.

```
27  # connect the new nets to the new cell
28  connect_net -quiet -net $net0 -objects      "$in_pin $in_port $cell_in"
29  connect_net -quiet -net $net1 -objects      "$cell_out $out_pins
        $out_ports"
30
31  return $cell
32 }
```

In this example none of the circuitry involved in the design modification was placed or routed. If it had been, the above code could be augmented with steps to save the placement information of the affected circuits as well as the routing information of its associated routed nets. The affected circuits would then be unplaced and unrouted (by modifying their object properties). Once the newly-created cell was inserted and interconnected, the placement and routing properties would be restored for the affected circuits, the new cell placed, and the

new nets routed. The routing of the new nets could be done using custom router code or it could be done using Vivado's `route_design -net <netName>` Tcl routine (which can be called to route individual nets).

4.4 Case Study: Implementing a Placer in Tincr

Tincr offers procedures that significantly reduce the amount of work needed to implement CAD tools in Vivado Tcl. To illustrate the benefits Tincr provides, this section walks through the implementation of a simple random placer using Tincr. Without Tincr, implementing a placer using Vivado's native low-level commands is non-trivial, and requires very specific knowledge about the composition of the device, the relationships between different types of cells and BELs, and more. Tincr encapsulates nearly all of this peripheral information into a handful of useful functions that makes writing a placer in Vivado's Tcl interface a trivial task.

4.4.1 Random Placer

The tool implemented for this case study is a very simple random placer. This particular tool was written solely as a demonstration of the possibility for writing a placer in Vivado, and is not intended as a replacement for Vivado's proprietary placer. In order for this placer to operate correctly, the user must have a copy of Vivado installed (version 2014.2 or higher recommended) accompanied by an install of the latest Tincr tools. To route the design, use Vivado's routing tool, which will require a valid implementation license. To write out a bitstream, a license is also required.

The random placer simply places every primitive cell in the netlist onto a compatible BEL chosen at random. This placer uses no cost functions and thus has no way of measuring the quality of its result. The algorithm for this placer is as follows:

```
1 cells = All leaf cells in the netlist
2 foreach cell in cells {
3   if (cell is placed) skip it
4   if (cell is IO or GND/VCC) skip it
5   bels = List of BELs that are compatible with cell's type
6   idx = Random integer from 0 to bels.length-1 inclusive
7   watchDog = 0
```

```
8   bel = bels[idx]
9   while (bel is used or invalid) {
10    watchDog++
11    if (watchDog >= bels.length) throw error: "Placement failed."
12    idx++ (wrap to 0 when idx == bels.length)
13    bel = bels[idx]
14  }
15  place cell on bel
16}
```

To summarize, the algorithm dictates that for every leaf cell in the netlist, the following must occur: First, skip the cell if it is already placed, I/O related, or global logic. Cells associated with I/O (such as buffers) are placed with the ports they are connected to, while cells that are part of the global logic network (i.e. VCC or GND) are just placeholders and should not be placed. Second, obtain a list of all BELs on the part that are compatible with the current cell's type. In this case, a cell's type refers to the library cell it references. Next, chose one BEL from this list at random. If this BEL is already used, or if the cell cannot otherwise be placed on the BEL, move on the the next sequential BEL in the list and try again. If a valid BEL is found, the placement is considered legal and the algorithm places the cell on it and moves on to the next cell. If all BELs in the list have been evaluated and none of them are legal, then throw an error. Once the algorithm has processed every leaf cell, then placement has completed. The fulll implementation may be found in appendix B.

4.4.2 Implementation

To implement the algorithm outlined above, some basic functionality is required. This section seeks to identify a set of commands that most effectively yield said functionality, drawing from either Vivado's native command set or Tincr's extended set of commands. Ultimately, these Tcl commands will be brought together in a Tcl procedure that will realize the algorithm defined above, and then be tested on a moderately sized design. Runtime and various design measurements will be provided to give the reader some indication of the feasibility of such a tool.

Required Functionality

Aside from the basic data and control constructs available in Tcl 8.5, the following functionality is need to implement a rudimentary placer in Vivado:

- Obtain a set of primitive cells
- Indicate whether or not a cell is already placed
- Get the type (library cell) of a cell
- Obtain a set of BELs that are compatible with a specific cell type
- Determine whether or not a placement is legal
- Place a cell on a BEL

While these operations may seem relatively elementary, that does not necessarily mean that they are trivial to implement using Vivado's native command set. For example, the type (or library cell) of a cell may be obtained using a single intuitive command, while getting a set of compatible BELs for that type is a complex operation requiring a thorough understanding of the polymorphic nature of sites and how it affects the availability of BELs on a part. Even placing a cell is a complicated procedure, as it involves negotiating BEL availability across a number of different site layouts.

Commands

With the use of Tincr, the operations outlined in the previous section can be performed with minimal effort. Table 4.4 lists these operations and the commands chosen to execute them, as well as their namespace (Vivado or Tincr) and compression ratio. In this context, the compression ratio is a measurement of the amount of typing the listed command saves the user. Specifically, this is the number of lines of code it takes to execute the given command divided by the number of lines of code within the command. Since all operations can be executed using a single command, this simply becomes the reciprocal of the number of lines of code within each command. Hence, a command from Vivado's namespace will

Table 4.4: Placer operations and their corresponding commands

Functionality	Command	Command Set	Compression
Get primitive cells	`cells get_primitives`	Tincr	1
Is a cell placed?	`cells is_placed`	Tincr	1
Get a cell's type	`get_lib_cells -of`	Vivado	1
Get compatible BELs	`bels compatible_with`	Tincr	0.0079
Is a placement legal?	`cells is_placement_legal`	Tincr	0.2
Place a cell	`cells place`	Tincr	0.03

always have a compression ratio of one, while a command from Tincr will be either less than or equal to one. The smaller the compression ratio, the less work the programmer has to do.

It should be pointed out that, while the first two operations use commands from Tincr, their compression ratio is still unity. This seems to suggest that the command is not saving the programmer any time. Since the compression ratio is just measured by lines of code, it is unable to account for the amount of work a command saves the programmer either because of the length of the command or the amount of low-level knowledge required to correctly execute the operation. For example, the command `cells get_primitives` returns a list of primitive cells in a netlist. Its implementation is a single line of code:

```
1 return [get_cells -hierarchical -filter {PRIMITIVE_LEVEL==LEAF ||
    PRIMITIVE_LEVEL==INTERNAL}]
```

The successful execution of this operation requires knowledge of the `-hierarchical` switch, which tells Vivado to include cells from all levels of hierarchy (rather than just the top level). This is well documented by Xilinx under its definition of the `get_cells` command in [18]. It also requires specific knowledge of the `PRIMITIVE_LEVEL` property in a cell and that two of its three possible values, `LEAF` and `INTERNAL`, qualify that cell as primitive. This particular piece of information is not documented by Xilinx and can be a source of frustration for developers. Thus, Tincr is able to provide a higher layer of abstraction to this specific operation in a single line of code.

Optimizations

The final implementation of the random router employs a number of optimizations to its original algorithm which are outlined here. These optimizations are not included in the main discussion of this placer because they are not relevant to its goals (i.e. to demonstrate the benefits of Tincr in the development of a placer). They are mentioned here in the spirit of full disclosure.

Perhaps the most significant optimizations lie within the code of the Tincr commands themselves. As mentioned previously, Tincr's first priority is performance, which has led to the caching of nearly all of its generated information. That being said, this placer accesses several tiers of these caches throughout its entire operation. For instance, the `bels compatible_with $cell` command references a cache of cell type to BEL mappings, which is generated using a cache of site type to BEL mappings, which in turn is generated using a cache of site to site type mappings. Bearing this in mind, the initial run of `bels compatible_with` will be significantly longer than subsequent runs. For this random placer, the benefits of these caches becomes apparent as the same types of cells are placed repeatedly, such as LUTs or flip flops.

Currently this placer uses Vivado's own `place_cell` command to determine the validity of a placement instead of Tincr's `cells is_placement_legal` command. This is done by calling `place_cell`, and then catching any errors that the command throws. If there are no errors, then it is a legal placement. While this may seem like a hack, there is a good reason behind it. In Vivado, there are a few rules regarding what configurations of cells can coexist within a site that `cells is_placement_legal` is currently unable to detect. In a slice for example, all cells that instance flip flops must use the same set of control signals, otherwise the placement is illegal. Integrating these rules into Tincr requires information on the routing network within a site that is presently unobtainable, although efforts are being made towards a workaround for this issue. Once this information is available to Tincr, the `cells is_placement_legal` will be able to return the correct value for all cases.

4.4.3 Results

It is worth reiterating that this random placer is not intended for any real world applications, but is meant as a demonstration of the possibility of implementing a placer in Vivado. To prove its credibility, this random placer was executed on a legitimate design, and measurements were made to evaluate its performance. Xilinx's example BFT design from [20] was used, and may be found in the `<Vivado install directory>/examples/Vivado_Tutorial` directory. All of the same synthesis settings as `create_bft_kintex7_batch.tcl` were used to generate the netlist, including the part targeted which was the xc7k70tfbg484-2. It was tested on a computer running Windows 8.1, Vivado 2014.2, and Tincr 1.0, with an Intel Core i7 processor at 2.40 GHz and 16 GB of RAM.

The BFT's creation script was executed, which in turn generated a checkpoint called `post_synth.dcp`. The experimental procedure is as follows:

```
1 open_checkpoint post_synth.dcp
2 tclapp::byu::tincr::random_placer -seed 85
3 route_design -directive Quick
4 write_bitstream out.bit
```

First, the checkpoint was opened in Vivado and then Tincr's random placer was called and its runtime recorded. To repeat the results of this work, use the seed 85. Vivado's proprietary router was then called on the placed design and a bitstream was written once it completed. To speed up the routing process, the `Quick` directive was used. Figure 4.3 depicts the placed and routed BFT design using Vivado's proprietary placer versus the random placer.

The BFT design has a total of 4,223 primitive cells, which includes 2,230 LUT, 1,562 FF, 96 DSP, and 16 BRAM cells. There are two runtime measurements that are of interest in this case, the runtimes of the placer operating off of a cold cache or a warm cache. The cold cache measurement was recorded after all caches had been cleared and Tincr had to regenerate all of its site and BEL information. The warm cache measurement was recorded after all of the caches had been completely populated. The placer placed the entire design from a cold cache in 04:00.183, and took 00:16.671 to place it from a warm cache. This

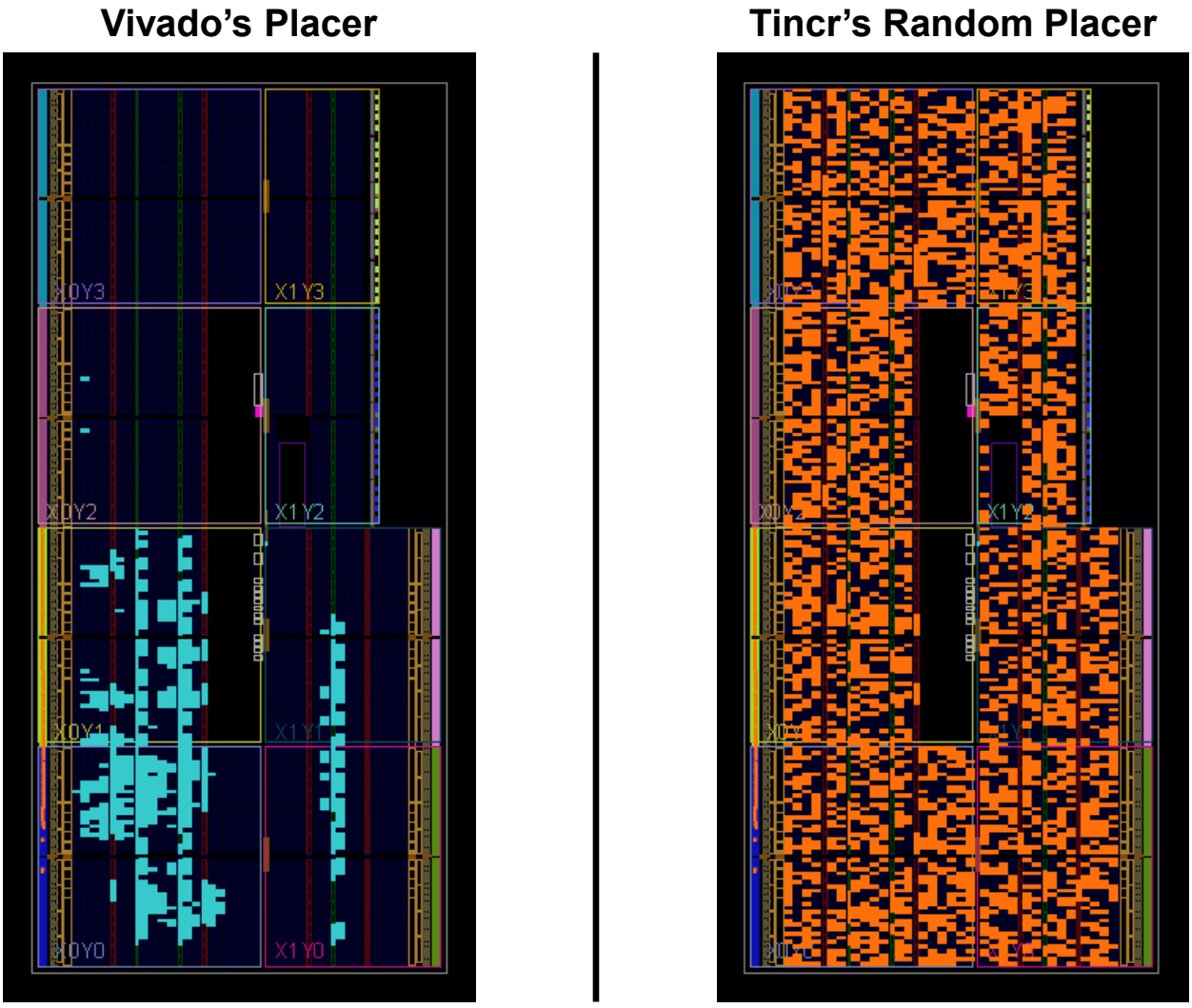

Figure 4.3: Depictions of the BFT design placed using Vivado's proprietary placer (left, in blue) versus Tincr's random placer (right, in orange).

suggests that it takes approximately four minutes to populate the cache from scratch. Bear in mind that this is a fixed overhead as it is completely independent of design size. These two measurements can provide a rough basis for extrapolating the performance of placers implemented in Vivado's Tcl environment using Tincr.

4.5 Conclusion

This chapter introduced the Tincr suite and gave an in-depth discussion on its TincrCAD library. TincrCAD was designed for three primary purposes:

1. Provide a high-level, CAD tool centric API into Vivado's Tcl interface.

2. Extend Vivado's functionality by manually adding missing information.

3. Seamlessly manage complex constructs such as sites and BELs.

This chapter illustrated how TincrCAD fulfills each of these purposes through the various packages it offers. Collectively, they comprise a complete Tcl-based CAD tool framework upon which a number of applications may be realized. This chapter showed how one would implement a placer tool using the TincrCAD command set. The following chapter discusses the second half of Tincr, its TincrIO library, and how it can be used to implement CAD tool frameworks outside of Vivado.

Chapter 5

TincrIO: An External CAD Tool Framework Interface into Vivado

Chapter 4 discusses TincrCAD, a Tcl-based CAD tool API built inside of the Vivado environment. It gives an example of how such a framework can be used to implement a rudimentary placer to operate on designs in Vivado. This chapter proposes the second of two possible methods for implementing a CAD tool framework for Vivado, a file interface similar to XDL called TincrIO.

5.1 Motivation

While TincrCAD does provide a fully featured framework for implementing CAD tools in Vivado, it still has a few disadvantages — primarily the result of its base language, Tcl. As it is written in an interpreted language, TincrCAD experiences slower runtimes than comparable frameworks written in compiled languages. Also, Tcl does not support many of the useful features found in more popular languages such as Java. Finally, Tcl is not a widely used language, and as a result there are no known CAD tools already written in Tcl.

Considering these disadvantages, the second part of this work focuses on creating an interface for external CAD tool frameworks like RapidSmith and Torc to interact with Vivado. Since most existing external CAD tool frameworks that target Xilinx FGPAs are built on top of XDL (see section 2.2), it makes sense to model such an interface after XDL. In order to operate in a capacity similar to XDL, however, this interface must meet three basic requirements. First, it must be able to export a functionally equivalent design description that can be represented in a human-readable ASCII format (like `xdl -ncd2xdl`). Second, this interface must be able to parse said format and populate Vivado's data structures to match the exact state represented (like `xdl -xdl2ncd`). Third, this interface must be able

to extract a complete device description into a human-readable ASCII format (like `xdl -report`).

This work introduces the TincrIO library, a set of Tcl commands that collectively fulfill these requirements using primitive Vivado operations alongside commands from the TincrCAD library. Using TincrIO, a user can interface directly with Vivado through an external CAD framework of his or her choice. A member of the Tincr suite, TincrIO brings with it a number of advantages:

1. CAD tools written for existing frameworks may be repurposed for Vivado designs.

2. Better performance, either because the base language is faster (Torc is written in C++, a compiled language) or a number of optimization mechanisms have been implemented (see section 2.3.1).

3. Existing frameworks have already undergone an extensive verification process.

This interface is made up of two components which are organized into the `design` and `device` packages. The `design` package provides the framework for exporting designs from or importing them into Vivado via a set of archived files, in a fashion similar to XDL files in ISE. The `device` package handles the extraction of device information through a single XDLRC file. Both are discussed further in this chapter.

5.2 Design Interface

The `design` package encompasses TincrIO's design interface. Unlike TincrCAD, ensembles are not used in the organization of the commands. Instead the package itself is comprised of a number of commands that center around the importing and exporting of designs to and from Vivado. This section describes and justifies the methods used to implement this interface.

5.2.1 File Format

As discussed in section 3.1.3, Vivado uses checkpoint files to save the current state of a design. Checkpoints are archives of files that can completely restore a design into memory

when read into Vivado with the Tcl command `open_checkpoint` (see section 3.1.3). Initial attempts to export a design to file targeted this checkpoint feature, but these efforts were eventually abandoned because the placement and routing information of a design is contained within an encrypted XDEF file, rendering it useless to external CAD frameworks.

From there, research turned to Tcl commands in Vivado's file IO category. As section 3.2.2 describes, the `write_edif` command allows a design's netlist to be written out to an unencrypted EDIF file. `read_edif` is able to read an EDIF file into Vivado and completely populate its netlist data structure. Thus, EDIF files are able to offer an interface for the netlist of a design. The placement and routing information is not conveyed, however, and must be obtained from elsewhere.

Sections 3.4.2 and 3.4.3 both discuss how object properties can be used to constrain the netlist objects of a design to specific placement and routing configurations. Additionally, the new Xilinx Design Constraint (XDC) file is able to set the properties of any netlist object, and can be both read and written by Vivado using the `read_xdc` and `write_xdc` commands respectively. Hence, XDC files may be used as a mechanism for exporting and importing the placement and routing mappings of a design by constraining elements in the netlist. Between EDIF and XDC files, an entire design's netlist, constraint, placement, and routing information can be represented.

Tincr Checkpoint

TincrIO uses what are called "Tincr checkpoints" to store a design from Vivado. This checkpoint is modeled after the Vivado design checkpoint, and primarily consists of EDIF and XDC files. Specifically, a Tincr checkpoint is an archive of the following files:

design.info Design information file. This file is unique to TincrIO and contains miscellaneous information about the design such as its part, name, and top-level module.

netlist.edf A complete description of the design's netlist in the electronic design interchange format.

constraints.xdc The design's user-defined constraints that are applied to a design through XDC files (instead of UCF files) or Tcl commands.

placement.xdc The design's placement constraints. This file consists of a number of Tcl commands that set the `LOC`, `BEL`, and `LOCK_PINS` properties of cells in the netlist. Together, these properties can completely restore the placement information of a design.

routing.xdc The design's routing constraints. This is an XDC file that contains a set of Tcl commands that set the `ROUTE` property of any routed nets in the netlist. These commands are able to mostly restore the routing of a netlist, with the exception of route-throughs.

The Tincr checkpoint was chosen because Vivado's data structures already cater to the checkpoint model. Like XDL, all files in the archive are in a human-readable ASCII format. The XDL format itself was not used because it is not compatible with Vivado's design database. This is primarily due to the site-based nature of XDL, which would require the implementation of complex packing and unpacking algorithms in Vivado. Additionally, XDL supports constructs that do not exist in Vivado, such as the hard macro. Ultimately, the decision to abandon the XDL format as a design representation was carefully made after considering a number of factors, and it is the position of this work that the Tincr checkpoint presents a much better alternative.

Converting from XDL to the Tincr Checkpoint

Existing CAD tool frameworks that target Xilinx FPGAs are outfitted for operating on XDL design representations. This means that external CAD tool frameworks that wish to utilize the Tincr checkpoint interface may need to make some changes to their design API to accommodate the differences. This subsection briefly discusses these changes.

First, as mentioned earlier, XDL is a site-based representation, while the Tincr checkpoint is BEL-based. Most frameworks that are built on XDL have data structures that are also site-based, such as RapidSmith and Torc's physical API. These data structures would have to be extended to support a BEL-level representation to be compatible with Tincr checkpoints. Torc is already able to do this through its generic API and even provides an unpacker for converting XDL designs to BEL-level representations. RapidSmith is currently in the process of extending its design and device representations to support this feature.

Second, these BEL-level design representations must be populated using the Tincr checkpoint's `netlist.edf` file. This means they will need an EDIF parser to read a Tincr checkpoint and an EDIF generation capability to write a Tincr checkpoint. The former should not be difficult to implement as there are many open-source EDIF parsing utilities available for a variety of programming languages. In fact, Torc is already outfitted for both reading and writing EDIF files to and from its generic representation.

Finally, a XDC parser must be implemented to extract the placement and routing mappings from the Tincr checkpoint's `placement.xdc` and `routing.xdc` files. Also, if the ability to write Tincr checkpoints is desired, the framework must be able to generate these files as well. This should be a trivial task, as these files are simply a sequence of Tcl commands setting a handful of properties.

5.2.2 Implementation

By design, the implementation of the Tincr checkpoint interface is simple. More complex methods and file formats were tested, and all resulted in significantly lower performance. As Tincr's primary objective is performance, these alternatives were dismissed. The ideology behind the implementation of TincrIO's design interface is to minimize the use of custom methods, using native commands and files where possible. Adhering to this principle has led to the best possible performance for a design interface into Vivado. This section will discuss the two primary functions of the design interface, exportation and importation.

Exportation

Exporting a design from Vivado is the more involved aspect of the design interface, as it requires the implementation of a few custom commands for generating the required files. Generally, there are five parts to exporting a design from Vivado, as can be observed directly from lines 8-12 of the `export_checkpoint` function in listing 5.1:

Listing 5.1: TincrIO's `export_checkpoint` function

```
1 proc ::tincr::export_checkpoint {args} {
2   parse_args {} {} {} {filename} $args
3
```

```
4   set filename [add_extension ".tcp" $filename]
5
6   file mkdir $filename
7
8   write_design_info "${filename}/design.info"
9   write_edif -force "${filename}/netlist.edf"
10  write_xdc -force "${filename}/constraints.xdc"
11  write_placement_xdc "${filename}/placement.xdc"
12  write_routing_xdc "${filename}/routing.xdc"
13 }
```

Lines 1-7 in listing 5.1 are just housekeeping items, parsing the function's arguments, making sure the given filename has the correct Tincr checkpoint extension (.tcp), and creating a directory for all of the checkpoint files to be dumped into. The ensuing lines comprise the generation of the Tincr checkpoint.

The `design.info` file is generated using the TincrIO command `write_design_info` as shown in line 8. This command simply writes any miscellaneous information about a design to `design.info`, using a very simple line-based key/value grammar: `<key>=<value>`. For instance, the part that the BFT design (section 4.4) was implemented on would be written on its own line in the file like so: `part=xc7k70tfbg484-2`.

Lines 9 and 10 are calls to native Vivado commands. The `netlist.edf` file is generated using the `write_edif` command and contains a complete description of the design's netlist. The `constraints.xdc` file is generated by Vivado's `write_xdc` Tcl command and describes all user-defined constraints on the design.

As there are no Vivado commands for generating the placement and routing constraint files, TincrIO provides the commands `write_placement_xdc` and `write_routing_xdc`, as shown on lines 11 and 12. The `write_placement_xdc` automatically generates an XDC file describing the netlist's current placement configuration for all cells, pins, and ports in the design. The `write_routing_xdc` writes out an XDC file detailing the routing configuration for every net in the design. This may or may not include global logic nets, depending on whether the `-global_logic` switch is asserted. Additionally, `write_routing_xdc` handles route-throughs using the TincrIO command `remove_route_throughs`, which automatically detects all of the route-throughs in a design and replaces them with `BUF` cells. Removing all route-throughs is necessary for preserving the exact routing of the design on a subsequent

import of this particular checkpoint. Without this, the user would have to call Vivado's `route_design` function to fix all of the route-throughs in the design, which also has the potential to change the rest of the routing configuration, thus breaking the preservation of the design.

Importation

Importing a Tincr checkpoint is relatively straight forward. Listing 5.2 shows an abbreviated version of TincrIO's `import_checkpoint` command (message statements were removed). As can be seen, Vivado's `read_edif` and `read_xdc` commands provide an efficient means for importing all Tincr checkpoint files.

Listing 5.2: An abbreviated version of TincrIO's `import_checkpoint` function

```
1 proc ::tincr::import_checkpoint {args} {
2   parse_args {} {} {} {filename} $args
3
4   set filename [add_extension ".tcp" $filename]
5
6   read_edif "${filename}/netlist.edf"
7   read_xdc "${filename}/constraints.xdc"
8   read_xdc "${filename}/placement.xdc"
9   read_xdc "${filename}/routing.xdc"
10  link_design -part [get_design_info $filename part]
11
12  place_design -directive Quick
13
14  route_design -physical_nets
15 }
```

A brief note on lines 10-14. On line 10, the `link_design` command calls TincrIO's `get_design_info` command to parse the part name out of the Tincr checkpoint's `design.info` file. As discussed in section 3.4.2, there is a flag somewhere in Vivado that only gets asserted when the `place_design` command is called. If this flag is not set, it is not possible to execute any downstream tools in Vivado's EDA flow, such as the router. Thus, line 12 calls Vivado's `place_design` command with the `Quick` directive, assuring the fastest runtime. As all of the design's cells are fixed at this point in the importation process, the placer will not alter the design's imported placement configuration. Finally, on line 14, a

call is made to Vivado's `route_design` command, which is directed to route all of the global logic signals, since these nets were not exported with the design (see section 3.4.3 for more information).

5.2.3 Verification

To prove that this method of design exportation and importation does indeed work, a verification procedure was identified and executed on a representative design. Specifically, the BFT design from [20] was synthesized, placed, and routed and then written out to both Vivado and Tincr checkpoints. All projects were then cleared from Vivado's memory and the Tincr checkpoint was imported and compared to the Vivado checkpoint using TincrCAD's `designs diff` command. This method was chosen over comparing bitstreams because of small differences in the bitstreams due to global logic nets and route-throughs in the BFT design that rendered them unreliable. In fact, even bitstreams generated from the exact same Vivado checkpoint had a significant number of differences. Notwithstanding, `designs diff` reported that both the Tincr and Vivado designs matched, suggesting that a Tincr checkpoint is able successfully represent and transmit a Vivado design.

This may seem to present a conflict of interest, using a TincrCAD function to verify a TincrIO function. Therefore, additional information will be provided on exactly how the two checkpoints were compared. As the actual `designs diff` command is over 250 lines long, a brief overview of the command's methods is provided here.

First, `designs diff` accepts two Vivado checkpoints as inputs and loads both of them into memory, creating new projects for either one. From that point, the properties of both designs are compared, and any missing properties or mismatching values are reported.

The cells are then compared by first making sure that both designs have the same number of cells with the same names. The cells' properties are compared, and they are checked to make sure that they are placed on the same BEL. The BEL object itself is not checked because it is part of the device representation — if it has the same name, it's the same BEL. The pins of each cell are also checked in the same manner as the cell: lists are compared by pin names and then properties are compared.

Next, the nets in the designs are compared in a like manner to the cells and pins. Rather than compare each net's `ROUTE` property, the node set (`get_nodes -of $net`) of each net is compared instead. This is because the `ROUTE` property uses a non-deterministic grammar, resulting in several different strings that can represent the the same route (see section 3.4.3). A net that uses the same set of nodes, however, is certain to have the exact same route.

Finally, the ports in the design are compared in a similar manner to cells. Their properties and package pins are compared. Lastly, any differences between the two designs are reported.

5.2.4 Performance

The Tincr checkpoint presents the exciting possibility of achieving XDL-like functionality for the Vivado EDA suite. While this possibility exists and has been verified, there is a serious issue preventing the Tincr checkpoint from operating in the same capacity as an XDL file.

The process of importing XDL back into ISE's tool flow (`xdl -xdl2ncd`) presented a significant bottleneck for rapid prototyping tools such as HMFlow [8]. This conversion process can take on the order of minutes, and accounts for 84% of HMFlow's total runtime. Lavin points out that for HMFlow to become a viable technique for commercial applications, the conversion process from XDL to NCD would need to be optimized [12].

Importing a Tincr checkpoint also incurs a time penalty, but to the point that it can make the whole process infeasible. Where the XDL to NCD conversion process took on the order of several minutes, importing a Tincr checkpoint can take on the order of several hours, if not days. As the placement and routing mappings are expressed as a sequence of Tcl commands, each of these commands is executed during the import process. These are normal Vivado Tcl commands that perform a number of checks every time they are called. These checks impose a relatively large amount of time on the operation itself, and actually take longer as the size of the design increases.

Figures 5.1 and 5.2 show the export and import times of dummy designs of various sizes for both a Vivado (DCP) and Tincr (TCP) checkpoint. Each dummy design consists of

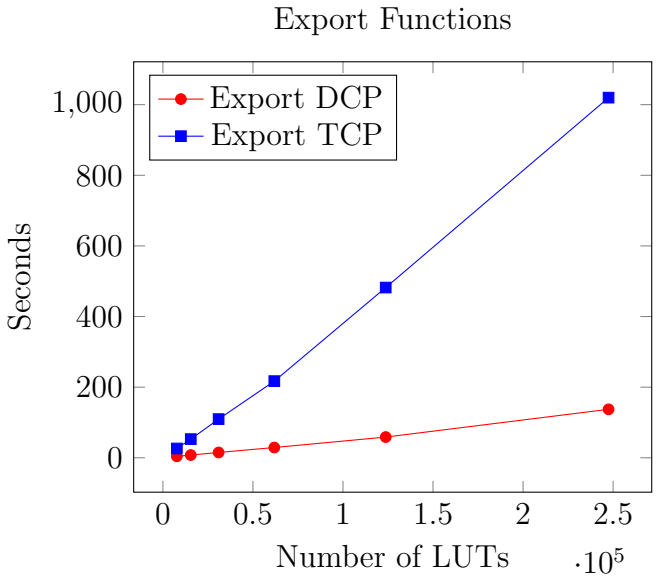

Figure 5.1: Exporting Vivado and Tincr checkpoints

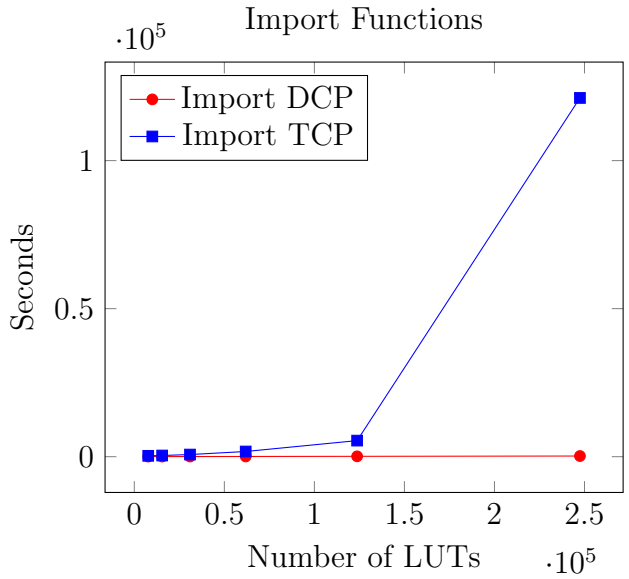

Figure 5.2: Importing Vivado and Tincr checkpoints

some number of DFT modules with their outputs all muxed together. The part used was the xc7v2000t, and all operations were run on the supercomputer in the Fulton Supercomputing Lab at Brigham Young University. The size of the designs in terms of LUTs is shown on the horizontal axis, and runtime in the vertical axis. As figure 5.1 shows, the time it takes to export a design has a roughly linear relationship with the size of the design. This is to be expected, as the exportation process simply involves dumping the design database to file.

Figure 5.2, on the other hand, shows that the runtime of importing the design back into Vivado grows exponentially with the size of the design, likely the result of the internal checks that Vivado runs as each Tcl command in the XDC files is executed. Profiles of the importation process suggest that Vivado actually slows down as more elements are processed. For instance, the first 1,000 nets in a 250,000-LUT design take just as long to import as the first 1,000 nets in an 8,000-LUT design. The majority of the time is spent in the last elements being imported.

This is opposed to the Vivado checkpoint, whose export and import runtimes are linearly related to design size. This is due to the fact that the proprietary XDEF file format in the design checkpoints can be imported very quickly, likely skipping the internal checks that XDC files are subjected to.

5.2.5 Summary

The `design` package contains the functionality for exporting and importing Vivado designs to and from an external file format similar to XDL. The exportation procedure is slower than Vivado's `write_checkpoint` command, but still completes in a reasonable amount of time. Unfortunately, the importation process takes an inordinate amount of time to complete, and in its current state, is not feasible for some applications. While it is able to import the netlist very quickly, a significant slowdown occurs during the importation of the netlist's placement and routing constraints. Testing suggests that this is due to a number of checks that Vivado imposes on the Tcl commands used to the set the `LOC`, `BEL`, `LOCK_PINS`, and `ROUTE` properties.

5.3 Device Extraction

The second component of TincrIO is device extraction, which is provided by its `device` package. The ability to extract a detailed device description is crucial to the implementation of an external CAD tool framework interface, as it is responsible for populating the framework's device database. This section describes the format chosen for representing this information as well as how its generation was implemented in Vivado.

5.3.1 File Format

As mentioned in the previous section, most external CAD tool frameworks are modeled on the XDL representation. Consequently, their device data structures closely resemble the `xdl` executable's XDLRC format. As mentioned in section 2.2.2, an XDLRC file (unlike XDL files) is considered BEL-based, as its primitive definitions section provides all the necessary sub-site information. This is a promising quality that makes XDLRC files a viable option for representing device information extracted from Vivado. To summarize, XDLRC has the following advantages:

1. XDLRC is capable of describing all the required information to adequately represent Xilinx FPGAs.
2. It is an established language that many CAD tool frameworks already support.
3. Tools are in place that can provide for the verification and debugging of XDLRC files generated by Vivado.

These are presented in contrast to the disadvantages of XDLRC, which are:

1. The language itself is verbose, and the files generated are very large.
2. XDLRC does not take advantage of any optimization techniques that are inherent to an FPGA's layout, such as the template representation.
3. XDLRC is on its way out and may be unable to represent future Xilinx FPGA constructs.

After considering both the pros and cons for using the XDLRC file format to represent devices from Vivado, it is the opinion of this work that the advantages outweigh the disadvantages. After studying the architecture of Xilinx's new Ultrascale line, there is no indication that XDLRC will become obsolete any time in the near future. Considering the verboseness of XDLRC files, this is not of much concern as the files are only generated once and can be trimmed and compressed by the external CAD tool framework itself. Therefore, the XDLRC file format was chosen to represent Vivado devices in TincrIO.

5.3.2 Implementation

TincrIO provides the `write_xdlrc` command for extracting an XDLRC file from Vivado. As mentioned in section 4.2.1, the `write_xdlrc` command employs parallelization to both speed up runtime and solve a serious memory leak that Vivado has. Ultimately, this function is able to generate a functionally equivalent XDLRC file in what is considered a reasonable amount of time.

A number of challenges had to be addressed in order to successfully extract a device description from Vivado into the XDLRC format. This section will discuss the major problems encountered during the implementation of Vivado's device extraction mechanisms, as well as their solutions.

Floating Site Pins

Due to an incomplete set of object relationships in Vivado (see section 3.2.2), the only way to get the wire that a site pin connects to is by first getting the node of the site pin, and then getting the wires of that node (`get_wires -of [get_nodes -of $site_pin]`). Unfortunately, there are situations in which a wire may exist without belonging to a node. This often happens on the edge of an FPGA, when a wire would normally be sourced or sinked by another wire in the next tile over, but there is no node because there is no tile. Since there is not a has-a relationship between a site pin and a wire, and the node is missing, there is no way to obtain the wire the site pin connects for the XDLRC's `pinwire` declaration. Figure 5.3 shows such a case, in which the `CIN` site pin is not sourced by a node because

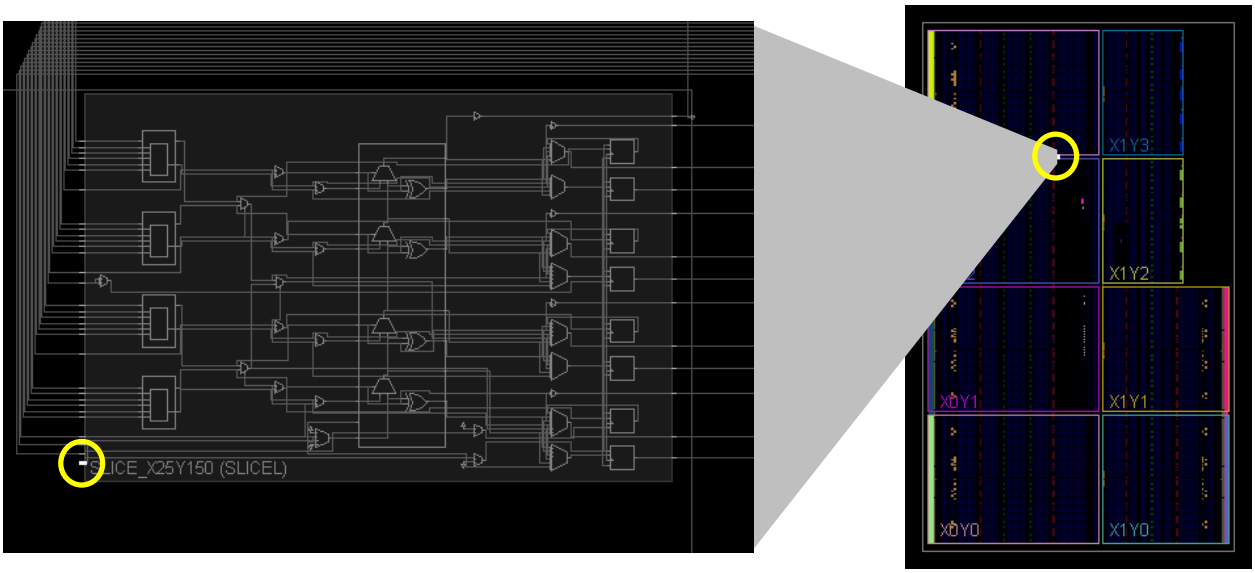

Figure 5.3: A site pin in Vivado that has no node. Without a node, it isn't possible to obtain the wire this pin connects to.

there is no tile south of it. By design every site pin has a wire, which in this case is wire `CLBLL_L_X18Y150/CLBLL_L_CIN`, but without a node there is no way of associating the two.

The only way to correctly obtain these site pin/wire pairings from Vivado is by manually checking the GUI. As a way to bypass this problem, `write_xdlrc` simply adds another (fake) wire to the tile, and connects it to the site pin in the XDLRC file. The reason this works is because the wire does not actually go anywhere, and thus adding a fake wire to the pin does not affect the rest of the routing network, achieving the same basic functionality.

Missing Resources

When XDLRC files generated by Vivado and ISE are compared literally, often ISE's file will have more resources than Vivado's file does. The reason for this is because many floating resources on the FPGA have been trimmed from Vivado's device data structure, while they still remain in ISE's. These resources are disconnected from the routing graph and have no impact on the FPGA's resource structure, and thus may safely be ignored.

Similarly, when the XDLRC files from both Vivado and ISE are compared, there are cases in which Vivado may have more wires than ISE. All tests have shown that these wires do not interconnect with the primary routing structure, and typically are a part of some

clock network that is not represented in ISE's device representation for unknown reasons. As this is the case, these wires may be included without corrupting the device representation.

Missing Intrasite Routing

As mentioned in section 3.3.2, Vivado does not provide access to its intrasite routing structure. This is a key portion of the XDLRC file's primitive definitions section, and without it an external CAD tool framework for Vivado becomes useless. Therefore, this information must be acquired manually through visual inspection of site diagrams from Vivado's device browser.

The RapidSmith team has just finished development on a tool called the Vivado Sub-site Routing Tool (VSRT), which allows users to construct a site's intrasite routing structure through a GUI that mirrors Vivado's device browser window. By positioning the two applications side-by-side, the intrasite routing for a number of site type can be manually generated in a reasonable amount of time. The generated primitive definitions are then stored in a database that TincrIO is able to access and append to its generated XDLRC files.

5.3.3 Verification

Verifying the XDLRC files that were generated by Vivado was a long and arduous process, but was instrumental in the discovery of many of the caveats discussed in section 5.3.2. The various decisions, assumptions, and compromises made throughout the verification process will be outlined and discussed in this section.

The `DeviceDiff` Class

The verification tools used to verify TincrIO's XDLRC files were built upon the RapidSmith framework. RapidSmith was chosen for its compact device files and performance optimizations. Also, previous familiarity with the tool allowed for a quicker implementation. A `DeviceDiff` class was added to the RapidSmith `Device` package and compares two of RapidSmith's compact device files. These compact device files are created from the original XDLRC using RapidSmith's XDLRC parsing utility located in the `XDLRCParser` class. The

`XDLRCParser` populates a `Device` object, which then is used to output the compact device file by calling its `writeDeviceToCompactFile` function. These compact device files are very small in size — the largest Virtex 7 part is just under 6 MB in size, as opposed to its 70 GB XDLRC file.

Since both Vivado and ISE support 7 Series Xilinx FPGAs, a compact device file was generated for each of two XDLRC files: one that was created using ISE's `xdl` executable, and the other from TincrIO's `write_xdlrc` command. These files are then read into two separate `Device` objects in RapidSmith, the former being the golden model that the latter will be compared against. The `DeviceDiff` class compares two `Device` objects by recursively calling a `diff` function on every data member in each class in the `Device` object's data structure. The `diff` function uses a polymorphic interface to correctly process any type of object that occurs in the `Device` class's hierarchy. For example, listing 5.3 shows the implementation of the `diff` function for the `PrimitiveSite` object.

Listing 5.3: RapidSmith's `DeviceDiff`'s `diff` function for `PrimitiveSite` objects

```
1 private static boolean diff(PrimitiveSite gold, PrimitiveSite test, String
    path, int properties) {
2   if (floatingSites.contains(gold)) return true;
3
4   boolean result = true;
5
6   result = diff(gold.getName(), test.getName(), path + ".getName()",
      properties) && result;
7   result = diff(gold.getType(), test.getType(), path + ".getType()",
      properties) && result;
8   result = diff(gold.getPins(), test.getPins(), path + ".getPins()",
      setProperty(properties, Property.WIRE));
9   result = diff(gold.getInstanceX(), test.getInstanceX(), path + ".
      getInstanceX()", unsetProperty(properties, Property.WIRE));
10  result = diff(gold.getInstanceY(), test.getInstanceY(), path + ".
      getInstanceY()", unsetProperty(properties, Property.WIRE));
11
12  return result;
13 }
```

The first line in listing 5.3 describes the prototype for the `diff` function. It accepts as input a golden object and a test object. The golden object is assumed to correct, while the test object is being compared to it. This method of comparing the two files brings

up an important decision made regarding TincrIO-generated XDLRC files: *An XDLRC file generated by TincrIO is considered to be functionally equivalent if it comprises a superset of the resources in an ISE-generated XDLRC file.* In other words, if all of the resources in the golden object are found and match resources in the test object, then the `diff` function will report that the two objects match. This means that the test object could potentially have more resources than the golden object, but still be considered equivalent according to `DeviceDiff`. This allows the class to handle situations such as the extra clock network found in Vivado that is mentioned in section 5.3.2.

Line 2 shows a call to a set of `PimitiveSite` objects called `floatingSites`. This set is generated in response to the resource trimming that happens to Vivado's device representation, but not ISE's (also mentioned in section 5.3.2). If a particular site is determined to be "floating" in the routing network (i.e. it is disconnected), then it is ignored and the function returns true. Floating sites are found by fanning out from a site's pins and checking if they connect to any other resources on the device. If not, the site is added to `floatingSites`.

Lines 6-10 show the actual recursive comparison of the golden and test `PrimitiveSite` objects. The `PrimitiveSite` class in RapidSmith has five data members, `name`, `type`, `pins`, `instanceX`, and `instanceY`. These lines call `diff` on each of these data members and ANDs their outcome with the `result` being returned by the calling function. If the data members match one another, `diff` returns true.

Results

The `DeviceDiff` was run on XDLRC files generated by ISE and TincrIO for a set of sample parts that span the three major families in Xilinx's 7 Series. Every XDLRC file generated by TincrIO was reported to be functionally equivalent to the XDLRC file generated by ISE.

5.3.4 Performance

Since the `write_xdlrc` command was outfitted with the parallelization optimization discussed in section 4.2.1, it has enjoyed significant runtime improvements. Prior to parallelizing the function, the `write_xdlrc` took over 4 hours to generate an XDLRC file for

Table 5.1: Runtimes (in hours:minutes:seconds) of TincrIO's `write_xdlrc` command across varying numbers of processes. For the `xc7k70tfbg484` part.

# of Processes	Runtime
1	04:17:46.482
2	02:12:18.723
4	01:10:13.387
8*	00:58:08.752

Table 5.2: Runtimes for TincrIO's `write_xdlrc` command for the largest parts in each Series 7 family.

Part	# Sites	# Nodes	Runtime
xc7a200tffg1156	60,693	7,857,396	02:44:07.648
xc7k480tffg1156	131,651	17,495,020	06:26:50.686
xc7v2000tflg1925	489,060	57,693,368	25:52:28.084
xc7z100ffg1156	122,450	16,174,017	05:52:13.082

the xc7k70tfbg484 part, as it was only running a single process. As table 5.1 illustrates, its runtime is cut down to just over an hour when the parallelized `write_xdlrc` function is run with 4 processes on a 4 core machine, realizing approximately an 4X speedup. To reiterate, the machine being used to obtain these data has a Core i7-4770 processor at 3.4GHz. The four physical cores on the machine are hyperthreaded, meaning they can each simultaneously process two threads, resulting in 8 virtual processors. When `write_xdlrc` is called with 8 processes, all 8 virtual processors are 100% utilized, but the function sees a decrease in speedup, likely the result of the over-utilization of some shared resources on the processors.

Table 5.2 shares the runtimes of `write_xdlrc` for the largest parts in each of the 7 Series FPGA families. Each of these were run with 8 processes on the machine mentioned above. The number of sites and nodes are provided for comparison purposes.

5.3.5 Summary

The `device` package contains all the needed procedures for outputting a functionally equivalent XDLRC file from Vivado. Using the `write_xdlrc` functions, XDLRC files for both 7 Series and Ultrascale devices have been generated and imported into RapidSmith.

For example, RapidSmith's Device Browser tool was adjusted to support Ultrascale parts, and was able to successfully load and navigate the `xcku035-fbva676-3-e-es1` part.

As expected, the XDLRC generation process was resource hungry, which led to unreasonable runtimes. Using Tincr's parallelization functionality, the performance of the `write_xdlrc` was significantly improved, to a factor proportional to the number of physical processor cores available on the host machine.

5.4 Conclusion

This chapter presents an alternative to the Tcl-based CAD framework presented in chapter 4. Unlike TincrCAD, TincrIO shows that it is possible to create an interface between Vivado and external CAD tool frameworks. This in turn provides a way for existing CAD tools written in XDL-based frameworks to be adapted for future generations of Xilinx FPGAs.

The `design` package introduces Tincr checkpoints as the medium for transferring designs in and out of Vivado. Unfortunately, limitations imposed by Vivado and the Tcl language cause the importation process to take an unreasonable amount of time. This could possibly be remedied through various optimizations to TincrIO's `import_checkpoint` function, which is discussed further in section 6.3.

The `device` package provides the `write_xdlrc` operation, a means by which a functionally equivalent XDLRC file may be generated from Vivado. This process has already been optimized to overcome various limitations introduced by the Vivado environment, cutting down its total runtime for all supported FPGAs to a reasonable time frame.

While there is still much expanding to be done on the work presented in this chapter, it has proven that it is indeed possible to build an interface into Vivado that is compatible with third party frameworks such as RapidSmith and Torc.

Chapter 6

Conclusions

Chapters 1 and 2 gave extensive background on the current state of FPGA CAD tool frameworks, while chapter 3 introduced the new Vivado EDA suite and described its capabilities in detail. Chapters 4 and 5 discuss two possible methods for building custom CAD tools for Vivado, TincrCAD and TincrIO. This concluding chapter presents an overview of this work, including its contributions to the FPGA CAD tool community and the natural directions of future work.

6.1 Motivation

As chapter 1 mentions, there is a demand in the FPGA community for custom CAD tool frameworks. The ability to apply custom algorithms to FPGA designs offers advantages to both designers and researchers alike. Frameworks such as RapidSmith and Torc are able to target actual Xilinx FPGAs through the XDL interface, a feature available in Xilinx's ISE design suite. These frameworks have been the proving grounds for a number of tools and algorithms that have been published across several venues.

The Vivado design suite was released as a replacement for ISE. Vivado does not support XDL, rendering XDL-based CAD tool frameworks obsolete. As chapter 3 shows, what Vivado does provide is unprecedented access into its design and device databases through a Tcl interface. It includes a number of Tcl commands for the querying and manipulation of designs in real time. This thesis set out to show how Vivado's Tcl interface could support the implementation of a CAD tool framework for the future of Xilinx FPGAs. This question was answered in the form of the Tincr suite, and its two separate methods for implementing a CAD tool framework for use in the Vivado design environment.

6.2 Contributions

The contributions of this thesis are listed below:

- This work introduced TincrCAD, a Tcl-based framework that operates within the Vivado environment. It was shown that this framework provides all the necessary functionality and resources to implement a CAD tool within Vivado. This point was punctuated by a case study on the implementation of a placer within TincrCAD.

- It also introduced TincrIO, a file interface into Vivado for external CAD tool frameworks. TincrIO uses Tincr checkpoints to convey designs between Vivado and an external CAD tool framework. While this functionality was shown to behave correctly, importing a checkpoint takes an inordinate amount of time, rendering the operation infeasible in some cases. TincrIO also provides the ability to extract functionally correct XDLRC files from Vivado.

- Chapter 3 provides a detailed discussion of the various CAD-related aspects of Vivado, many of which are undocumented by Xilinx. The material in this chapter comprises two years of research into the Vivado tool and countless hours of trial and error testing.

- This work presented performance evaluations on the methods developed as a means of supporting the conclusions made about them. The method for importing a Tincr checkpoint was declared infeasible based on the performance data collected on it. Additionally, the performance optimizations made on the `write_xdlrc` command from TincrIO were qualified by the runtime measurements obtained across a number of varying test runs.

6.3 Future Work

This thesis touches on largely unexplored territory because Vivado is still a relatively new addition to the FPGA community. At the time of writing, this is the only work that discusses the implementation of a CAD tool framework for Vivado. As a result, this work presents several opportunities for future work in its area, including expanding on TincrCAD,

optimizing features of TincrIO, and implementing CAD tools for Vivado using either of these methods.

Currently, TincrCAD's API has all the basic functionality needed to build a placer (section 4.4), among other tools. Despite this, TincrCAD still has plenty of functionality that needs to be expanded upon, the most notable being how it describes intrasite routing information. At present, this information is obtained through the `sites` ensemble, which really is just a temporary fix. The better solution for representing site wires is to create a `site_wire` class that can be queried similar to the `site_pin` class. This class could then be operated on through the `site_wires` ensemble. Another aspect of TincrCAD that could be improved upon is its alternate site types interface. At present, the `sites get -all` command returns a list of site/site type pairings. This can be confusing to navigate and perhaps would be better represented using a class.

Additionally, a number of optimizations can and should be applied to TincrCAD. One such optimization involves how its caches are populated. As opposed to generating the information from scratch (as it presently is), this information should just be generated once and written to a binary file. This could get tricky depending on how Vivado handles its pointers, but this would likely yield some performance gains.

The primary issue for TincrIO is its long design importation runtime. Further work on the importation process would likely see some performance gains. One possible approach may be to parallelize the process like was done with TincrIO's `write_xdlrc` command. Conceivably, a design could be split up into smaller parts before being exported. These parts could be imported concurrently into separate Vivado instances and saved as OOC checkpoints. From this point, these checkpoints could be read into a top-level design and stitched together.

Finally, the Tincr suite should be outfitted with a set of example CAD tools for users to use as points of reference. At the time of this writing, the `placer` package already has a simple random placer, and future work might include the implementation of a simulated annealing placer. Once the intrasite routing interface is cleaned up, a `router` package could be added to TincrCAD with a simple routing tool implemented inside. Additionally, TincrIO provides a set of files that existing tools such as RapidSmith and Torc can use to interface

with Vivado. Should the Tincr checkpoint import issue be fixed, future work may include adapting said tools to accept these checkpoints as input to their internal data structures. Once this has been accomplished, it would be an interesting pursuit to attempt to run existing CAD tools built on these frameworks on designs from Vivado.

Bibliography

[1] S. Hauck and A. DeHon, *Reconfigurable computing: the theory and practice of FPGA-based computation*. Morgan Kaufmann, 2010. 1

[2] V. Betz and J. Rose, "Vpr: A new packing, placement and routing tool for fpga research," in *Field-Programmable Logic and Applications*. Springer, 1997, pp. 213–222. 1

[3] E. Hung, F. Eslami, and S. J. E. Wilton, "Escaping the academic sandbox: Realizing vpr circuits on xilinx devices," in *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 45–52. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2013.40 2

[4] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, and J. Rose, "Vpr 5.0: Fpga cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '09. New York, NY, USA: ACM, 2009, pp. 133–142. [Online]. Available: http://doi.acm.org/10.1145/1508128.1508150 2

[5] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "Rapidsmith: Do-it-yourself cad tools for xilinx fpgas," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, Sept 2011, pp. 349–355. 2, 17

[6] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: Towards an open-source tool flow," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 41–44. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950425 2, 18

[7] K. Kepa, F. Morgan, K. Kościuszkiewicz, L. Braun, M. Hübner, and J. Becker, "Fpga analysis tool: High-level flows for low-level design analysis in reconfigurable computing," in *Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, ser. ARC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 62–73. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00641-8_9 2, 18

[8] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "Hmflow: Accelerating fpga compilation with hard macros for rapid prototyping," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, May 2011, pp. 117–124. 2, 87

[9] A. Love and P. Athanas, "Rapid modular assembly of xilinx fpga designs," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013, pp. 1–1. 2

[10] Xilinx, "Xilinx unveils the vivado design suite for the next decade of 'all programmable' devices," Press Release, Apr. 2012. [Online]. Available: http://press.xilinx.com/2012-04-24-Xilinx-Unveils-the-Vivado-Design-Suite-for-the-Next-Decade-of-All-Programmable-Devices 2, 19, 20

[11] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, B. Hutchings, and M. Wirthlin, "Rapidsmith, a library for low-level manipulation of partially placed-and-routed fpga designs," NSF Center for High Performance Reconfigurable Computing (CHREC), Tech. Rep., 2014. 11, 17

[12] C. M. Lavin, "Using hard macros to accelerate fpga compilation for xilinx fpgas," Ph.D. dissertation, Brigham Young University, apr 2012. 15, 87

[13] Xilinx, *Vivado Design Suite User Guide: Implementation*, UG904 (v2014.2), June 2014. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug904-vivado-implementation.pdf 25, 49

[14] ——, *Vivado Design Suite User Guide: Hierarchical Design*, UG905 (v2014.1), Apr. 2014. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug905-vivado-hierarchical-design.pdf 25

[15] ——, *Vivado Design Suite Tutorial: Hierarchical Design*, UG946 (v2014.2), June 2014. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug946-vivado-hierarchical-design-tutorial.pdf 25

[16] ——, *Vivado Design Suite User Guide: Using Constraints*, UG903 (v2014.1), May 2014. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug903-vivado-using-constraints.pdf 26

[17] J. K. Ousterhout and K. Jones, *Tcl and the Tk toolkit*. Pearson Education, 2009. 26

[18] Xilinx, *Vivado Design Suite Tcl Command Reference Guide*, UG835 (v2014.2), June 2014. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug835-vivado-tcl-commands.pdf 28, 31, 43, 49, 74

[19] ——, *Vivado Design Suite User Guide: Synthesis*, UG901 (v2014.2), June 2014. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug901-vivado-synthesis.pdf 43

[20] ——, *Vivado Design Suite Tutorial Design Flows Overview*, UG888 (v2014.2), June 2014. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug888-vivado-design-flows-overview-tutorial.pdf 76, 86

Appendix A

Full Adder Example

A.1 VHDL Circuit Description

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 library UNISIM;
4 use UNISIM.VCOMPONENTS.ALL;
5 entity netlist_EMPTY is
6   port (
7     a_port : in STD_LOGIC;
8     b_port : in STD_LOGIC;
9     cin_port : in STD_LOGIC;
10    cout_port : out STD_LOGIC;
11    s_port : out STD_LOGIC
12  );
13  attribute NotValidForBitStream : boolean;
14  attribute NotValidForBitStream of netlist_EMPTY : entity is true;
15 end netlist_EMPTY;
16
17 architecture STRUCTURE of netlist_EMPTY is
18  signal a_ibuf_net : STD_LOGIC;
19  signal a_net : STD_LOGIC;
20  signal and1_net : STD_LOGIC;
21  signal and2_net : STD_LOGIC;
22  signal b_ibuf_net : STD_LOGIC;
23  signal b_net : STD_LOGIC;
24  signal cin_ibuf_net : STD_LOGIC;
25  signal cin_net : STD_LOGIC;
26  signal cout_obuf_net : STD_LOGIC;
27  signal or_net : STD_LOGIC;
28  signal s_obuf_net : STD_LOGIC;
29  signal xor1_net : STD_LOGIC;
30  signal xor2_net : STD_LOGIC;
31  attribute XILINX_LEGACY_PRIM : string;
32  attribute XILINX_LEGACY_PRIM of and1_lut : label is "AND2";
33  attribute XILINX_REPORT_XFORM : string;
34  attribute XILINX_REPORT_XFORM of and1_lut : label is "AND2";
35  attribute XILINX_LEGACY_PRIM of and2_lut : label is "AND2";
36  attribute XILINX_REPORT_XFORM of and2_lut : label is "AND2";
37  attribute XILINX_LEGACY_PRIM of or_lut : label is "OR2";
38  attribute XILINX_REPORT_XFORM of or_lut : label is "OR2";
39  attribute XILINX_LEGACY_PRIM of xor1_lut : label is "XOR2";
40  attribute XILINX_REPORT_XFORM of xor1_lut : label is "XOR2";
```

```
41   attribute XILINX_LEGACY_PRIM of xor2_lut : label is "XOR2";
42   attribute XILINX_REPORT_XFORM of xor2_lut : label is "XOR2";
43 begin
44   a_net <= a_port;
45   b_net <= b_port;
46   cin_net <= cin_port;
47   cout_port <= cout_obuf_net;
48   s_port <= s_obuf_net;
49 a_ibuf: unisim.vcomponents.IBUF
50     port map (
51       I => a_net,
52       O => a_ibuf_net
53     );
54 and1_lut: unisim.vcomponents.LUT2
55     generic map(
56       INIT => X"8"
57     )
58     port map (
59       I0 => a_ibuf_net,
60       I1 => b_ibuf_net,
61       O => and1_net
62     );
63 and2_lut: unisim.vcomponents.LUT2
64     generic map(
65       INIT => X"8"
66     )
67     port map (
68       I0 => xor1_net,
69       I1 => cin_ibuf_net,
70       O => and2_net
71     );
72 b_ibuf: unisim.vcomponents.IBUF
73     port map (
74       I => b_net,
75       O => b_ibuf_net
76     );
77 cin_ibuf: unisim.vcomponents.IBUF
78     port map (
79       I => cin_net,
80       O => cin_ibuf_net
81     );
82 cout_obuf: unisim.vcomponents.OBUF
83     port map (
84       I => or_net,
85       O => cout_obuf_net
86     );
87 or_lut: unisim.vcomponents.LUT2
88     generic map(
89       INIT => X"E"
90     )
91     port map (
92       I0 => and2_net,
93       I1 => and1_net,
94       O => or_net
```

```
95      );
96  s_obuf: unisim.vcomponents.OBUF
97      port map (
98        I => xor2_net,
99        O => s_obuf_net
100     );
101 xor1_lut: unisim.vcomponents.LUT2
102     generic map(
103       INIT => X"6"
104     )
105     port map (
106       I0 => a_ibuf_net,
107       I1 => b_ibuf_net,
108       O => xor1_net
109     );
110 xor2_lut: unisim.vcomponents.LUT2
111     generic map(
112       INIT => X"6"
113     )
114     port map (
115       I0 => xor1_net,
116       I1 => cin_ibuf_net,
117       O => xor2_net
118     );
119 end STRUCTURE;
```

A.2 EDIF Netlist Description

```
1 (edif netlist_EMPTY
2   (edifversion 2 0 0)
3   (edifLevel 0)
4   (keywordmap (keywordlevel 0))
5 (status
6  (written
7   (timeStamp 2014 09 09 23 29 42)
8   (program "Vivado" (version "2014.2"))
9   (comment "Built on 'Thu Jun  5 18:17:50 MDT 2014'")
10  (comment "Built by 'xbuild'")
11 )
12 )
13  (Library hdi_primitives
14    (edifLevel 0)
15    (technology (numberDefinition ))
16   (cell LUT2 (celltype GENERIC)
17     (view netlist (viewtype NETLIST)
18       (interface
19        (port O (direction OUTPUT))
20        (port I0 (direction INPUT))
21        (port I1 (direction INPUT))
22       )
23     )
24   )
25   (cell IBUF (celltype GENERIC)
26     (view netlist (viewtype NETLIST)
```

```
27        (interface
28         (port O (direction OUTPUT))
29         (port I (direction INPUT))
30        )
31      )
32    )
33    (cell OBUF (celltype GENERIC)
34      (view netlist (viewtype NETLIST)
35        (interface
36         (port O (direction OUTPUT))
37         (port I (direction INPUT))
38        )
39      )
40    )
41    (cell INV (celltype GENERIC)
42      (view netlist (viewtype NETLIST)
43        (interface
44          (port I (direction INPUT))
45          (port O (direction OUTPUT))
46        )
47      )
48    )
49  )
50  (Library hdi_lib_etc
51    (edifLevel 0)
52    (technology (numberDefinition ))
53   (cell netlist_EMPTY (celltype GENERIC)
54     (view netlist (viewtype NETLIST)
55       (interface
56        (port a_port (direction INPUT))
57        (port b_port (direction INPUT))
58        (port cin_port (direction INPUT))
59        (port cout_port (direction OUTPUT))
60        (port s_port (direction OUTPUT))
61       )
62       (contents
63         (instance and1_lut (viewref netlist (cellref LUT2 (libraryref
              hdi_primitives)))
64           (property XILINX_LEGACY_PRIM (string "AND2"))
65           (property XILINX_REPORT_XFORM (string "AND2"))
66           (property INIT (string "4'h8"))
67         )
68         (instance and2_lut (viewref netlist (cellref LUT2 (libraryref
              hdi_primitives)))
69           (property XILINX_LEGACY_PRIM (string "AND2"))
70           (property XILINX_REPORT_XFORM (string "AND2"))
71           (property INIT (string "4'h8"))
72         )
73         (instance or_lut (viewref netlist (cellref LUT2 (libraryref
              hdi_primitives)))
74           (property XILINX_LEGACY_PRIM (string "OR2"))
75           (property XILINX_REPORT_XFORM (string "OR2"))
76           (property INIT (string "4'hE"))
77         )
```

```
78          (instance xor1_lut (viewref netlist (cellref LUT2 (libraryref
               hdi_primitives)))
79            (property XILINX_LEGACY_PRIM (string "XOR2"))
80            (property XILINX_REPORT_XFORM (string "XOR2"))
81            (property INIT (string "4'h6"))
82          )
83          (instance xor2_lut (viewref netlist (cellref LUT2 (libraryref
               hdi_primitives)))
84            (property XILINX_LEGACY_PRIM (string "XOR2"))
85            (property XILINX_REPORT_XFORM (string "XOR2"))
86            (property INIT (string "4'h6"))
87          )
88          (instance a_ibuf (viewref netlist (cellref IBUF (libraryref
               hdi_primitives))))
89          (instance b_ibuf (viewref netlist (cellref IBUF (libraryref
               hdi_primitives))))
90          (instance cin_ibuf (viewref netlist (cellref IBUF (libraryref
               hdi_primitives))))
91          (instance cout_obuf (viewref netlist (cellref OBUF (libraryref
               hdi_primitives))))
92          (instance s_obuf (viewref netlist (cellref OBUF (libraryref
               hdi_primitives))))
93          (net a_net (joined
94           (portref I (instanceref a_ibuf))
95           (portref a_port)
96           )
97          )
98          (net a_ibuf_net (joined
99           (portref O (instanceref a_ibuf))
100          (portref I0 (instanceref xor1_lut))
101          (portref I0 (instanceref and1_lut))
102          )
103         )
104         (net b_net (joined
105          (portref I (instanceref b_ibuf))
106          (portref b_port)
107          )
108         )
109         (net b_ibuf_net (joined
110          (portref O (instanceref b_ibuf))
111          (portref I1 (instanceref xor1_lut))
112          (portref I1 (instanceref and1_lut))
113          )
114         )
115         (net cin_net (joined
116          (portref I (instanceref cin_ibuf))
117          (portref cin_port)
118          )
119         )
120         (net cin_ibuf_net (joined
121          (portref O (instanceref cin_ibuf))
122          (portref I1 (instanceref xor2_lut))
123          (portref I1 (instanceref and2_lut))
124          )
```

```
125           )
126           (net cout_obuf_net (joined
127            (portref O (instanceref cout_obuf))
128            (portref cout_port)
129            )
130           )
131           (net s_obuf_net (joined
132            (portref O (instanceref s_obuf))
133            (portref s_port)
134            )
135           )
136           (net and1_net (joined
137            (portref O (instanceref and1_lut))
138            (portref I1 (instanceref or_lut))
139            )
140           )
141           (net and2_net (joined
142            (portref O (instanceref and2_lut))
143            (portref I0 (instanceref or_lut))
144            )
145           )
146           (net or_net (joined
147            (portref O (instanceref or_lut))
148            (portref I (instanceref cout_obuf))
149            )
150           )
151           (net xor1_net (joined
152            (portref O (instanceref xor1_lut))
153            (portref I0 (instanceref xor2_lut))
154            (portref I0 (instanceref and2_lut))
155            )
156           )
157           (net xor2_net (joined
158            (portref O (instanceref xor2_lut))
159            (portref I (instanceref s_obuf))
160            )
161           )
162         )
163       )
164     )
165   )
166 (comment "Reference To The Cell Of Highest Level")
167
168   (design netlist_EMPTY
169     (cellref netlist_EMPTY (libraryref hdi_lib_etc))
170     (property part (string "xc7a35tcpg236-3"))
171   )
172 )
```

A.3 XDC Constraints

```
1 set_property PACKAGE_PIN U14 [get_ports a_port]
2 set_property PACKAGE_PIN V14 [get_ports b_port]
3 set_property PACKAGE_PIN V13 [get_ports cin_port]
```

```
4 set_property PACKAGE_PIN U16 [get_ports cout_port]
5 set_property PACKAGE_PIN U15 [get_ports s_port]
6
7 set_property IOSTANDARD LVCMOS18 [all_inputs]
8 set_property IOSTANDARD LVCMOS18 [all_outputs]
```

Appendix B

Random Placer

B.1 Tcl Code

```
proc ::tincr::random_placer { args } {
  set verbose 0
  ::tincr::parse_args {seed} {verbose} {} {} $args

  # Let Vivado place all of the ports in the design
  place_ports -quiet

  # Seed the RNG if a seed was provided
  if {[info exists seed]} {
    expr srand($seed)
  }

  set cells [::tincr::cells get_primitives]

  foreach cell $cells {
    if {$verbose} {
      puts "<[clock format [clock seconds] -format %H:%M:%S]> Placing cell
          $cell..."
    }

    # Skip cells that have already been placed, i.e. IO
    if {[::tincr::cells is_placed $cell]} continue

    set cell_type [get_lib_cells -of_objects $cell]
    if {$cell_type == ""} {
      puts "WARNING: No library cell was found for $cell."
      continue
    }

    # Skip IO cells (the user can let Xilinx's placer handle this stuff)
    if {[get_property NAME $cell_type] == "BUFG" || [get_property NAME
        $cell_type] == "IBUF" || [get_property NAME $cell_type] == "OBUF"} {
      puts "INFO: Skipping cell $cell because it is of type $cell_type."
      continue
    }

    # Skip GND/VCC cells - Vivado must place these
    if {[::tincr::get_name $cell_type] == "VCC" || [::tincr::get_name
        $cell_type] == "GND"} {
      puts "INFO: Skipping cell $cell because it is of type $cell_type."
```

```
38        continue
39      }
40
41      # Get the list of BELs compatible with this cell
42      set bels [::tincr::bels compatible_with $cell]
43
44      set idx [expr int(rand() * [llength $bels])]
45      set watch_dog 0
46
47      # Find a free bel
48      set bel [lindex $bels $idx]
49
50      if {$verbose} {
51        puts "\tCandidate BEL: $bel"
52      }
53
54      # while {[::tincr::cells place $cell $bel] != 0} {}
55      while {[catch {place_cell [::tincr::get_name $cell] [::tincr::get_name
            $bel]} fid] || [::tincr::get_type $bel] == "RAMBFIFO36E1_RAMBFIFO36E1
            "} {
56        incr watch_dog
57        if {$watch_dog >= [llength $bels]} {
58          error "Placement failed."
59        }
60
61        incr idx
62        if {$idx >= [llength $bels]} {
63          set idx 0
64        }
65
66        set bel [lindex $bels $idx]
67        if {$verbose} {puts "\tCandidate BEL: $bel"}
68      }
69    }
70  }
```