2017-07-01

Vivado Design Interface: Enabling CAD-Tool Design for Next Generation Xilinx FPGA Devices

Thomas James Townsend
Brigham Young University

Vivado Design Interface: Enabling CAD-Tool Design for

Next Generation Xilinx FPGA Devices

Thomas James Townsend

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

Master of Science

Department of Electrical and Computer Engineering

Brigham Young University

ABSTRACT

Vivado Design Interface: Enabling CAD-Tool Design for
Next Generation Xilinx FPGA Devices

Thomas James Townsend
Department of Electrical and Computer Engineering, BYU
Master of Science

The popularity of field-programmable gate arrays (FPGA) has grown in recent years due to their potential performance advantages over sequential software, and as a prototyping platform for application-specific integrated circuits (ASIC). Vendors such as Xilinx offer automated tool suites that can be used to program FPGAs based on a RTL description. These tool suites are sufficient for general users, but they usually don't provide the opportunity to integrate custom computer-aided design (CAD) tools into the regular design flow.

Xilinx first offered this capability in their ISE tool suite with the Xilinx Design Language (XDL). Using XDL, a Xilinx design could be extracted from the regular CAD flow, run through an external tool, and injected back into the flow. Research tools targeting commercial FPGAs have most commonly been based on XDL. Vivado (Xilinx's newest tool suite) no longer supports XDL, preventing similar tools from being created for next-generation devices. Instead, Vivado includes a Tcl interface that exposes Xilinx's internal design and device data structures. Considerable challenges still remain to users attempting to leverage this Tcl interface to develop external CAD tools.

This thesis presents the Vivado Design Interface (VDI), a set of file formats and Tcl functions that address the challenges of exporting and importing designs to and from Vivado. To demonstrate its use, VDI has been integrated with RapidSmith2, an external FPGA CAD framework. To the best of our knowledge this work is the first successful attempt to provide an open-source tool-flow that can export designs from Vivado, manipulate them with external CAD tools, and re-import an equivalent representation back into Vivado.

Keywords: Vivado, FPGA, Tcl, CAD, RapidSmith2, Xilinx, Java

ACKNOWLEDGMENTS

I would like to dedicate this thesis to my wife Michelle and daughter Aubrey. Michelle, you constantly inspire me by how hard you work to achieve your own dreams, and give me the extra push to do the same with mine. Aubrey, you are, without a doubt, the light of our lives. Your laugh gave me many enjoyable moments between the late nights spent studying and completing the research of this thesis. It is a great feeling to know that there will always be such love and support at home from both of you, even after a difficult day.

I must also express my thanks to my parents Daniel and Gillian Townsend. Mom and Dad, you have always been supportive of my life decisions and pursuits, and for that I will be forever grateful. You have taught me with love and patience (even through my difficult teenage years), and I will always cherish our wonderful memories together. I must also thank all members of my family and extended family, who have supported me throughout this journey.

Of course, I owe tremendous gratitude to my adviser, Dr. Brent Nelson, who oversaw and guided me as this thesis took shape. Not only did he introduce me to the Configurable Computing Lab three years ago, but he weathered the storm during a project switch after my first year in graduate school. His insight and guidance was very helpful in keeping my mind focused on the most important tasks moving forward, which was integral in the completion of this research.

I would also like to thank all professors in the Electrical and Computer Engineering Department whose courses turned me into a better learner and engineer. This is especially true of Dr. Mike Wirthlin and Dr. Brad Hutchings, whose courses on digital design and embedded systems inspired me to work with FPGAs. I also owe special thanks to Dr. Jeff Goeders who agreed to be on my committee at the last minute, and made several useful suggestions to improve this document.

Finally, I would like to thank all students in the Configurable Computing Lab who I took coursework with or who helped brainstorm ideas for my research. Specifically, I would like to thank Travis Haroldsen, Brad White, Bond Denhalter, and Mark Crossen for their help in making this thesis possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

1.1 Motivation

The popularity of field-programmable gate arrays (FPGA) has grown in recent years due to their potential performance advantages over sequential software, and as a prototyping platform for application specific integrated circuits (ASIC). Several companies fabricate FPGAs, including Xilinx, Intel (Altera), Microsemi, and Lattice. Each of these vendors offer a set of design automation tools to help customers implement hardware designs on their devices. Figure 1.1 shows the typical implementation flow for FPGA-based systems using these automated tools.

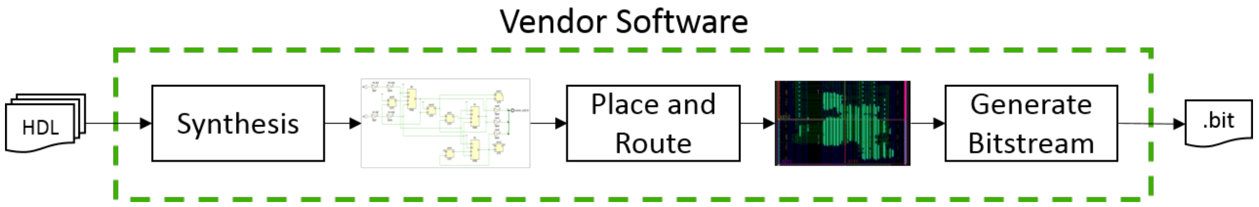


Figure 1.1: Simplified FPGA Design Flow

As the figure shows, the process begins with a hardware circuit that has been expressed using a hardware description language (HDL). Common HDLs include VHDL, Verilog, and SystemVerilog. The HDL description is passed into the FPGA vendor software, where it is synthesized into a device-compatible netlist. The logical netlist is then mapped onto the physical components of the FPGA, and a bitstream (.bit file) is generated that can be loaded into the FPGA's configuration memory. Once the design is loaded into the configuration memory, the FPGA is dynamically configured to implement the corresponding circuit.

Due to the proprietary nature of vendor tools, however, it is difficult to customize the FPGA implementation flow with user created computer-aided design (CAD) tools. For example, it is not possible to create a plug-in for Xilinx's ISE or Vivado tool suite that runs an experimental placer or router instead of the proprietary version. This lack of control limits the type of research that can be done with vendor software, and motivates the use of external, open-source FPGA CAD tools. Over the past few decades, a significant amount of CAD tool research for FPGA-based systems has been pursued outside the confines of vendor tools.

The most common approach to creating external tools has been to use the extremely successful VPR/VTR CAD suite [2] [3]. VTR (verilog-to-routing) offers a complete open-source FPGA CAD flow that includes algorithms for synthesis, technology mapping, packing, placement, routing, and timing and area estimation. Users of VTR can define hypothetical FPGA architectures, and run tools against those architectures to see how they perform. Alternatively, they can modify any stage of the implementation flow (such as the packer, placer, or router) to test and evaluate new CAD ideas and algorithms. VTR has helped lead to many significant contributions in the area of FPGA CAD research, but has had little success in targeting commercial devices.

Targeting commercial FPGA devices has traditionally been more difficult. Most research experiments and open-source CAD tools that aim to do this have been built upon the Xilinx Design Language (XDL), which is capable of interfacing with Xilinx's ISE tool suite [1]. The contributions of projects leveraging XDL have been many. Frameworks such as RapidSmith [5] and Torc [6] provide easy-to-use APIs to modify XDL netlists in a variety of useful ways. [7] and [8] create partial reconfiguration frameworks, capable of swapping logic segments at runtime. [9] and [10] look at ways to decrease the implementation times of FPGA designs and increase designer productivity. [11] and [12] look to create reliable, fault-tolerant FPGA systems. And [13] tries to bridge the gap between the VPR CAD flow with real Xilinx devices. Many significant contributions using XDL have also been made in other areas of FPGA research such as security and debugging.

1.2 Problem Statement and Goals

Clearly, analyzing and modifying commercial FPGA designs outside of vendor tools has proven useful. With the release of Vivado, however, Xilinx discontinued support for XDL, making external tools and frameworks that rely on the interface incompatible with next-generation Xilinx devices (such as UltraScale and Ultrascale+). Vivado instead provides access to its internal design and device data structures through a Tcl application programming interface (API) exposed to the

[1]Tools that target other vendor parts have been created [4], but are less common.

user. The same information that was contained within XDL can now be extracted through Tcl API calls, but there are many challenges associated with using Tcl for this purpose. The first goal of this work is to navigate through the challenges of Vivado's Tcl interface to create a general-purpose way to export design information from Vivado, and import design information back into Vivado (i.e. a XDL replacement). The second goal is to integrate the Vivado XDL replacement with an existing CAD Tool Framework for testing and verification. The third goal is to create an example CAD tool, to demonstrate the possible applications for Vivado CAD tools.

1.3 Contributions

The work described in this thesis is the first successful attempt to provide an open-source tool-flow that can export designs from Vivado, manipulate them with external CAD tools, and re-import valid design representations back into Vivado. The main contributions of this thesis are listed below, and demonstrate why the work is novel:

- It provides insight into the many low-level details of a fully implemented Xilinx design, and how to best represent those details in an external format. It also documents how to successfully reconstruct a design in Vivado.
- It introduces the Vivado Design Interface (VDI), an extension of *Tincr* [14]. VDI uses Vivado Tcl commands to define an organized framework for interfacing with Vivado designs and devices. Specifically, VDI allows users to extract complete design and device information from Vivado to be used with external tools. It also allows users to import modified designs back into Vivado. The performance characteristics of VDI (i.e. how long it takes to import/export designs) are presented.
- It describes the integration of VDI with RapidSmith2 [15], an external CAD tool framework. This integration enables a variety of CAD tools to be written for Vivado designs.
- It demonstrates the correctness of VDI and RapidSmith2 through two methods of verification: (1) Java unit tests which verify that the RapidSmith2 design representation matches the corresponding Vivado design, and (2) On-board hardware tests which verify that designs are still functionally correct after being passed through RapidSmith2 and back into Vivado.

- It presents the implementation and results of a simulated-annealing placer that has been implemented in RapidSmith2.
- It introduces support for UltraScale devices in RapidSmith2. CAD tools targeting UltraScale devices can now be created, which was previously impossible.
- It documents the open-source release of RapidSmith2 at GitHub to allow FPGA CAD researchers across the country and world to develop Vivado CAD algorithms.

One other important contribution to note is that the work presented in this thesis has also been formatted into a conference paper that was accepted into Field Programmable Logic and Applications (FPL) 2017. The title of the FPL paper is "Vivado Design Interface: An Export/Import Capability for Vivado FPGA Designs," and presents many of the same concepts introduced in this thesis.

1.4 Thesis Organization

The remainder of the thesis is outlined as follows. Chapter 2 provides required background for the remainder of the document. This includes an overview of FPGA architecture, Xilinx netlists, Vivado's Tcl interface, Tincr, and the VDI flow. Chapter 3 describes how VDI represents Vivado device information. Chapter 4 describes how VDI represents Vivado design information. Chapter 5 describes the necessary updates to RapidSmith2 in order to support importing and exporting Vivado designs through VDI. Chapter 6 presents the results of several experiments using VDI and RapidSmith2. Finally, chapter 7 concludes the thesis and discusses potential future work.

Unless otherwise stated, tests and experiments in this work were performed on a machine running Windows 7 64-bit with a Core i7-860 processor, 8GB of DDR3 RAM, and 1TB 7200RPM SATA hard disk. Also, this work was done with Vivado version 2016.2. Future versions of Vivado may work with VDI, but they have not yet been tested.

CHAPTER 2. BACKGROUND AND RELATED WORK

As stated in the introduction, the primary goal of this work is to create an interface into Vivado that can be used with external FPGA CAD tools. Specifically, the goal is to enable the flow depicted in Figure 2.1 while determining an appropriate solution for the black box. This chapter provides insight to previous projects that the work presented in this thesis builds upon. It also introduces important terminology. The following sections give a high-level overview of the relevant background information for the reader.

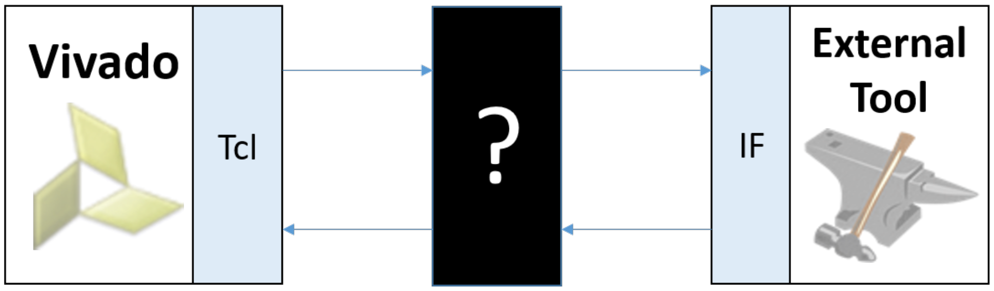


Figure 2.1: External FPGA CAD Flow (the blue boxes represent interfacing software)

Sections 2.1 and 2.2 introduce Xilinx FPGA terminology that is used throughout the document. Section 2.3 discusses Vivado's Tcl interface and why it is not suitable for the creation of FPGA CAD tools. Section 2.4 describes `Tincr`, a Vivado Tcl plugin. `Tincr` was the first research project to explore Vivado design and device extraction. It created a device extraction API and demonstrated a proof-of-concept design extraction which this thesis further explores. Sections 2.5 and 2.6 describe the Xilinx CAD tool flow based on XDL, ISE, and the original RapidSmith CAD framework. Since XDL is no longer supported by Xilinx, this work attempts to create a similar tool flow with Vivado's Tcl interface. It is assumed that the reader has a basic knowledge of FPGAs and the FPGA implementation flow, as those topics are not described here.

2.1 Xilinx FPGA Architecture

Xilinx FPGAs can be broken down into `series`, `families`, and individual `parts`. At the highest level, a series defines a unique FPGA architecture. Vivado currently supports three different series: Series7, UltraScale, and UltraScale+. As shown in Table 2.1, each series can be broken down into a list of families. These families all use the same series architecture, but are optimized for cost, power, performance, size, or another metric.

Table 2.1: Vivado Device Families (organized by series)

Series7	**UltraScale**	**UltraScale+**
Kintex	Kintex	Kintex
Virtex	Virtex	Virtex
Artix		
Spartan		
Zynq		

Families can further be broken down into one or more parts (actual FPGA devices). A Xilinx part has a variety of attributes including its number, package type, and speed-grade. Take the part "xcku025-ffva1156-1-c" as an example. This part is within the Kintex UltraScale family, uses a "ffva1156" package type, and has a speed grade of "1-c". Figure 2.2 shows the device hierarchy of a Xilinx part . The following subsections describe each internal component of a Xilinx FPGA as shown in the figure.

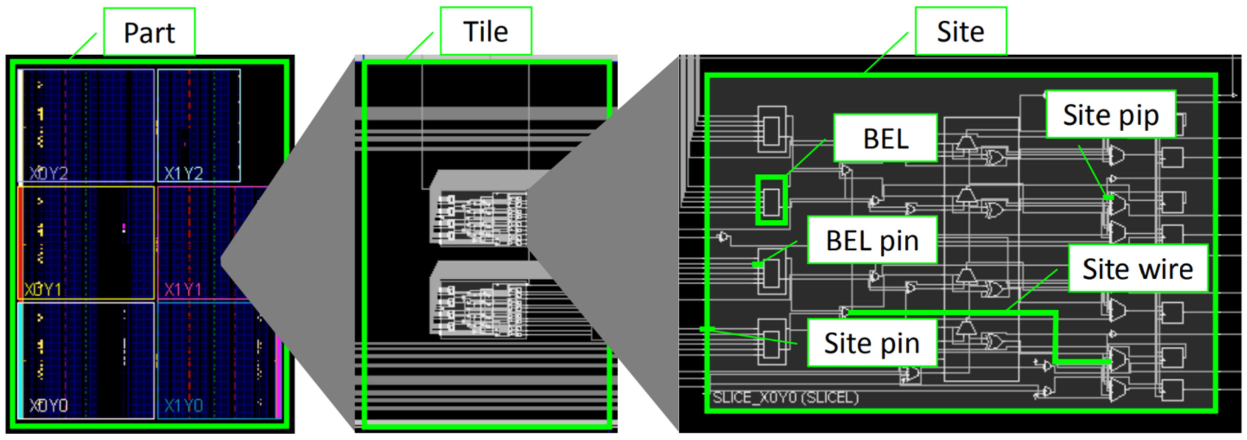


Figure 2.2: Xilinx Device Hierarchy [1]

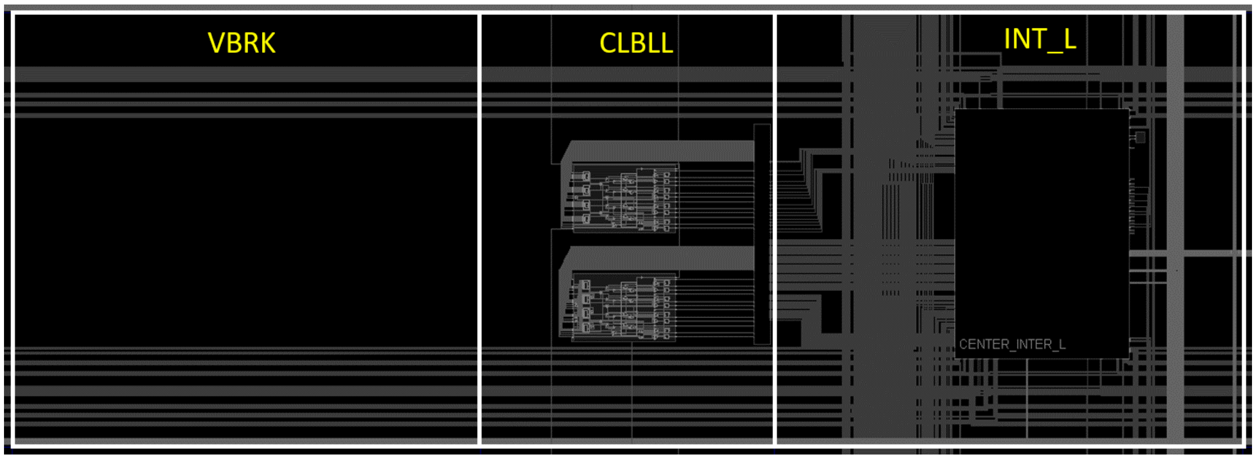


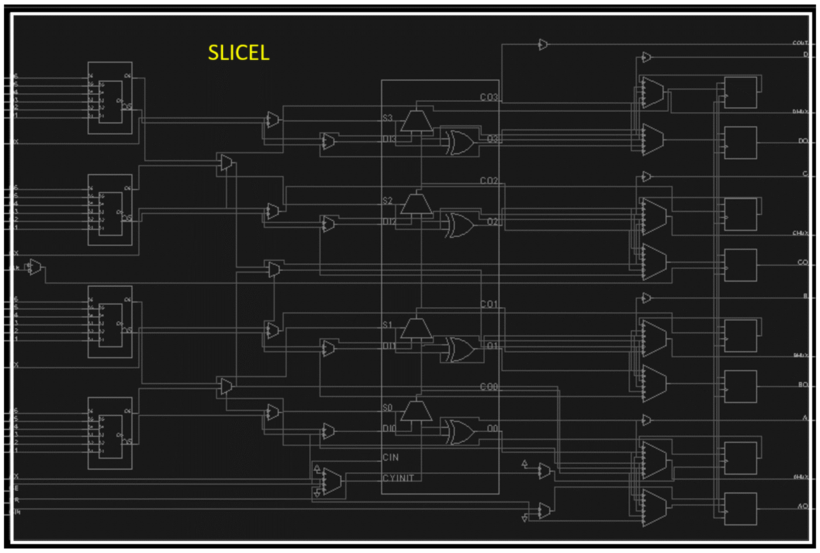
Figure 2.3: Artix7 Tiles

2.1.1 Tiles

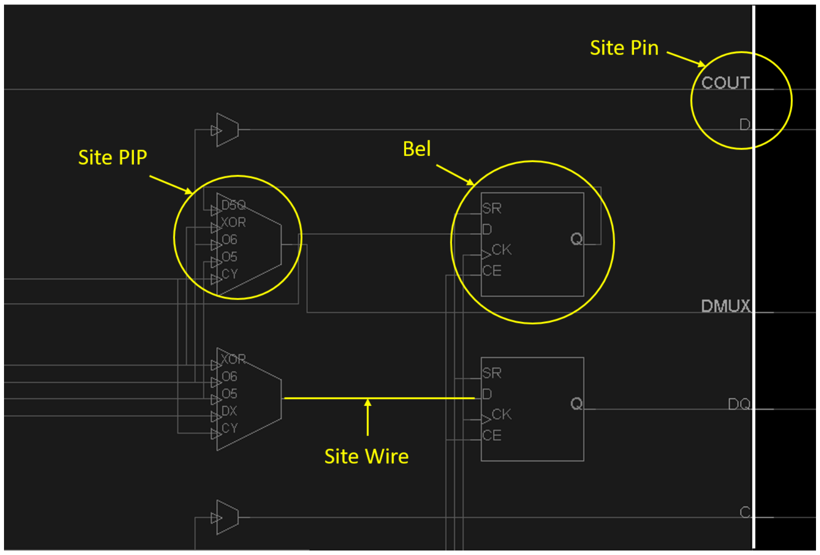
A Xilinx FPGA is organized into a two-dimensional array of `Tiles`. Each tile is a rectangular component of a device that performs a specific function such as implementing digital logic or storing BRAM memory. Tiles are stamped across a device and wired together through the general routing fabric. All copies of a tile type are identical or nearly identical (they may have minor routing differences). Figure 2.3 displays three types of tiles in an Artix7 device. The **VBRK** tile on the left is used for wiring signals between other tiles (these connections are not programmable). The **INT_L** tile on the right is a switchbox tile. These are reconfigurable routing tiles that allow a single wire to be routed to various locations within the FPGA. The **CLBLL** tile in the middle is used to implement combinational and sequential digital logic, and is the fundamental component of Xilinx FPGAs. Other tile types include DSP, BRAM, FIFO, and IOB.

2.1.2 Sites

Tiles generally consist of one or more `Site` objects, which organize the hardware components of the tile into related groups. Specifically, sites are the part of a tile which perform the tile's "useful" function. The remainder of the tile is used to wire signals to and from its corresponding sites. Figure 2.4 shows an example site of type SLICEL within a Series7 CLBLL tile. As the figure shows, a site consists of three main components which are connected through wires:



(a)



(b)

Figure 2.4: Series7 SLICEL Site (a), and Highlighted Site Components (b)

- **Site PIP**s: Also called routing muxes, these are reconfigurable routing PIPs used to specify the internal routing of a site. In Vivado, site PIPs are usually configured automatically as cells in a design are placed (based on cell properties and placement locations).
- **BEL**s: **B**asic **EL**ements are hardware components within a site for implementing digital logic. For example, look-up-tables (LUT) within a SLICEL are used to implement logic

equations, and flip-flops (FF) are used as storage. In a synthesized netlist, design elements are mapped to BELs during implementation.

- **Site Pin**s: These pins are connected to wires of the parent tile and typically drive/receive signals from the general fabric.

2.1.3 Wires and PIPs

FPGA components are connected together using metal `Wires` (called `Nodes` in Vivado). To make the FPGA reconfigurable, wires are connected through programmable interconnect points (PIPs). Individual PIPs can be enabled or disabled as a design is being routed, and a string of enabled PIPs uniquely identify the used wires of a physical route. PIPs are most commonly found in switchbox tiles, and enable a single wire to be routed to several locations on the chip. Figure 2.5

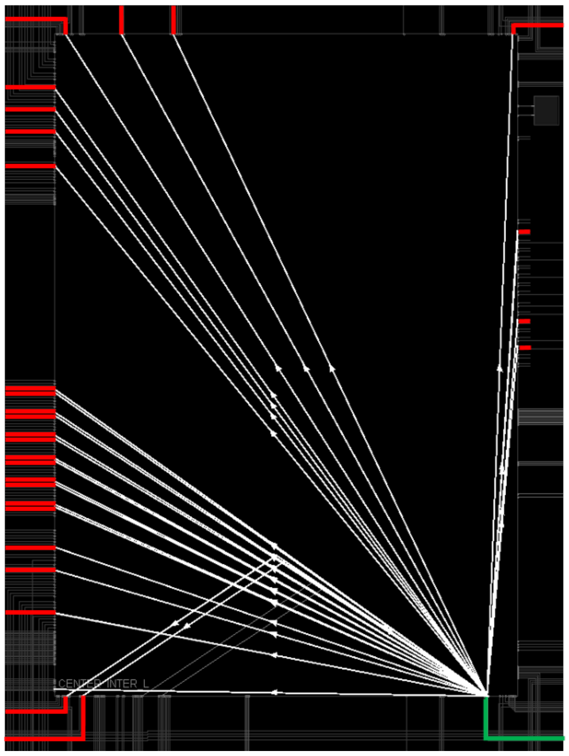


Figure 2.5: An example switchbox tile. The green wire represents a source wire, and the red wires represent all possible sink wires in the switchbox. The highlighed white sections of the figure are PIP connections.

shows an example switchbox with its corresponding PIPs. The red wires represent all downhill nodes that the green wire can connect to through a PIP connection.

2.2 Xilinx Netlist Structure

During the synthesis stage of implementation, a digital circuit expressed using RTL (VHDL or Verilog) is translated to a lower level Xilinx netlist. This netlist describes a digital circuit in terms of primitive elements that can directly target hardware on a Xilinx FPGA. In terms of granularity, a Xilinx netlist is more abstract than gates and transistors, but more detailed than RTL. A list of valid primitives that can be used within a Xilinx netlist can be found at [16] for Series7 devices and [17] for Ultrascale devices. The primitives of a Xilinx netlist are wired together to create a digital circuit capable of being implemented on a FPGA. Figure 2.6 shows an example netlist for a 3-bit counter created in Vivado.

As the figure shows, Vivado netlists are composed of three primary components: `Cells`, `Nets`, and `Ports`. Cells are **instances** of Xilinx primitives. They are the basic building blocks of a Xilinx netlist and implement the actual logic of a digital design. The most commonly used cells include:

- **Look Up Tables** (LUTs): Implement logic equations such as $O6 = (A1 + A2) \oplus A3$.

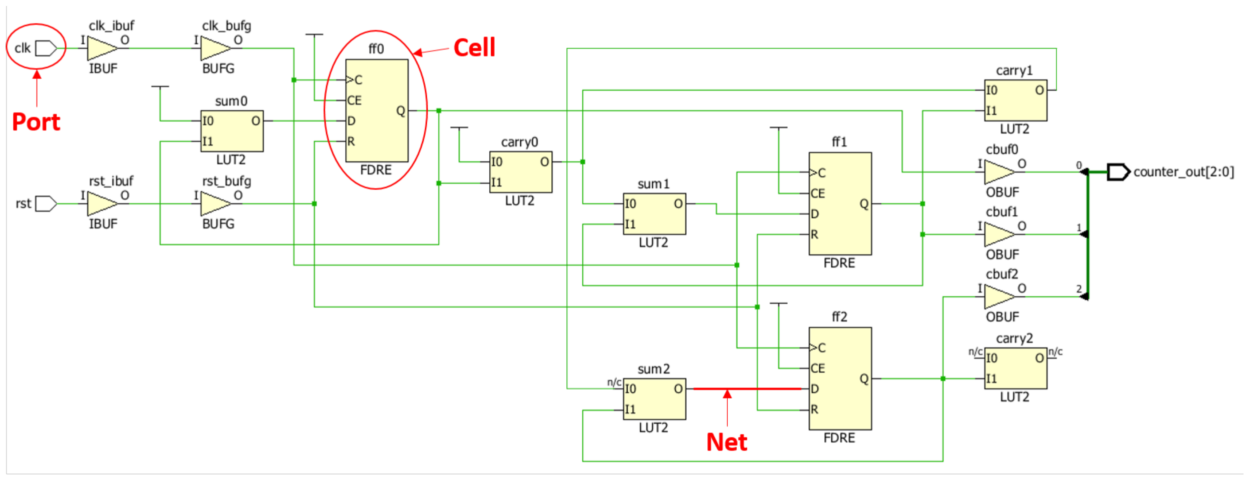


Figure 2.6: Schematic of a 3-bit counter in Vivado using LUT and FDRE cells. The yellow boxes are cells, the green lines are nets, and and the white figures on the edge of the diagram are ports.

- **Flip-Flops** (FDxx): Single-bit storage elements. Figure 2.6 uses a FDRE cell which specifies a rising-edge flip-flop with a reset port, but ties the clock enable port high. Other types of FDxx cells can be used to include a clock enable port.
- **Block Ram** (BRAMs): On-chip FPGA memory.
- **Digital Signal Processing Units** (DSPs): Perform complex arithmetic functions efficiently.
- **Buffers** (BUF): IO, clock, and other types of signal buffers.

Several other types of cells can be used, but the ones in the list above are the most common. Nets connect cells together. In other words, the output of one cell is wired to the input of another cell using a net. Ports are simply design input/output (IO). In terms of a FPGA design, ports are mapped to specific peripheral pins of the FPGA for chip IO.

It is important to note that a Xilinx netlist is purely logical. There is no physical information within the netlist (i.e. there is no information about where the cells have been placed, or how the nets have been routed). When exporting a design from Vivado, the Xilinx netlist representation is converted to an electronic design interchange format (EDIF).

2.3 Vivado Tool Suite

In recent years, Xilinx released their new vendor tool for programming FPGAs: Vivado. Vivado supersedes ISE (the previous tool), and is the only tool suite that supports the latest Xilinx families such as UltraScale. The most significant change with Vivado is the introduction of a Tcl interface. Using Tcl commands, users of Vivado can write Tcl code to script design flows, set constraints on a design, and perform low-level design modifications. There are Tcl commands for a variety of useful functions including: finding all tiles in a device, getting all of the used PIPs in a routed design, and grouping related cells into a macro. For example, the Tcl command `[get_sites -filter (SITE_TYPE==SLICEL && !IS_USED)]` will return a list of all unused SLICEL sites in the current device. This list could potentially be used to add additional logic to a design post-route. The Tcl interface is a powerful addition to the Xilinx tool suite, but suffers some major drawbacks:

- Tcl, being an interpreted language, is slow. Compiled or managed runtime systems are better options for performance.

- Vivado's Tcl interface does not manage memory well. A simple Tcl script can cause the memory usage of Vivado to grow indefinitely.
- Tcl does not natively support higher-level programming constructs, making it more difficult to implement complex algorithms (such as PathFinder).

These drawbacks largely motivate using the Tcl interface as an *extraction* tool, instead of a CAD framework itself. Vivado Tcl scripts are the basis for much of the work found in this thesis.

2.4 Tincr

`Tincr` [14] is a Tcl plugin to Vivado created by Brad White. It introduces two useful packages: `TincrCAD` and `TincrIO`. These packages augment Vivado's native Tcl interface with a set of high-level commands that (a) simplify a variety of Vivado tasks and (b) add additional functionality to the Tcl interface in the form of new Tcl commands. `TincrCAD` focuses on commands for implementing CAD tools directly in Vivado, and `TincrIO` focuses on commands for extracting device and design data from Vivado. The work in this thesis builds upon the initial work done with `TincrIO`, which demonstrated the plausibility of manipulating Vivado designs outside of Vivado.

Two aspects of `TincrIO` were particularly useful. The first is `[tincr::write_xdlrc]`, a Tcl command capable of creating a complete XDLRC device file (described in the next section) for any Vivado device. Series7 XDLRC files were exhaustively tested against their ISE counterpart to verify that the command generated valid and complete device files. However, UltraScale and UltraScale+ XDLRC files generated from this command were incomplete and not well tested. Chapter 3 describes the required modifications to `[tincr::write_xdlrc]` to support UltraScale devices.

The second useful aspect of `TincrIO` is the proposed `Tincr` Checkpoint (TCP) format. In the original distribution of `Tincr`, a directory full of XDC constraint files was used to externally represent a Vivado design. The authors of `Tincr` demonstrated that by using these checkpoints, a FPGA design could be theoretically exported from Vivado in an open-source format, and then reimported back into Vivado. These checkpoints, however, were simply a *proof-of-concept*. Several important aspects of a design (such as BEL routethroughs and an accurate depiction of routing) can not be represented in the original TCP specification. Chapter 4 describes an improved checkpoint format, which creates an accurate external design representation that encapsulated *all* parts of a design.

2.5 Xilinx Design Language (XDL)

The Xilinx Design Language is a command line interface into Xilinx's ISE tool suite. Using a single command, `xdl`, both device and design information can be extracted from ISE for external use. Device specific information is exported via XDLRC files, which contain a detailed listing of all the physical components inside of a Xilinx FPGA. This includes the tiles, sites, BELs, and routing resources (without timing information) available in the specified part. XDLRC files are very verbose (with the largest devices being up to 100GB), and so they need to be compressed before they can practically be used with external CAD tools. Listing 2.1 shows the general format for a tile within a XDLRC file. As can be seen, the tile's sites, wires, and PIP connections are all listed. To learn more about XDLRC files (such as the meaning of each specific token), review [18].

Listing 2.1: XDLRC Tile Format

```
(tile 1 14 CLB_X6Y63 CLB 4
    // List of sites in the tile and wires that connect to the site pins
    (primitive_site SLICEL_X9Y127 SLICEL internal 27
        (pinwire BX input BX_PINWIRE3)
        ...
    )
    ...
    // List of wires in the tile and their connections
    (wire E2BEG0 5
        (conn CLB_X7Y63 CLB_E2BEG0)
        (conn INT_X8Y63 E2MID0)
        ...
    )
    ...
    // List of pips in the tile
    (pip INT_X7Y63 BEST_LOGIC_OUTS0 -> BYP_INT_B5)
    ...
  (tile_summary CLB_X6Y63 CLB 122 403 148)
 )
```

Design data is exported via XDL files. XDL files are a combination of the logical portion of a design (a netlist), as well as placement and routing information for the design. A XDL netlist is organized into **instances**. Each instance represents a site on the corresponding device, with attribute strings describing how the site is internally configured (i.e. what BELs are being used and how they are connected). In a placed design, instances are assigned a corresponding site value. Listing 2.2 gives an example of a XDL instance targeting a SLICEL site.

Listing 2.2: XDL Instance

```
inst "instanceName" "SLICEL", placed CLB_X14Y4 SLICE_X23Y8 ,
  cfg " BXINV::#OFF BYINV::#OFF CEINV::#OFF CLKINV::#OFF COUTUSED::#OFF
  CY0F::#OFF CY0G::#OFF CYINIT::#OFF DXMUX::#OFF DYMUX::#OFF F::#OFF
  F5USED::#OFF FFX::#OFF FFX_INIT_ATTR::#OFF FFX_SR_ATTR::#OFF FFY::#OFF
  FFY_INIT_ATTR::#OFF FFY_SR_ATTR::#OFF FXMUX::#OFF FXUSED::#OFF
  G:DCM_AUTOCALIBRATION_DCM_clock/DCM_clock/md/RSTOUT1:#LUT:D=A1
  _BEL_PROP::G:LIT_NON_USER_LOGIC:DCM_STANDBY GYMUX::#OFF REVUSED::#OFF
  SRINV::#OFF SYNC_ATTR::#OFF XBUSED::#OFF XMUXUSED::#OFF XUSED::#OFF
  YBUSED::#OFF YMUXUSED::#OFF YUSED::0 "
  ;
```

Routing is represented using XDL **net**s. XDL nets contain a list of connected instance pins (i.e. site pins), and a list of PIPs used to physically connect the pins together. An example net is shown in Listing 2.3. XDL provided a means for representing fully placed and routed designs from ISE.

Listing 2.3: XDL Net

```
net "netName" ,
 outpin "instanceNameOfSourcePin" Y ,
 inpin "instanceNameOfSinkPin" RST ,
 pip CLB_X14Y4 Y_PINWIRE1 -> BEST_LOGIC_OUTS5_INT ,
 pip DCM_BOT_X15Y4 SR_B0_INT3 -> DCM_ADV_RST ,
 pip INT_X14Y4 BEST_LOGIC_OUTS5 -> OMUX8 ,
 pip INT_X15Y5 OMUX_EN8 -> N2BEG0 ,
 pip INT_X15Y7 N2END0 -> SR_B0 ,
 ;
```

The combination of XDLRC and XDL files make it possible to perform design manipulations outside the confines of vendor tools. It has been a great resource to the FPGA CAD research community. There are, however, a few problems with XDL. The first is that, as Listing 2.2 shows, XDL instances are largely black boxes. They don't explicitly show the BELs and wires that are being used *inside* of a site. This makes it difficult to write CAD tools that need access to this information (such as a packer). The second, and more challenging, issue is that with Vivado Xilinx discontinued support for XDL. CAD tools and frameworks built on XDL will not be compatible with next-generation devices. As previously stated, the primary objective of this thesis is to create a XDL-alternative for Vivado designs so that CAD development can continue to happen for future Xilinx devices. The interested reader is referred to [18] for a complete guide to XDL.

2.6 RapidSmith

The original RapidSmith [5] is a FPGA CAD Tool framework for Xilinx FPGAs. Written in Java, it offers a rich set of APIs, data structures, and useful functions to analyze and manipulate XDL netlists. RapidSmith was developed by Chris Lavin, who initially used it to reduce FPGA

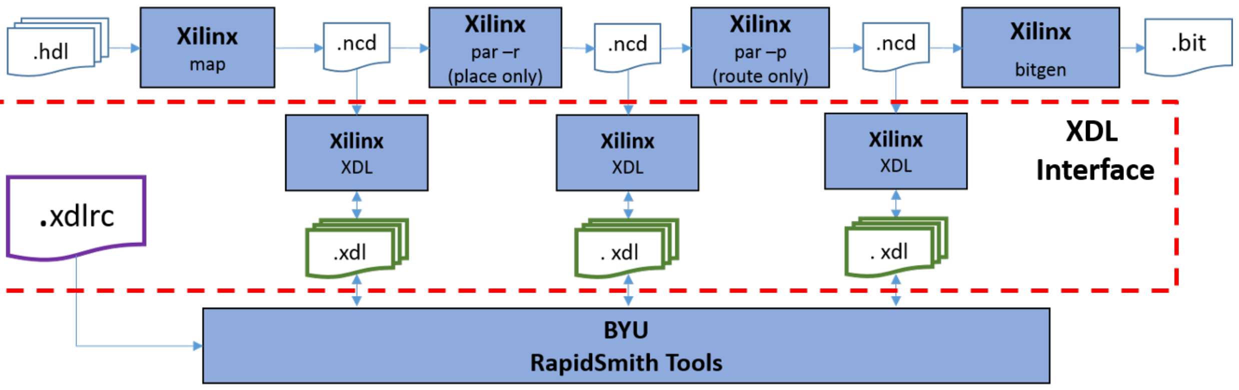


Figure 2.7: Original RapidSmith Tool Flow Using XDL

compilation times by creating a hard macro design flow. It has since grown to be the basis of several research projects and applications (many of which are listed in chapter 1). RapidSmith has been especially popular in the area of FPGA security and reliability. Figure 2.7 shows the available design flows using RapidSmith.

Users of RapidSmith can write CAD tools that operate on a Xilinx design post-synthesis, post-place, or post-route. The modified design can then be re-imported back into ISE to complete the remainder of the implementation flow. To handle very large XDLRC files, RapidSmith compresses them into compact device files. Once compressed, the RapidSmith device files are MegaBytes in size as opposed to GigaBytes. RapidSmith only supports designs from Xilinx's ISE tool suite, and can be downloaded at `http://rapidsmith.sourceforge.net`.

2.6.1 RapidSmith2

The RapidSmith framework has helped researchers create several different CAD tools targeting Xilinx devices. Due to the format of XDL however, these CAD tools always operate at site boundaries. This makes it difficult to explore sub-site CAD algorithms that work explicitly

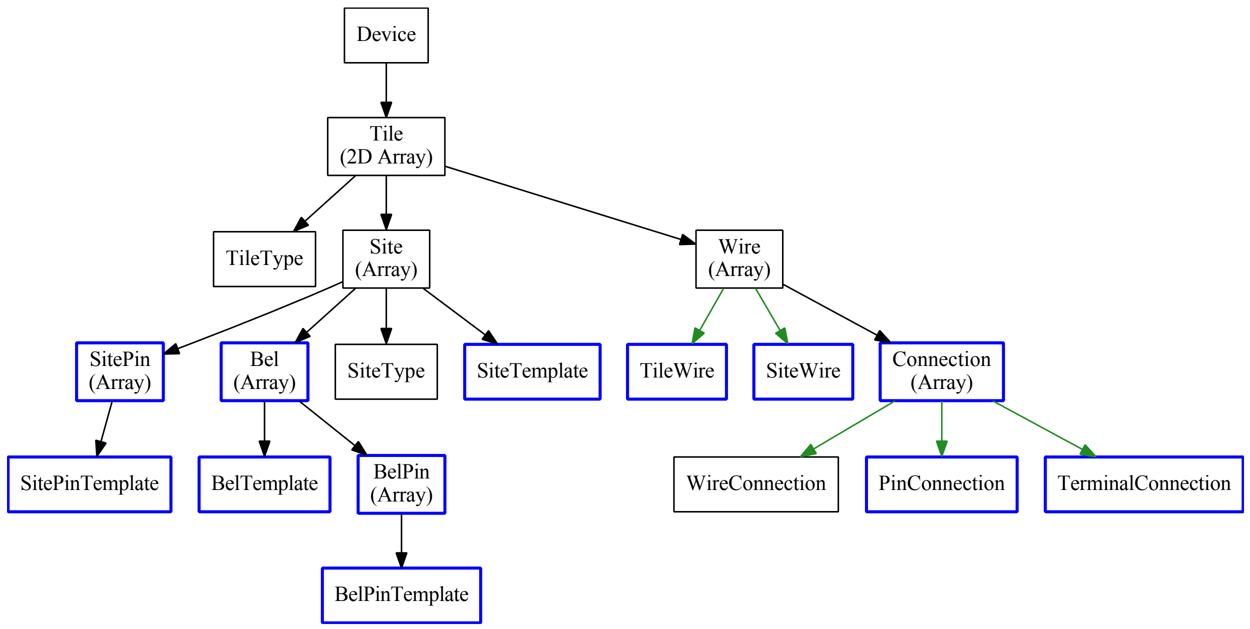


Figure 2.8: RapidSmith2 Device data structure tree. Classes and interfaces bolded in blue are sub-site components new to RapidSmith2.

with sub-site components (i.e. BELs, site PIPs, etc.) such as packers. Travis Haroldsen was interested in exploring packing algorithms for Virtex6 devices [19], and so created RapidSmith2 [15]. RapidSmith2 updated the internal data structures of RapidSmith to the BEL and cell-level, allowing algorithms to have finer grained control over sub-site placement and routing. Because it was originally targeting the Virtex6 architecture, RapidSmith2 took as input a XDL netlist, and **unpacks** the netlist to its corresponding cells, nets, BELs, and wires. On design export, the sub-site data structures are **packed** back into a XDL netlist representation. Figure 2.8 shows the device data structure differences between RapidSmith and RapidSmith2. It is important to note that the initial version of RapidSmith2 still did not support Vivado designs. One of the main contributions of this thesis is to recreate the flow of Figure 2.7 using RapidSmith2 and Vivado. Since the RapidSmith2 data structures closely match those of Vivado, the work presented in chapter 5 builds upon the initial RapidSmith2 work.

2.7 Vivado Design Interface

Section 2.3 shows that Vivado's Tcl interface is not suitable for complex CAD tool development (and it was never intended for that). External tools, in languages such as C, C++, and Java, are better options for FPGA CAD researchers looking to implement new ideas and algorithms. In the past, these tools were built around Xilinx's XDL (as shown with the RapidSmith CAD framework). Because XDL is no longer supported in Vivado, a different interface to enable similar tools in the future is required.

This thesis presents the **Vivado Design Interface** (VDI), the proposed XDL-alternative to support Vivado design import and export. Figure 2.9 shows the individual components of VDI, and how they interact with external CAD tools or frameworks. VDI is included with `Tincr`, and is available at `https://github.com/byuccl/tincr`. It is a significant contribution for two reasons in particular:

1. VDI defines a set of file formats used to externally represent Vivado devices and designs in a general, open-source way.

2. VDI includes Vivado Tcl code to parse and generate design and device files.

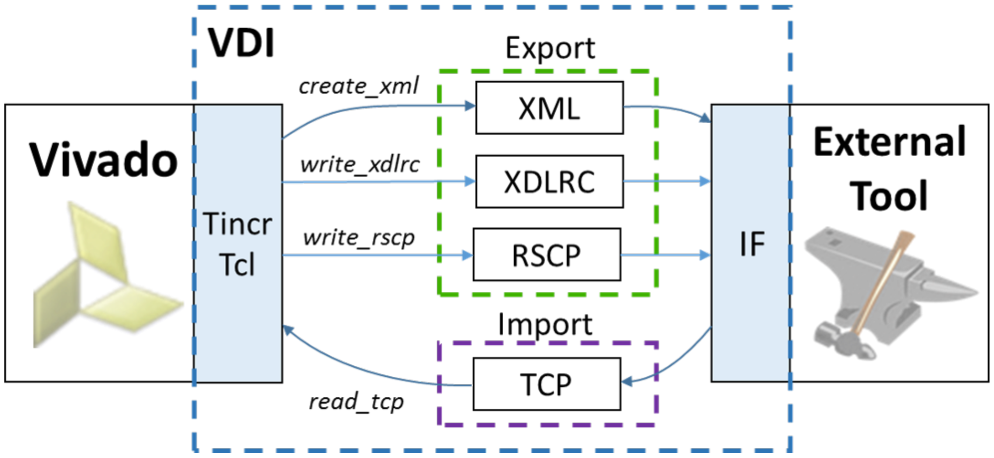


Figure 2.9: Components of the Vivado Design Interface (VDI)

VDI is meant to serve as a general-purpose interface into the Vivado design suite which can be used with *any* CAD tool or framework. Devices are represented with a XDLRC file and a set of XML files (described more in chapter 3). Designs are represented with RSCP and TCP checkpoint files (described more in chapter 4). The remaining chapters in this thesis explore specific aspects of VDI, and how they can be used to create Vivado CAD tools.

CHAPTER 3. VDI: DEVICE EXPORT

All CAD tools targeting commercial FPGAs are built upon an external representation of the vendor's device. A placement algorithm, for example, needs to understand all valid placement locations for cells in a netlist. Similarly, a routing algorithm needs to understand all available wires and connections that can be taken when routing a net. The first goal of VDI is to create an open-source fileset capable of fully representing any Xilinx FPGA supported in Vivado. Figure 3.1 shows the proposed set of files for VDI device export, along with the newly implemented `Tincr` functions to generate each. There are four required VDI files to accurately represent Vivado FPGAs, which are explored throughout the remainder of this chapter.

1. XDLRC
2. Family Info XML
3. Device Info XML
4. Cell Library XML

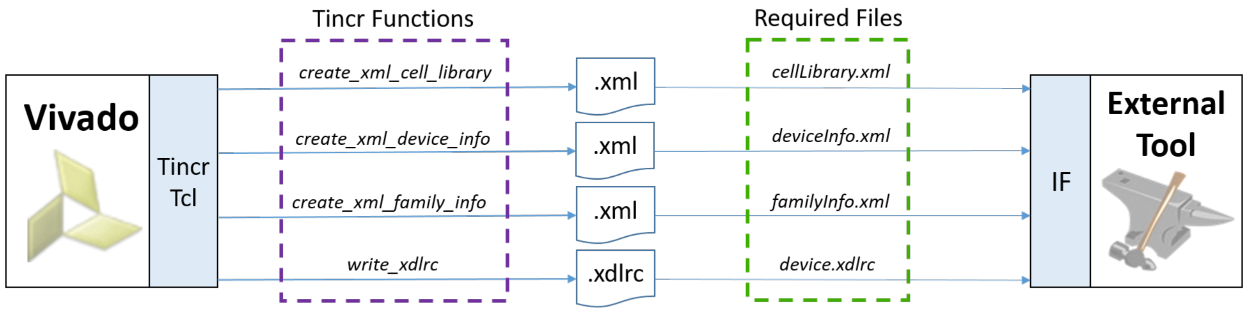


Figure 3.1: VDI Device Files

3.1 XDLRC

XDLRC files are the main components of VDI device export. As described in section 2.5, they contain a detailed listing of all the physical components inside of a Xilinx FPGA. The original `Tincr` distribution included a command, `[tincr::write_xdlrc]`, capable of creating XDLRC files for Series7 devices. This command, however, was only able to create partial device files for UltraScale and later families. **VDI supports creating complete UltraScale and UltraScale+ XDLRC files**. The following subsections describe the required updates to `Tincr` to support UltraScale XDLRC files.

3.1.1 Site PAD Names

In Series7 XDLRC files, PAD sites (that connect to an external pin on the FPGA) are named according to the corresponding package pin of the site. An example is shown in Figure 3.2 for a site of type IOB33M. For the site shown in the figure, the actual site name is "IOB_X0Y68," but the site name reported in the XDLRC is "V15," the name of the package pin in the upper left corner.

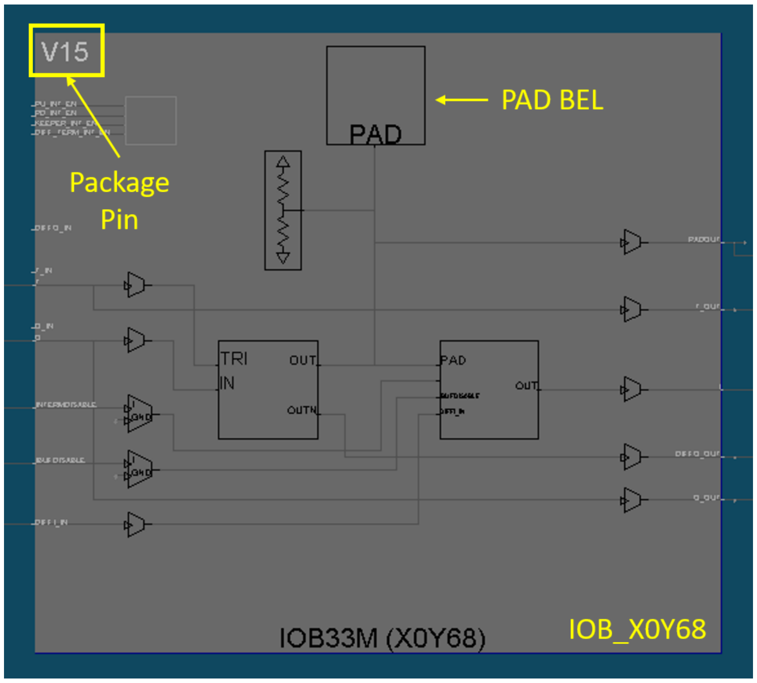


Figure 3.2: IOB Site in Vivado's Device Browser

This naming convention assumes that there is no more than one package pin per site. UltraScale devices, however, break that assumption. There are several sites that contain more than one package pin: SYSMONE1, GTHE3_CHANNEL, GTHE3_COMMON, GTYE3_COMMON, and GTYE3_CHANNEL. Figure 3.3 shows a SYSMONE1 site with two package pins.

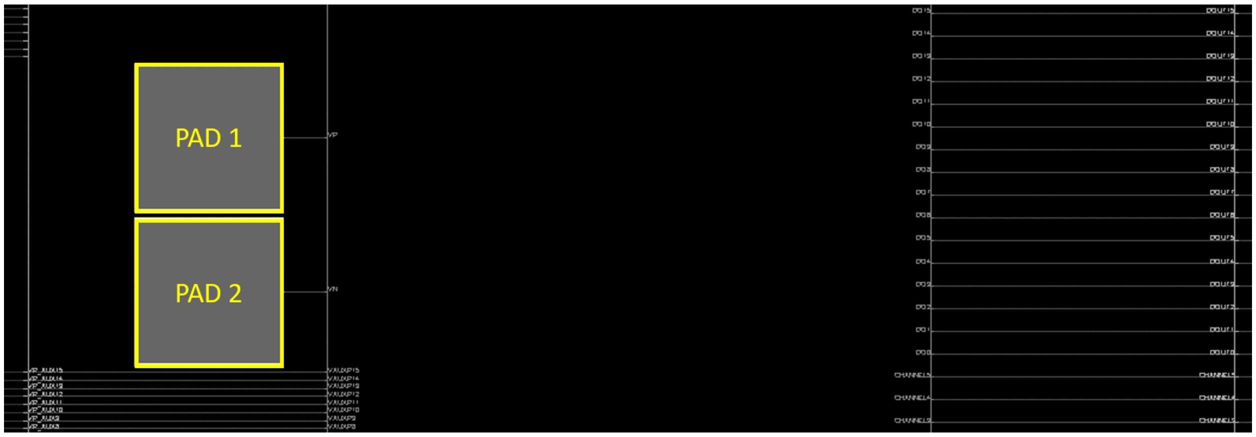


Figure 3.3: UltraScale SYSMONE1 Package Pins

Because UltraScale and later devices have multiple package pins per site, the old naming convention for ISE XDLRC files is no longer valid. Therefore, `[tincr::write_xdlrc]` now uses the actual name of PAD sites (instead of the package pin name) for UltraScale devices.

3.1.2 Floating VCC/GND Sources

Xilinx's Series7 architecture represents power (VCC) and ground (GND) signal sources with TIEOFF sites (as shown in Figure 3.4). To route a GND or VCC net, external tools need to simply start routing at the pin of a TIEOFF. UltraScale devices change the structure of VCC and GND sources by including "floating" VCC and GND BELs instead of TIEOFFs. The term floating refers to the fact that the VCC or GND source is not within a primitive site, but rather within the tile itself. This breaks the well-defined device hierarchy described in section 2.1 where tiles

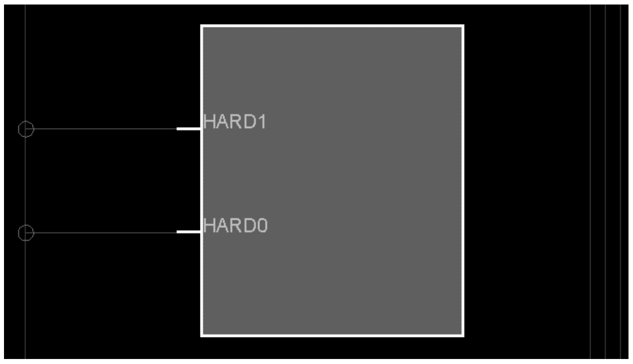


Figure 3.4: Series7 TIEOFF

contain one or more sites and sites contain one or more BELs. To preserve the device hierarchy in UltraScale devices, [tincr::write_xdlrc] transforms each floating BEL into a VCC or GND primitive site as shown in Figure 3.5. VCC and GND BELs can be identified in Vivado by looking for wires in the device whose name starts with either "VCC_WIRE" or "GND_WIRE" respectively. Listing 3.1 shows the corresponding primitive definition that is added to the XDLRC file for GND sites. A similar primitive definition is included for VCC.

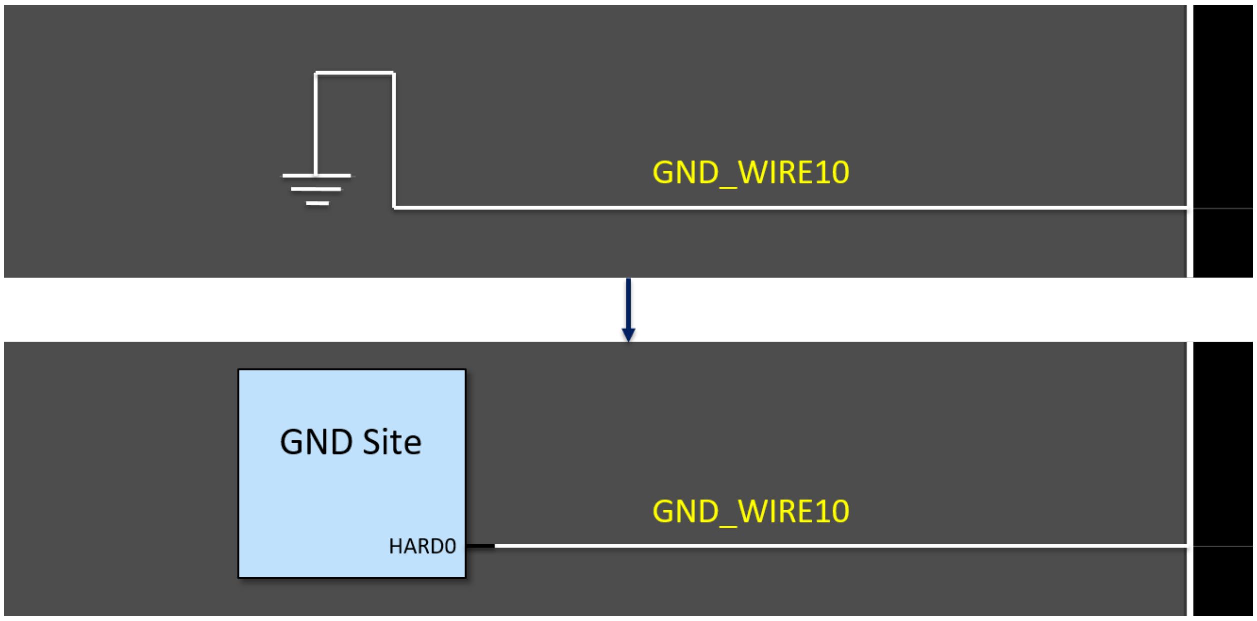


Figure 3.5: UltraScale floating GND BEL (top), and the GND primitive site replacement for XDLRC files (bottom).

Listing 3.1: GND Primitive Definition

```
(primitive_def GND 1 2
   (pin HARD0 HARD0 output)
   (element HARD0 1
      (pin HARD0 input)
      (conn HARD0 HARD0 <== HARD0GND G)
   )
   (element HARD0GND 1 # BEL
      (pin G output)
      (conn HARD0GND G ==> HARD0 HARD0)
    )
  )
```

3.1.3 Vivado Subsite Routing Tool

Vivado's Tcl interface provides several `[get_*]` methods which can be used to access a variety of FPGA components. Three examples used to build XDLRC files are `[get_tiles]`, `[get_sites]`, and `[get_site_pips -of $site]`. Vivado device extraction can be largely automated due to these commands. However, there is no such function to obtain the site wires in a device. The Tcl command `[get_site_wires]` is part of the internal Xilinx namespace, meaning it is not visible to regular users. This primarily affects the **primitive definition** section of the XDLRC file, where the internal structure of each site type is defined (primitive definitions are discussed in Appendix B). Without access to site wire information, the internal routing network of sites cannot be determined automatically. The original `Tincr` distribution used primitive definitions from ISE to create complete Series7 device files, but this is not possible for UltraScale and later devices.

Site wires are visible in the Vivado device browser, making it theoretically possible to create the intrasite connections manually. The Vivado Subsite Routing Tool (VSRT), shown in Figure 3.6, is a GUI application developed as part of this work that helps users accomplish this task. Using VSRT a user can bring up a primitive site in both Vivado's device browser and the VSRT GUI and manually draw the site wires. Once the connections have been drawn, VSRT will generate the required primitive definition file automatically, which can be integrated with the corresponding XDLRC. It is important to note that this time-consuming process only needs to happen once for each device family (usually once per series), and many of the sites can be done automatically with little to no manual intervention.

The most significant contribution involving VSRT is a complete set of primitive definitions for UltraScale and UltraScale+ families. These primitive definitions have been added to `Tincr`, enabling *complete* XDLRC generation for Ultrascale and UltraScale+ devices using `[ tincr::write_xdlrc]`. When future Vivado families are released, VSRT can be used to generate

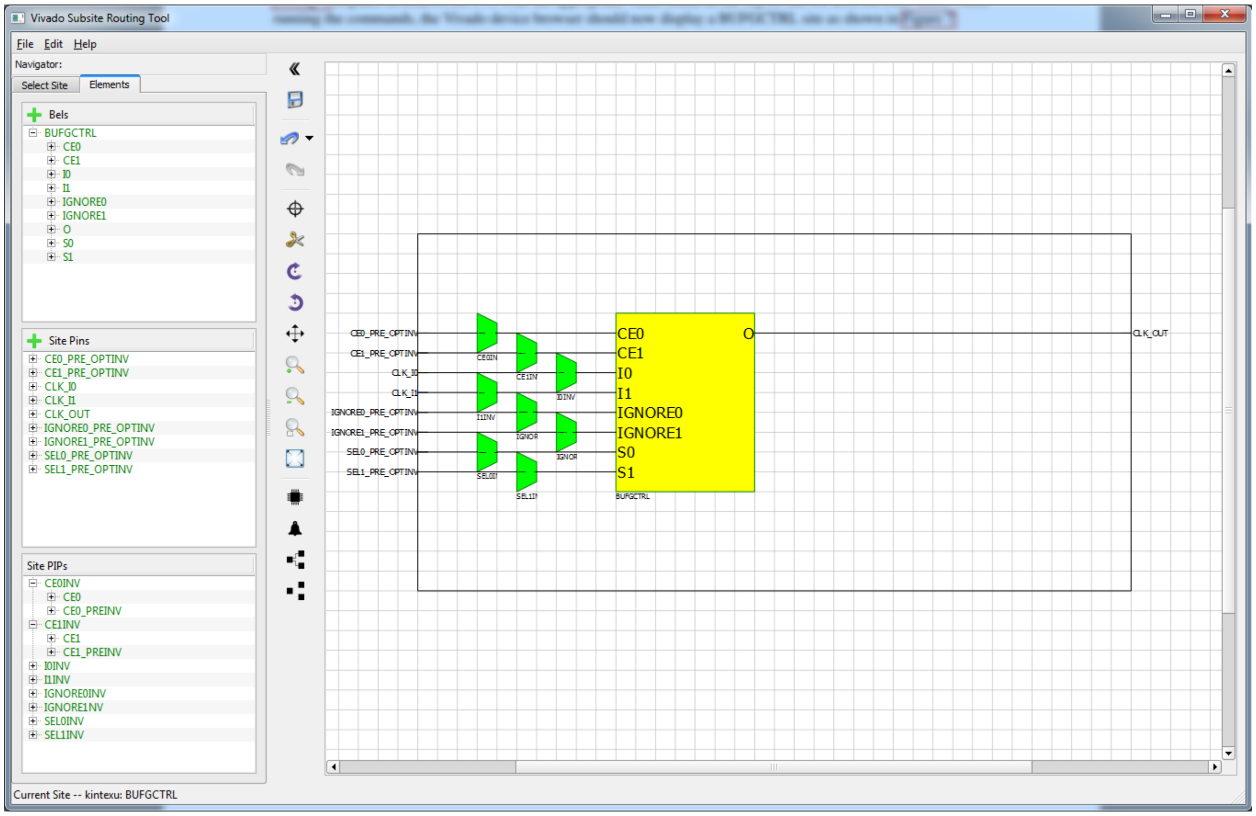


Figure 3.6: Vivado Subsite Routing Tool GUI.

primitive definitions for those families as well. A very detailed user guide for VSRT is included in Appendix B.

3.2 Family Info XML

A *familyInfo.xml* file contains useful information that is not present in the XDLRC files for a given family of devices. Specifically, it includes the following additional information about each site type in a family:

- **Alternate Types**
- **Compatible Types**
- **BEL Routethroughs**
- **Site PIP Corrections**
- **Pin Direction Corrections**

As the name suggests, only one *familyInfo.xml* is required for each of the supported Vivado families listed in Table 2.1 (all devices within a family share the same family info). A new `Tincr` command has been added to generate family info files: `[tincr::create_xml_family_info]`. Using this command, with a few required hand edits, **a complete set of family info XML files have been created for all Series7 and UltraScale families**. The following subsections describe (a) how the Vivado Tcl interface is used to generate a family info file (and the necessary hand edits for Series7 devices), and (b) a description of each of the items in the above list, and why they may be useful for external CAD tools.

3.2.1 Algorithm

The pseudocode for the family info generation script in `Tincr` is given in algorithm 1.

Algorithm 1 Family Info Generation

```
 1: procedure CREATE_XML_FAMILY_INFO($family$, $fileout$)
 2:     $unique\_devices \leftarrow$ get_unique_devices($family$)        ▷ Ignore device speedgrade
 3:     $processed\_types \leftarrow \emptyset$        ▷ Create an empty set of site types
 4:     for each $device \in unique\_devices$ do
 5:         open_device_in_vivado($device$)
 6:         $unique\_sites \leftarrow$ get_unique_sites($device$)
 7:         for each $site \in unique\_sites$ do
 8:             $site\_type \leftarrow$ get_type($site$)
 9:             if $site\_type \notin processed\_types$ then
10:                 print_xml($site$, $fileout$)
11:                 $processed\_types \leftarrow processed\_types \cup site\_type$
12:             end if
13:         end for
14:         close_device_in_vivado($device$)
15:     end for
16: end procedure
```

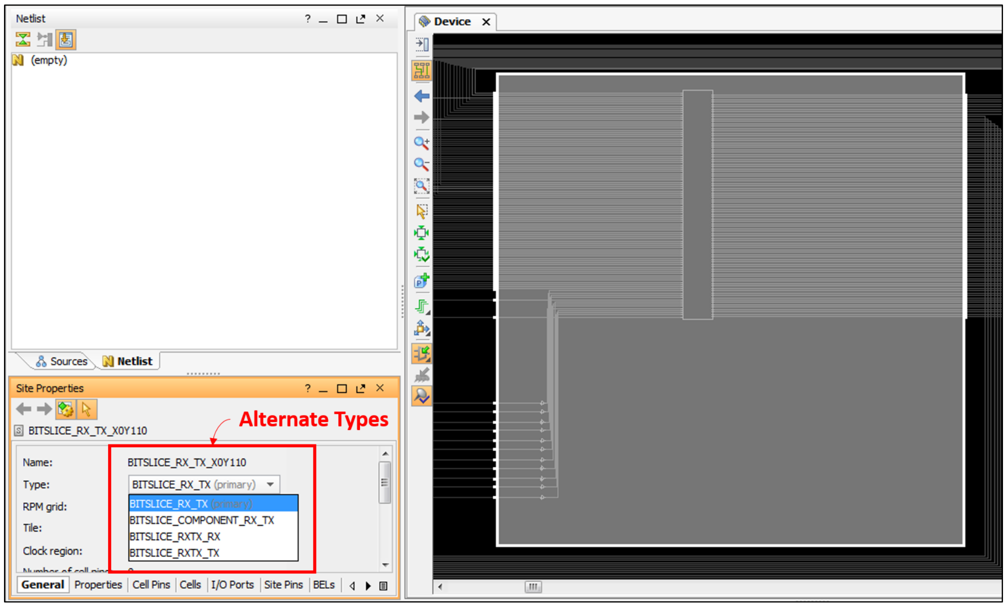
As can be seen, the first step of the algorithm is to find a set of unique devices for the specified family. A device is considered unique based on its part number and package type only (i.e. the speed grade is ignored). For example, the two parts `xcku025-ffva1156-1-c` and `xcku025-ffva1156-2-c` are **not** considered unique, and only one would be used during the generation process. The Tcl regular expression

```
^(x[a-z0-9]+(?:-[a-z0-9]+)?)-.+
```

is used to remove the speed-grade from a complete part name when filtering for unique devices. Once a set of unique devices is created, each device is loaded into Vivado where a list of site types in the device is determined. Since all parts across a family are identical, every site type in a family is only processed one time. Using Vivado Tcl commands, an instance of each site type is analyzed to determine the relevant information for the family info file, which is formatted into XML. It is important to note that the site type must be manually set for alternate-only types before processing the site.

3.2.2 Alternate Types

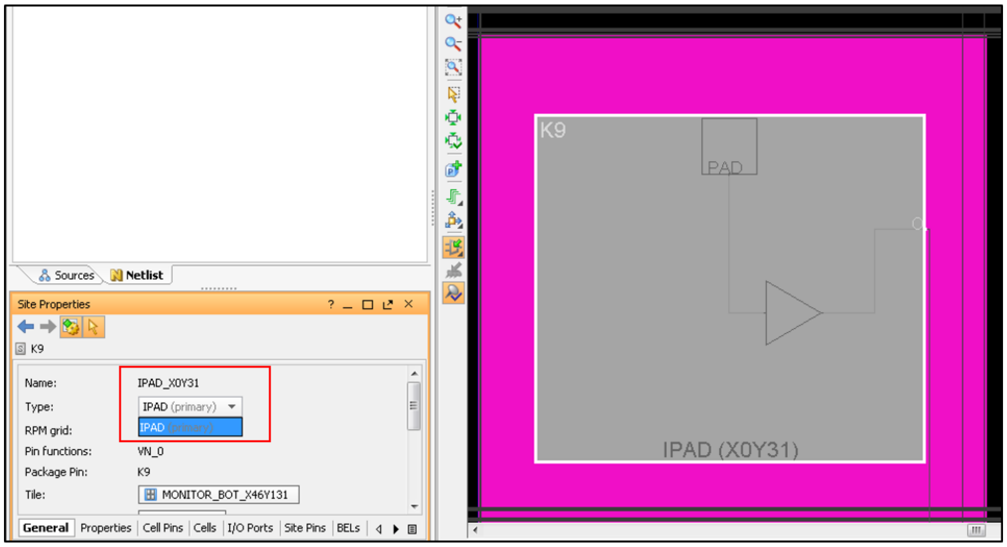
Each physical site in a device has an associated default type. Some sites, however, can be configured to be one of many types (called alternate types in Vivado). Alternate type information is required for external CAD tools because the type of a site can be changed during the placement phase of implementation. To accurately represent a placed design in an external tool, site types need to be changeable. An example of alternate types for an UltraScale BITSLICE_RX_TX site is shown in Figure 3.7. As the figure shows, a BITSLICE_RX_TX site can also be configured to be of type BITSLICE_COMPONENT_RX_TX, BITSLICE_RXTX_RX, or BITSLICE_RXTX_TX.



BITSLICE_RX_TX_X0Y110

Name: BITSLICE_RX_TX_X0Y110
Type: BITSLICE_RX_TX (primary)
RPM grid:
Tile:
Clock region:

Figure 3.7: BITSLICE_RX_TX Alternate Types



Name: IPAD_X0Y31
Type: IPAD (primary)
IPAD (primary)
RPM grid:
Pin functions: VN_0
Package Pin: K9
Tile: MONITOR_BOT_X46Y131

Figure 3.8: IPAD Alternate Types (there are none)

The alternate types of a given site can be determined in Vivado with the Tcl command `[get_property ALTERNATE_SITE_TYPES $site]`. The results returned from this command are always correct for UltraScale and UltraScale+ devices, but they can be incorrect for Series7 devices. That is, certain sites can return invalid alternate types through the Tcl interface. For example, a site of type IPAD returns four alternate types: IOB18M, IOB18S, IOB33M, and IOB33S. None of these types are valid alternates, as shown in Figure 3.8. It is difficult to detect invalid alternate types because setting a site to an invalid type and then querying its components can cause Vivado to crash. Therefore, it is the user's responsibility to manually remove the invalid alternate types from the family info file by looking at the correct alternate types in the Vivado GUI as shown in Figure 3.7.

Listing 3.2 shows how alternate types are represented in a family info XML. As can be seen in the XML, a series of "pinmap" tags are included for each alternate type. This is because when a site type is changed to one of its alternates, some site pins can be renamed (again, this is only true for Series7 devices). An example is shown in Figure 3.9. When an ILOGICE3 site is changed to be of type ISERDESE2, the `SR` pin is renamed to `RST`. Unfortunately, the renaming cannot be determined automatically because the Tcl command `[get_site_pins -of $site]` will always return the site pins attached to the **default type** of a site. Due to this Vivado bug, users must manually determine pin mappings using the Vivado GUI.

Listing 3.2: Example Alternate Type XML

```
<alternative>
    <name>ISERDESE2</name>
    <pinmaps>
        <pin>
           <name>RST</name>
            <map>SR</map>
         </pin>
    </pinmaps>
</alternative>
```

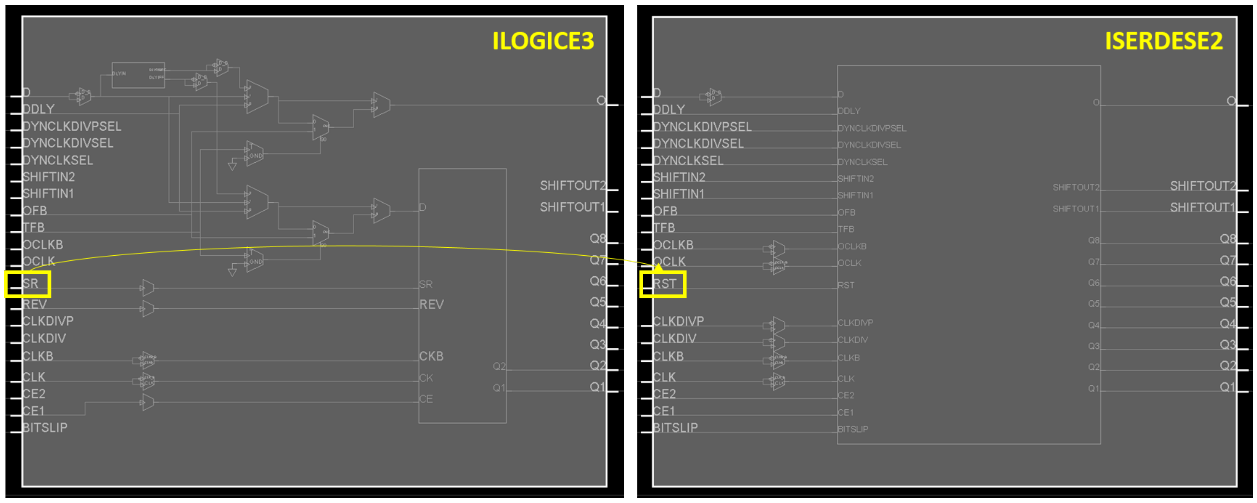


Figure 3.9: Example Alternate Site Pin Renaming

3.2.3 Compatible Types

Site `A` is said to be compatible with site `B` if the logical cells placed on site `A` can *always* be placed on site `B` as well. For example, as shown in Figure 3.10, SLICEL sites are compatible with SLICEM sites. The cells placed on the SLICEL in the figure can be moved to the SLICEM and still function identically. SLICEMs, however, are *not* compatible with SLICELs. This is because SLICEM sites support LUT RAM cells, which cannot be placed on SLICEL sites. In some cases of compatibility, the type of the compatible site must first be changed before placing cells on it.

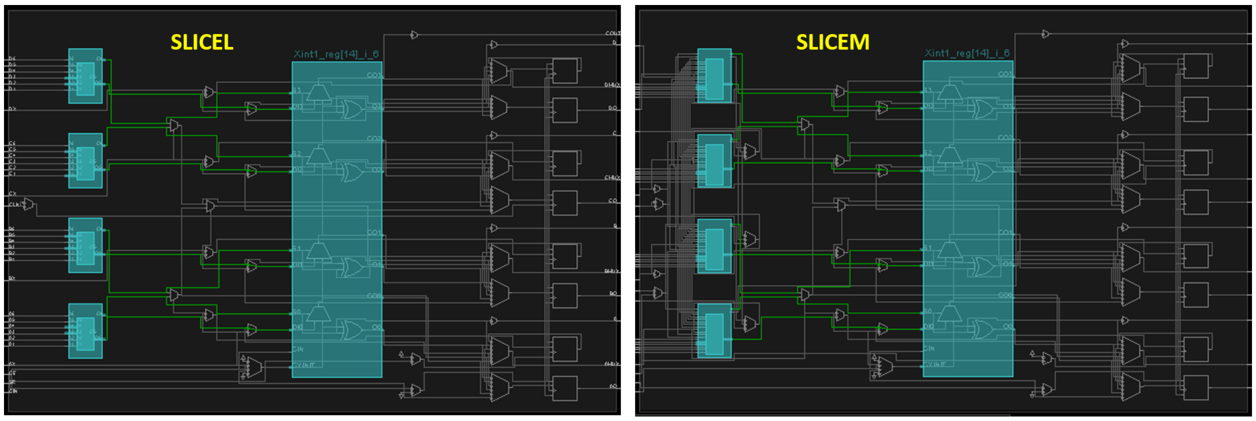


Figure 3.10: A group of cells placed on a SLICEL site (left) and a SLICEM site (right).

For instance, a RAMB36 site is compatible with a RAMBFIFO36 site, but the site type of the RAMBFIFO36 **must first be changed to RAMB36** before it is truly compatible.

Information about compatible types is useful in a variety of CAD applications. One such application is a site-level placer. To achieve the best placement results, the placer needs to understand *all* available locations where a set of cells can be placed. Without information about compatible types, the placer would only know how to target one specific site type for each group of cells, lowering the quality of results.

Compatible types are determined using Vivado's Tcl interface in two ways. The first way is looking at alternate types. In Series7 devices, a site type of IOB33M has two valid alternate types: IOB33 and IPAD. This indicates that both IOB33 and IPAD sites are compatible with IOB33M sites. A group of cells targeting an IOB33 or IPAD site can always be moved to a IOB33M site (the type just needs to be changed first). A second method is available for sites with a *single* BEL. To start, a map of site type to a list of valid cells that can be placed on the corresponding site is created. If two sites have the same list of valid placement cells (or one site is a superset of another), they can be marked as compatible. Figure 3.11 shows a visual representation of this. In the figure, "Site 1" and "Site 2" are both compatible to each other because they have the same set of placement cells. "Site 4" is compatible with "Site 3" because both cells that can be placed on "Site 4" (`X` and `Y`) can also be placed on "Site 3." "Site 3" is not compatible with "Site 4", however, because the cell `Z` cannot be placed on "Site 4."

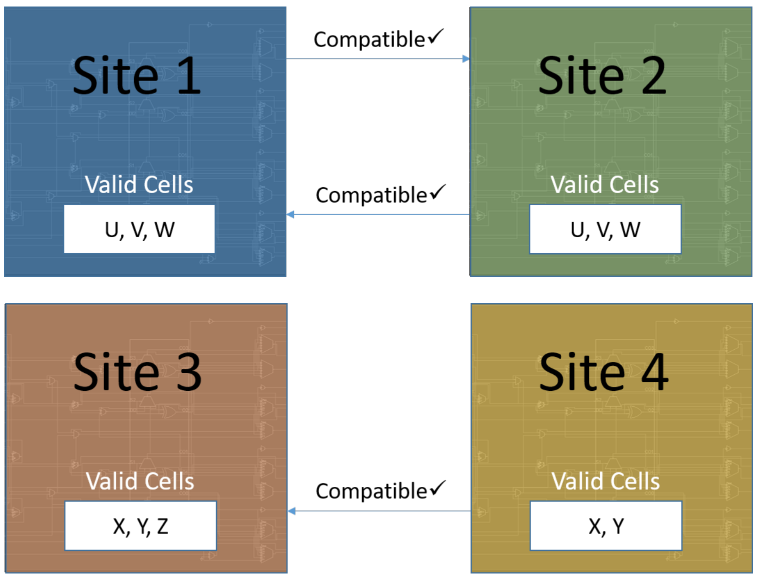


Figure 3.11: Compatibility Testing for Single-BEL Sites

Most compatible types in a family can be automatically determined using the two methods described above, but not all. The compatibility of SLICEL and SLICEM sites shown in Figure 3.10, for example, cannot be determined with Tcl commands. It is the user's responsibility to determine what other compatible types need to be added manually to the family info file (which is generally done by trial and error). Listing 3.3 shows example XML for compatible types. As was the case with alternate types, some invalid compatible types will be reported for Series7 devices. These need to be removed manually.

Listing 3.3: Compatible Type XML for SLICEL

```
<compatible_types>
  <compatible_type>SLICEM</compatible_type>
</compatible_types>
```

3.2.4 BEL Routethroughs

During the routing stage of implementation, certain BELs can be configured as PIPs in a device (i.e. they pass a signal from an input pin directly to an output pin). To fully represent the routing structure within a Xilinx FPGA, these **routethrough** connections are included in the family info file. Listing 3.4 shows how routethrough connections are represented. External tools can use this information to build a more accurate device data structure. At the time of writing, BELs are considered routethrough candidates if they are of type “LUT” or “Flip-Flop” on a SLICE site. A more detailed discussion of routethroughs is presented in section 4.1.6.

Listing 3.4: Example Routethrough XML

```
<bel>
    <name>D6LUT</name>
    <type>LUT6</type>
    <routethroughs>
        <routethrough>
            <input>A1</input>
            <output>O6</output>
        </routethrough>
        <routethrough>
            <input>A2</input>
            <output>O6</output>
        </routethrough>
    <routethroughs>
</bel>
```

3.2.5 Site PIP Corrections

XDLRC files do not distinguish the difference between site PIPs (routing muxes) and functional BELs within a site. Each of these components are simply marked as a “Bel”, even though site PIPs are certainly not BELs. The family info file corrects this by explicitly marking site PIPs as **routing muxes** or **polarity selectors**. A polarity selector is a site pip with one input that can be optionally inverted. Listing 3.5 shows how these site PIP corrections are represented.

Listing 3.5: Example Site PIP Corrections

```
<corrections>
    <modify_element>
```

```
            <name>A5FFMUX</name>
            <type>mux</type>
        </modify_element>
        <polarity_selector>
            <name>CLKINV</name>
        </polarity_selector>
    </corrections>
```

External tools can use this information to decompose a site PIP into its individual PIPs as shown in Figure 3.12. This decomposition generally makes creating routing algorithms easier. In Vivado, site PIPs are determined with the Tcl command `[get_site_pips -of $site]`. Polarity selectors are distinguished from regular site PIPs by checking if the string name of the PIP ends with either "INV" or "OPTINV".

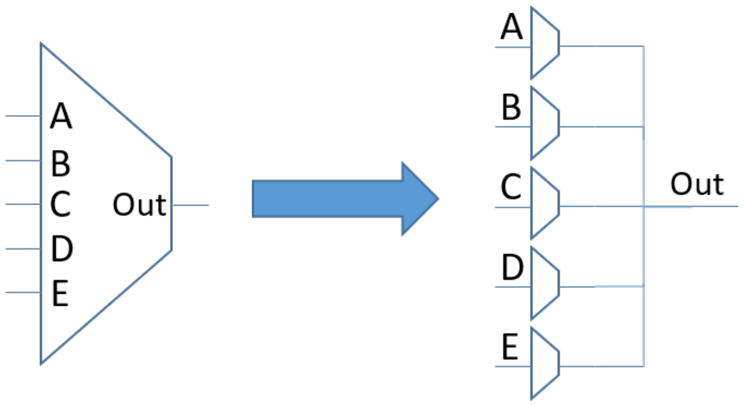


Figure 3.12: Site PIP Decomposition

3.2.6 Pin Direction Corrections

In XDLRC files, all BEL pins are given a direction of either INPUT or OUTPUT. However, there are several BEL pins in Xilinx devices that are of direction INOUT (bidirectional). The family info file marks INOUT BEL pins so that their direction can be corrected in external tools. The direction of a BEL pin in Vivado can be determined with the Tcl command `[get_property DIRECTION $belpin]`. Listing 3.6 shows how pin direction corrections are represented in XML.

Listing 3.6: Example Pin Direction Correction

```
<corrections>
    <pin_direction>
        <element>PAD</element>
        <pin>PAD</pin>
        <direction>inout</direction>
    </pin_direction>
</corrections>
```

3.3 Device Info XML

A device info XML file contains additional information **specific to a device** that is not found in the corresponding XDLRC for the device. Currently, the device info file contains only a list of package pins for the device as shown in Listing 3.7. Each package pin definition has three attributes:

1. **The name of the package pin**: Generally, package pin names are a single letter followed by a two-digit number (i.e. M17). For those that have written UCF or XDC constraints for a FPGA design targeting a Xilinx part, this format should be familiar.
2. **The corresponding PAD BEL for the package pin**: Each package pin maps to a specific BEL in the device. Both the name of the BEL as well as its parent site is recorded in the form "site/belname."
3. **An optional "is_clock" attribute**: Only a select number of package pins in a device can access the global clock routing resources. These package pins are explicitly marked in the device info file so external CAD tools can use this information when placing clock ports (or other signals that need access to global routing).

Device info files can be generated with the new `Tincr` command `[tincr::create_xml_device-_info]`. This command is fully automated, and requires no hand edits.

Listing 3.7: Example Device Info XML

```
<device_info>
    <partname>xcku025-ffva1156</partname>
    <package_pins>
        <package_pin>
            <name>AK33</name>
            <bel>IOB_X0Y155/PAD</bel>
        </package_pin>
        <package_pin>
            <name>AJ29</name>
            <bel>IOB_X0Y130/PAD</bel>
            <is_clock/>
        </package_pin>
        ...
    </package_pins>
</device_info>
```

3.4 Cell Library XML

As described in section 2.2, Xilinx netlists are composed of cell objects which are instanced from backing library primitives. The most common library primitives used in a Xilinx netlist include LUT (LUT1, LUT2, etc) and FF (FDRE, FDCE, etc.) cells. Figure 2.6 shows an example of a FDRE cell circled in red. In this case, FDRE is the backing library primitive of the cell, and the instance of the primitive is named “ff0.” A detailed knowledge of the available library cells for a device is required to perform any useful netlist modification in external tools. To provide this information, VDI defines the format for a **cell library XML**. A cell library contains the following information for each library cell that can target a specific device:

- Type
- Group (i.e. SLICE, DSP, IOB, BRAM, etc.)
- Name, direction, and type for each library cell pin

- Valid placement locations for instances of the library cell
- Default logical-to-physical pin mappings for each cell pin
- Configurable properties with default values
- Macro templates

The new `Tincr` command `[tincr::create_xml_cell_library]` can be used to generate a new cell library for a device. Listing 3.8 shows the basic layout of a cell library XML file. The following subsections discuss each portion of the cell library individually, and how they are created from Vivado.

Listing 3.8: Example Cell Library XML

```
<cell>
    <type>BUFCE_ROW</type>
    <level>LEAF</level>
    // library cell properties
    <libcellproperties>
        <libcellproperty>
            <name>CE_TYPE</name>
            <default>SYNC</default>
            <type>string</type>
            <values>ASYNC, SYNC</values>
        </libcellproperty>
        ...
    </libcellproperties>
    // library cell pins
    <pins>
        <pin>
            <name>CE</name>
            <direction>input</direction>
            <type>ENABLE</type>
        </pin>
```

```
            ...
        </pins>
        // valid bel placements
        <bels>
            <bel>
                <id>
                    <site_type>BUFCE_ROW</site_type>
                    <name>BUFCE</name>
                </id>
                // cell pin to BEL-pin mappings
                <pins>
                    <pin>
                        <name>CE</name>
                        <possible>CE</possible>
                    </pin>
                    ...
                </pins>
            </bel>
        </bels>
    </cell>
```

3.4.1 Finding Device Compatible Cells

The Tcl command `[get_lib_cells]` is used to return a list of library primitives that can be instantiated on a given device. Not all library primitives returned from this command, however, are truly compatible. For example, running the Tcl command `[ create_cell -reference AUTOBUF tmp_cell]` on an Artix7 device prints the following message: "`WARNING: [Coretcl 2-1024] Master cell 'AUTOBUF' is not supported by the current part and has been retargeted to 'IBUF'`." The AUTOBUF cell is retargeted to an IBUF to be compatible with the device. Library primitives such as AUTOBUF should not be included in the final cell library. To filter out incompatible cells, an instance of each library cell returned from `[get_lib_cells]` is created. The `REF_NAME` property on the newly created cell is compared against the expected cell library name. If the strings match, then the library cell is valid for the device and will be included in the cell library. Otherwise, the library cell will be ignored.

3.4.2 Cell Placement Locations

One of the most important components of a cell library is the valid placement locations for each available library primitive. Without this information, CAD tools that require placing cells onto the device would be much more difficult to create. Algorithm 2 shows how cell placement locations are extracted from Vivado's Tcl interface. As can be seen, the method is fairly simple.

Algorithm 2 Determining valid cell placements in Vivado

```
1:  procedure FIND_CELL_PLACEMENTS($device, cell$)
2:      $unique\_sites \leftarrow$ get_unique_sites($device$)
3:      $placement\_bels \leftarrow \varnothing$
4:      for each $site \in unique\_sites$ do
5:          try
6:              place_cell_on_site($cell, site$)          ▷ First, try placing the cell on the site
7:          catch (Invalid Placement Exception)
8:              continue
9:          end try
10:         $bels \leftarrow$ get_bels_of_site($site$)     ▷ Next, try placing the cell on each BEL of the site
11:         for each $bel \in bels$ do
12:             try
13:                 place_cell_on_bel($cell, bel$)
14:             catch (Invalid Placement Exception)
15:                 continue
16:             end try
17:             $cell\_bel \leftarrow$ get_bel_of_cell($cell$)
18:             if $bel = cell\_bel$ then
19:                 $placement\_bels \leftarrow placement\_bels \cup bel$
20:                 record_pin_mappings($cell, bel$)
21:             end if
22:         end for
23:     end for
24:     return $placement\_bels$
25: end procedure
```

For a given cell object, the first step is to try and place the cell onto each unique site type in the device using the Tcl command `[place_cell $cell $site]`. With this command, Vivado scans the site for a suitable placement location for the cell. If there is no suitable placement location, an exception is thrown and the site is skipped. Otherwise, the algorithm then tries to place the cell on each individual BEL in the site with the Tcl command `[place_cell $cell $bel]`. Again, an exception is thrown if the cell cannot be placed on the BEL. If both tests pass, one final check is performed. Using the command `[place_cell $cell $bel]`, Vivado sometimes automatically remaps the cell placement to a different BEL than the one specified (it is unclear why this happens). To work around this functionality, the actual BEL placement is compared to the expected BEL placement to make sure they match. If they do match, then the BEL is marked as a valid placement location for the cell. The last step is to find all cell pin to BEL pin mappings using the Tcl command `[get_bel_pins -of $cell_pin]` after the cell has been placed. This process is repeated for each cell available in the device.

3.4.3 Configurable Cell Properties

Most Vivado library cells have configurable properties that customize their functionality. Listing 3.9 shows the configurable properties of a FDRE cell with the corresponding default values. The property `CONFIG.IS_R_INVERTED`, for example, is used to specify either a high-asserted or low-asserted reset signal for a flip-flop.

Listing 3.9: FDRE configurable properties and default values

```
 Property                      Type   Read-only Value
CONFIG.INIT                   string true
CONFIG.INIT.DEFAULT           string true      1'b0
CONFIG.IS_C_INVERTED          string true
CONFIG.IS_C_INVERTED.DEFAULT  string true      1'b0
CONFIG.IS_D_INVERTED          string true
CONFIG.IS_D_INVERTED.DEFAULT  string true      1'b0
CONFIG.IS_R_INVERTED          string true
CONFIG.IS_R_INVERTED.DEFAULT  string true      1'b0
```

For each configurable property on a cell, the default value, possible values, max, and min are all stored in the cell library. It is important to include these configurable properties with the cell library XML for two reasons:

1. External netlist modification algorithms (specifically ones that add new cells to a design) need to understand how a cell can be configured to create functionally correct netlists.

2. When Vivado creates an equivalent EDIF representation of a design using the Tcl command `[ write_edif]`, only **non-default** properties of cells are included in the resulting file (Vivado already knows the default values so there is no need to print them). Therefore, to fully reconstruct the netlist in external tools, default cell properties are included in the cell library.

The following Tcl regular expression is used to determine the configurable property names for a library cell in Vivado:

```
CONFIG\.([^\.]+)\.DEFAULT$
```

3.4.4 Macro Primitives

The discussion so far in this section has been focused on Xilinx leaf primitives only. Xilinx also supports **macro** primitives. A macro is a hierarchical cell that groups one or more leaf cells together to perform a specific function. An example macro is shown in Figure 3.13 for an IOBUF cell. An IOBUF macro cell contains two **internal** cells: one of type OBUFT and the other of type IBUF. It also contains one internal net that connects the two internal cells together.

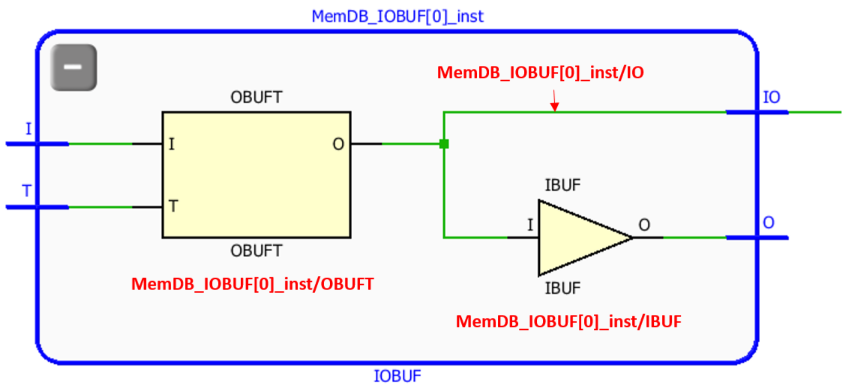


Figure 3.13: Vivado Macro Cell

In addition to the leaf primitives, a cell library XML also contains definitions for each of the available macros in a device. The XML macro definition, however, is much less detailed than its leaf cell counterpart. Only the internal structure of the macro is given as shown in Listing 3.10. As the listing shows, there are three parts to a macro XML definition:

1. A list of internal cells.
2. A list of external macro pins, and the corresponding internal cell pins to which they connect.
3. A list of internal nets. These are nets within the macro that only connect to internal pins (i.e. they do not connect to a macro pin).

Listing 3.10: Cell Library Macro XML

```
<macro>
  <type>RAM128X1D</type>
  // List of internal cells with name and type of each
  <cells>
      <internal>
          <name>DP.HIGH</name>
          <type>RAMD64E</type>
      </internal>
   ...
  </cells>
  // List of macro pins with name, direction, type, and internal connections
  <pins>
      <pin>
          <name>DPO</name>
          <direction>output</direction>
          <type>MACRO</type>
          <internalConnections>
              <pinname>F7.DP/O</pinname>
          </internalConnections>
      </pin>
```

```
        ...
    </pins>
    // List of internal nets, and the internal cell pins they connect to
    <internalNets>
        <internalNet>
            <name>DPO0</name>
            <pins>
                <pinname>F7.DP/I0</pinname>
                <pinname>DP.LOW/O</pinname>
            </pins>
        </internalNet>
        ...
    </internalNets>
</macro>
```

The goal of this macro format is to help user reconstruct the internal contents of macros in external tools. This is required because EDIF files produced from Vivado do not contain the internal structure of macro primitives (discussed in more detail in subsection 4.1.4). Information about possible placement locations, pin mappings, and configurable properties of a macro are not included in the current version of the cell library. They may be added in future revisions.

3.5 Conclusion

This chapter presents the required updates to `Tincr` to fully support Vivado device extraction through VDI. VDI defines four file formats that can represent a Vivado device:

1. XDLRC
2. Family Info XML
3. Device Info XML
4. Cell Library XML

XDLRC files are the most important device file, and verbosely detail the many components within a Xilinx FPGA. Using VSRT, a complete set of primitive definitions has been added to `Tincr` for UltraScale and UltraScale+ devices. Using these primitive definitions (and the other changes described in section 3.1) valid XDLRC files can now be generated for these architectures. This enables functioning CAD tools on these architectures for the first time.

A *familyInfo.xml* file augments a XDLRC file with additional useful information about the family of a device. This information includes alternate types, compatible types, BEL route-throughs, and site-level modifications and corrections. Using the `Tincr` command `[tincr::create_xml_family_info]` and applying the required hand edits described in section 3.2, a complete set of family info files has been created for all Series7 and UltraScale families. Users can choose to use these completed files instead of wrestling with the hand edits themselves.

A *deviceInfo.xml* files further augments a XDLRC with device-specific information not found in the XDLRC. Currently this only includes the package pins for a device as described in section 3.3. However, additional properties of a device deemed useful may be added to this file in future VDI revisions.

A *cellLibrary.xml* details the Xilinx primitive cells that can target a given device. It provides extremely useful information to external CAD tools including possible BEL placements for each cell, configurable cell properties, macro templates, and more (reference section 3.4). The files described in this chapter work together to provide a comprehensive representation of Vivado FPGAs. The hope is that it can be leveraged to create a variety of external CAD tools.

CHAPTER 4. VDI: DESIGN IMPORT/EXPORT

The second goal of VDI is to create an open-source format capable of fully representing Vivado FPGA designs at any stage of implementation. A first attempt at this was to use Vivado's native design checkpoint format (DCP). Design checkpoints save and restore the progress of an implemented design using the associated Tcl commands [`write_checkpoint`] and [`open_checkpoint`]. However, the format of DCPs is serialized, making them unsuitable for general purpose use. De-serializing files within a DCP is against Xilinx's terms of service and would not be guaranteed to work with future versions of Vivado.

Instead, Vivado Tcl commands are used to define an alternate checkpoint representation for open-source tools. VDI defines two different external design representations:

1. **RapidSmith Checkpoints (RSCP)**: Represents designs that have just been exported from Vivado. These checkpoints are intended to be loaded into external tools.

2. **Tincr Checkpoints (TCP)**: Represents designs that can be loaded back into Vivado.

Each of these checkpoints are capable of representing a Vivado design post-synthesis, post-place, and post-route, enabling the design flows shown Figure 4.1. The remainder of this chapter describes each external design representation, and the required Tcl commands for each. Section 4.1

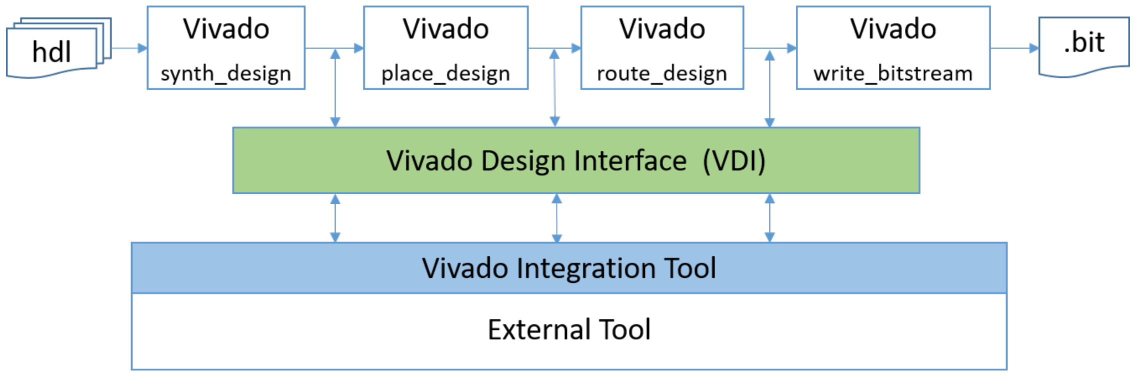


Figure 4.1: Vivado Design Interface (VDI) Design Flows

introduces RSCPs and section 4.2 introduces TCPs. Section 4.3 lists the variety of Tcl challenges associated with using the Vivado Tcl interface to represent designs externally. Section 4.4 includes a discussion on why two different external design formats are necessary.

4.1 Design Export

VDI exports Vivado FPGA designs in the form of **RapidSmith Checkpoints** (RSCP). A RSCP is a fileset containing 6 individual files:

- design.info
- netlist.edf
- macros.xml
- contraints.xdc
- placement.rsc
- routing.rsc

Each file represents a specific aspect of a Vivado FPGA design, and is described in the following subsections. The new `Tincr` command `[tincr::write_rscp]` can generate valid RSCPs for **fully-flattened** Vivado designs. It is important to note that the name "RapidSmith" does not make the checkpoints exclusive to RapidSmith. The name was arbitrarily chosen to match the external CAD framework used to test and validate VDI. Any external tool that desires to operate on Vivado designs can use these checkpoints.

4.1.1 Design.info

The *design.info* file of a RSCP is reserved for additional information about a design that is not related to the design netlist or implementation. It currently only holds two pieces of information: (1) the part name that the design is implemented on, and (2) the checkpoint "mode". The part name is technically redundant, in that is also included in the EDIF netlist (described in the next subsection), but can be easier to parse than its EDIF equivalent. The checkpoint mode refers to the type of checkpoint that was exported from Vivado. If the Vivado design was implemented out-of-context, then the mode value will be `out_of_context`. Otherwise, the mode value will be `regular`. Because out-of-context checkpoints do not route to peripheral FPGA pins, external tools or frameworks that parse RSCPs may need to handle out-of-context designs differently. Listing 4.1 shows an example *design.info* file.

Listing 4.1: Sample design.info

```
part=xc7a100tcsg324-3
mode=out_of_context
```

4.1.2 Netlist.edf

The *netlist.edf* file within a RSCP is an EDIF netlist representing the logical portion of a design. It details all cells, nets, and top-level ports within a design, and is generated from Vivado using the Tcl command `[write_edif]`. An example EDIF file is shown Listing 4.2. External tools can use any open-source EDIF parser to translate the EDIF into their own design representation to perform netlist modifications.

Listing 4.2: Sample EDIF

```
(Library work
  (edifLevel 0)
  (technology (numberDefinition ))
  (cell add (celltype GENERIC)
    (view add (viewtype NETLIST)
      (interface
        (port a (direction INPUT))
        (port b (direction INPUT))
        ...
      )
      (contents
        (instance cout_OBUF (viewref netlist (cellref OBUF (lib
            hdi_primitives))))
        (instance cout_1UT (viewref netlist (cellref LUT3 (libref
            hdi_primitives)))
          (property INIT (string "8hE8"))
)
...
```

4.1.3 Constraints.xdc

The *constraints.xdc* file within a RSCP stores all user-defined XDC constraints on a Vivado design. XDC constraint files are similar to UCF files for ISE designs. Constraints can be used to set the clock frequency, constrain a top-level port to a specific package pin on the device, and set other physical implementation details. An example RSCP constraints file can be seen in Listing 4.3, and is essentially a list of Tcl commands.

Listing 4.3: Sample constraints.xdc

```
create_clock -period 5.000 -name sysClk -waveform {0.000 2.500}
set_property IOSTANDARD LVCMOS18 [get_ports clk]
set_property IOSTANDARD LVCMOS18 [get_ports ena]
set_property PACKAGE_PIN E15 [get_ports {Yin[12]}]
set_property PACKAGE_PIN H17 [get_ports {Xin[14]}]
set_property PACKAGE_PIN D18 [get_ports {Xin[7]}]
```

4.1.4 Macros.xml

Many fully-flattened Vivado designs contain **hidden macros**. Hidden macros are Vivado macro primitives that are not returned from the Tcl command `[get_lib_cells]`, but are used in a netlist anyway. For example, the macro primitive IOBUF is found in a variety of Series7 designs, but the Vivado Tcl command `[get_lib_cells IOBUF]` returns an empty string indicating that the library cell cannot be found. This breaks the well defined assumption that `[get_lib_cells]` returns a list of *all* Xilinx primitives that can be used in a design netlist. Hidden macros create two problems in particular:

1. Because `[get_lib_cells]` is used to generate the cell library XML described in section 3.4, the default cell library XML returned from `[tincr::create_xml_cell_library]` is incomplete and only supports the subset of designs without hidden macros.

2. Hidden macros are "primitive" cells in Vivado's perspective. Therefore, the internal structure of hidden macros is not expanded in the *netlist.edf* file described above. They are simply black box cells to external tools

RSCPs handle hidden macros by including a *macros.xml* file which contains template information about each hidden macro in a Vivado design. This file has the same format as regular cell library macros as shown in Listing 3.10. External tools can augment their default cell library with the cells found in *macros.xml* to create a complete list of primitives used in a given design. An alternative approach is to include placement information for the macro.

4.1.5 Placement.rsc

The *placement.rsc* file within a RSCP stores all the placement information of a Vivado design. Specifically, the file includes four different tokens to differentiate placement data:

- **LOC**: Gives the site and BEL that a cell in the netlist is placed on. An example is shown on line 3 of Listing 4.4. In this case, the cell named "cout_OBUF_inst_i_1" is placed onto the "A6LUT" BEL of the "SLICE_X0Y51" site. The site type (SLICEL) is also given after the site name in case the site needs to be changed to an alternate type before placing the cell (not required for the example cell placement).

- **PINMAP**: Gives the logical-to-physical mapping for each cell pin attached to a given cell. Line 4 of Listing 4.4 shows the cell pin mappings for "cout_OBUF_inst_i_1." As can be seen, the "O" cell pin is mapped to the "O6" BEL pin of the A6LUT, the I0 cell pin is mapped to the A4 BEL pin, etc. These pin mappings are required in *placement.rsc* because certain cells can have different pin mappings based on how they are configured. For example, the pin mappings for a BRAM cell with a data width of 72 bits differ from the pin mappings of the same BRAM cell with a data width of 9 bits, even if they are placed at the same physical location. `PINMAP` tokens help external tools guarantee that the pin mappings for each cell are

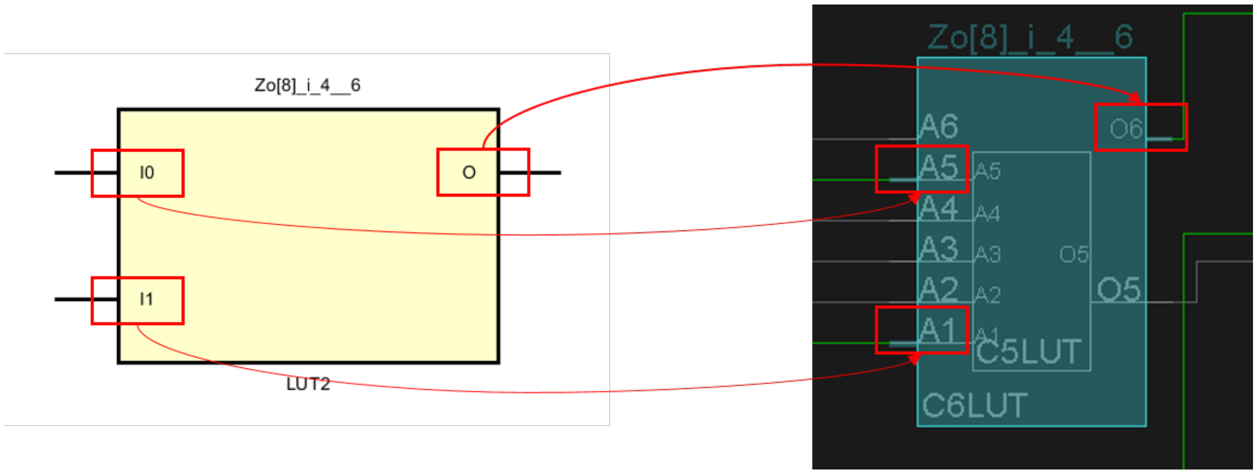


Figure 4.2: Example Cell Placement

correct. Cell pin mappings contained in the cell library (section 3.4) are for the cell's **default configuration** only. Future work includes creating a more detailed pin mapping in the cell library so that the `PINMAP` token is no longer required in *placement.rsc*.

- **PACKAGE_PIN**: Gives the site and BEL that a top-level port is mapped to. Line 6 of Listing 4.4 shows an example for a port named "fsync_out." In this case, "fsync_out" is mapped to the "PAD" BEL on the "IOB_X1Y30" site.
- **IPROP**: Gives the value of an internal cell property (to a macro). Internal cell properties are not included within in the EDIF netlist, and so are included in *placement.rsc* for completeness.

If a design has not yet been placed, *placement.rsc* will be empty. It is important to note that for macro primitives in a design, only the placement information for internal cells to the macro are included. Listing 4.4 demonstrates the order in which the above tokens appear in the file. Figure 4.2 shows a visual representation of how cells are placed onto BELs in Vivado.

Listing 4.4: Sample placement.rsc

```
1 IPROP seq_ram_reg_0_127_0_0/DP.HIGH INIT 64'h0000000000000000
2 ...
3 LOC a_IBUF_inst R10 IOB33 INBUF_EN
4 PINMAP a_IBUF_inst O:OUT I:PAD
```

```
LOC cout_OBUF_inst_i_1 SLICE_X0Y51 SLICEL A6LUT
PINMAP cout_OBUF_inst_i_1 O:O6 I0:A4 I1:A5 I2:A6
...
PACKAGE_PIN fsync_out IOB_X1Y30 PAD
PACKAGE_PIN data_out[7] IOB_X1Y33 PAD
...
```

4.1.6 Routing.rsc

The *routing.rsc* file is the most complex file of a RSCP, and stores a complete description of all routing resources used in a Vivado design. Specifically, it contains the following:

- The intrasite routing configuration for each site in the form of used site PIPs (routing muxes)
- A list of BELs configured as routethroughs (both LUT and flip-flops).
- A list of BELs configured as static VCC/GND sources.
- **Intrasite** nets.
- Routed **intersite** nets with their corresponding site pins and used PIPs.
- The **merged** physical routing information for VCC and GND nets.

An example *routing.rsc* is shown in Listing 4.5. If a design has not yet been routed, only the intrasite routing information will be included in this file. If the design has not yet been placed, this file will be empty. The rest of this section describes each component within a *routing.rsc* file, and how they can be determined in Vivado.

Listing 4.5: Sample routing.rsc

```
SITE_PIPS SLICE_X9Y80 SRUSEDMUX:0 CEUSEDMUX:IN CLKINV:CLK ...
SITE_PIPS SLICE_X13Y80 PRECYINIT:AX SRUSEDMUX:0 COUTUSED:0 ...
...
VCC_SOURCES SLICE_X5Y104/C6LUT/O6 SLICE_X5Y102/C6LUT/O6 ...
GND_SOURCES SLICE_X2Y106/D6LUT/O6 SLICE_X2Y106/C6LUT/O6 ...
```

```
LUT_RTS SLICE_X5Y101/B6LUT/A6/O6 SLICE_X40Y96/DFF/D/Q ...
...
INTRASITE AddSub[10]
INTERSITE AngStep1[0] SLICE_X2Y82/AX SLICE_X2Y87/AQ SLICE_X2Y84/A1 ...
ROUTE AngStep1[0] INT_X28Y26/INT.LOGIC_OUTS_W1->>SDNDSW_W_0_FTS ...
...
VCC INT_L_X2Y107/INT_L.VCC_WIRE->>IMUX_L42 ...
START_WIRES INT_L_X2Y107/VCC_WIRE INT_L_X2Y106/VCC_WIRE ...
GND INT_R_X3Y106/INT_R.GND_WIRE->>GFAN1 ...
START_WIRES INT_R_X3Y106/GND_WIRE INT_L_X4Y106/GND_WIRE ...
```

Site PIPs

The internal routing structure of nets inside Vivado sites are represented by a set of used site PIPs. A string of site PIPs enables a connection between site components. An example is shown in Figure 4.3 where the used site PIPs are circled in red. As can be seen, the `ACY0:O5` site PIP is enabled and connects the `O5` pin of the `A5LUT` BEL to the `DI0` pin of the `CARRY4` BEL. Site PIPs can also be used to connect site pins to BEL pins. The Vivado Tcl command `[get_site_pips -of $site -filter IS_USED]` can be used to obtain a list of used site PIPs within a given site. `[tincr::write_rscp]` formats the site pips as shown on line 1 and 2 of Listing 4.5. External

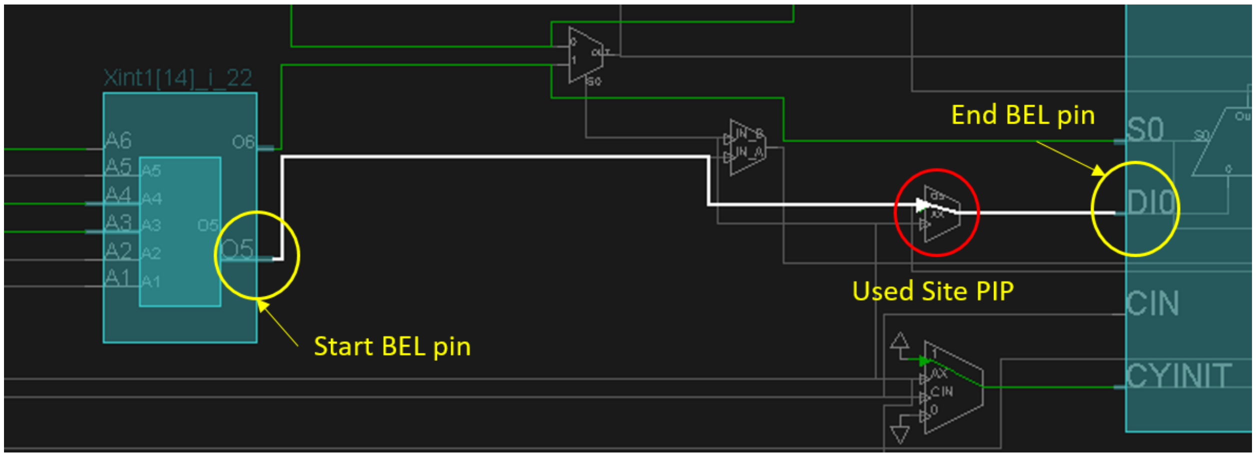


Figure 4.3: Site PIP Usage

tools can use the `SITE_PIPS` token within a *routing.rsc* file to reconstruct the internal routing of each site.

LUT Routethroughs

Besides their use in implementing logic equations, LUT BELs can also be configured as PIPs in a fully-routed FPGA design (known as a routethrough). A LUT is marked as a routethrough when its configuration equation, `CONFIG.EQN`, maps the value of a single input pin directly to the output pin. For timing, the A6 pin is the most preferable option for a routethrough since it is the fastest, but pins A1-A5 can also be used in cases of routing congestion. Routethrough LUTs are not explicitly represented in a design netlist since there is no cell placed on the corresponding BEL, so they need to be included in the *routing.rsc* of a RSCP. Otherwise, designs could not be fully represented in external tools. Figure 4.4 shows two example routethrough LUTs in Vivado.

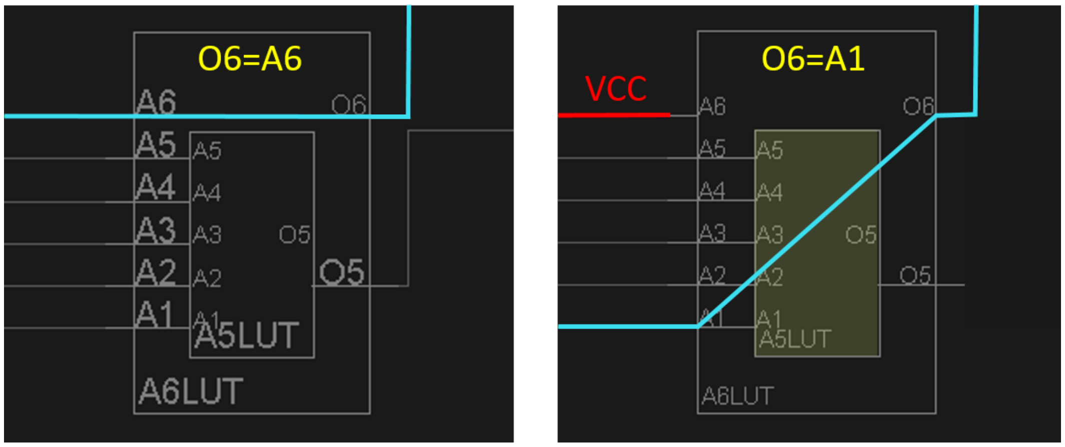


Figure 4.4: Two examples of LUTs configured as routethroughs in Vivado. The net highlighted in red represents VCC.

For the LUT on the left, the input pin `A6` is mapped directly to the output pin `O6`. The corresponding configuration equation for this BEL is `CONFIG.EQN:O6=(A6)`. For the LUT on the right, the input pin `A1` is mapped directly to the output pin `O6`. However, the structure of the configuration equation for this LUT is slightly different. As figure 4.4 shows, VCC is routed to the `A6` input pin. In this case, the configuration equation on the BEL is `CONFIG.EQN:O6=(A6+~A6)*((A4))`. Using Vivado's TCL interface, LUT routethroughs can be identified by matching their `CONFIG.EQN` property against the Tcl regular expression

```
(O[5,6])=(?:\(A6\+~A6\)\*)?\(+(A[1-6])\)+ ?.
```

On design export, LUTs that are identified as routethroughs are included in the *routing.rsc* file with the token `LUT_RTS`. An example is shown on line 6 of Listing 4.5. Each routethrough is represented in the form "site/bel/input_pin/output_pin." It is important to note that not all LUT BELs are tested for routethroughs. For an arbitrary LUT to be used as a routethrough, it needs to satisfy three conditions. (1) Its parent site needs to be used (i.e. there is at least one cell placed onto the site), (2) no cell is currently placed on the corresponding bel, and (3) at least one input bel pin is not currently being used.

Permanent Latches

A **permanent latch** in Vivado is a Flip Flop (FF) BEL which has been configured as a latch with its "set" signal tied to VCC. This means that the data pin of the latch always passes its value to the output pin of the latch, and no state is retained. An example is shown in Figure 4.5. As the figure shows, permanent latches look very similar to LUT routethroughs described in the previous section.

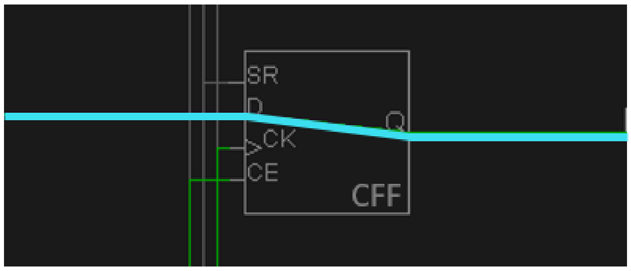


Figure 4.5: Flip-Flop BEL Configured as a Permanent Latch in Vivado

Because of this similarity, *routing.rsc* files treat permanent latches the same as LUT routethroughs. That is, each permanent latch is included alongside the `LUT_RTS` token as shown on line 6 of Listing 4.5. The string "SLICE_X40Y96/DFF/D/Q" gives an example of what a permanent latch looks like in the *routing.rsc* file. Permanent latches in Vivado can be identified based on

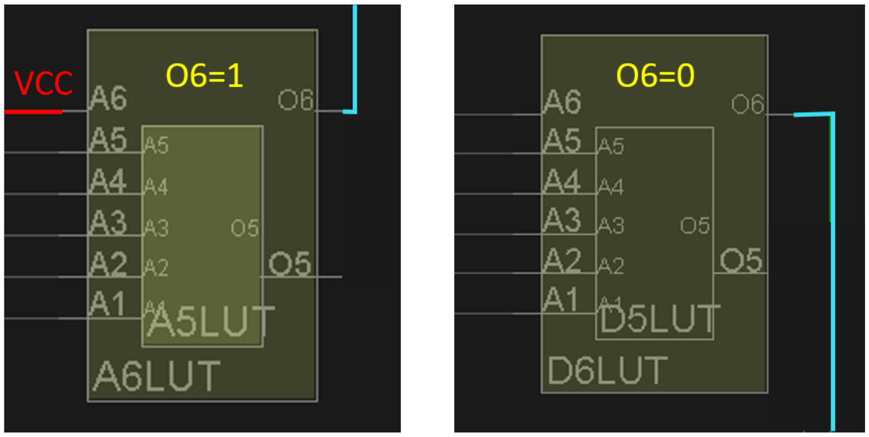


Figure 4.6: Two LUTs Configured as Static Sources in Vivado

two qualifications: (1) there is no cell placed on the corresponding FF BEL, and (2) the property `CONFIG.LATCH_OR_FF` of the BEL is set to `LATCH`.

LUT Static Sources

Similar to their use as routethroughs, LUT BELs can also be configured as GND or VCC signal sources. Examples of both are shown in Figure 4.6. The LUT on the left of the figure drives a VCC signal and the LUT on the right drives GND. In both cases, the logical netlist of a design does not represent the use of these LUTs in any way. Therefore, the *routing.rsc* file marks VCC and GND source LUTs with the tokens `VCC_SOURCES` and `GND_SOURCES` respectively. This is shown on line 4 and 5 of Listing 4.5. External tools can use the BELs marked `VCC_SOURCES` and `GND_SOURCES` to fully recreate the routing of GND and VCC nets. Static source LUTs are identified in Vivado by matching their `CONFIG.EQN` property against the Tcl regular expression

```
(O[5,6])=(?:\(A6\+~A6\)\*)?\(?[0,1]\)? ?.
```

LUTs that match the expression are formatted in *routing.rsc* with the form “site/bel/source_pin.” It is important to note that external CAD tools can also make use of this routing structure while creating routing algorithms for Xilinx devices.

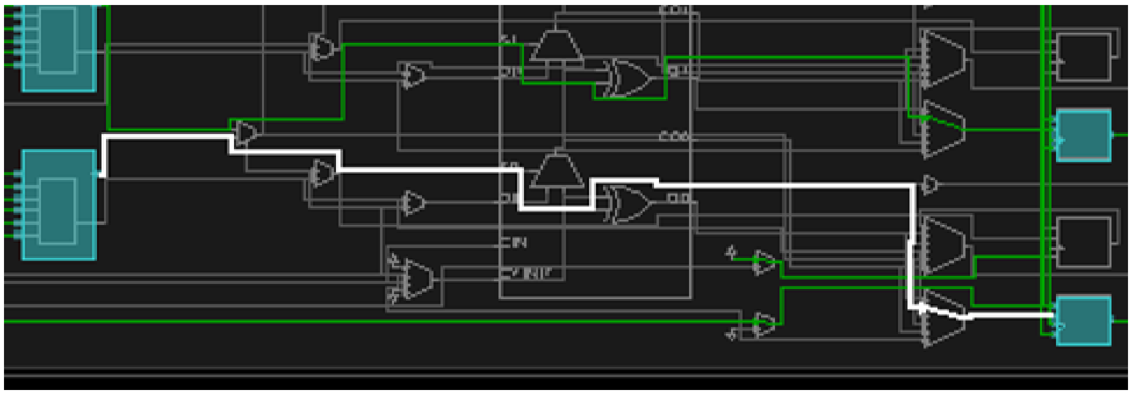


Figure 4.7: Vivado Intrasite Net (highlighted in white)

Intrasite Nets

There are two types of nets in Vivado: **intrasite** nets and **intersite** nets. Intrasite nets are those that are contained completely within site boundaries. Figure 4.7 shows an example intrasite net, which connects to only BEL pins. The *routing.rsc* file marks all intrasite nets with the token `INTRASITE` as shown on line 8 of Listing 4.5. No additional information is given about intrasite nets. External tools can reconstruct intrasite nets by using the source BEL pin of the net in conjunction with the used site pips of the corresponding site. A list of intrasite nets can be obtained in Vivado with the Tcl command `[get_nets -filter {ROUTE_STATUS==INTRASITE}]`.

Intersite Nets

A majority of nets in a FPGA design are **intersite** nets. Intersite nets are those that use general routing fabric to connect cells across site boundaries as shown in Figure 4.8. After a design has been placed in Vivado, each intersite net is partially routed. That is, the intrasite portions of the net (the wires within sites) are routed to site pins via site wires. To represent an intersite net at the placement stage of implementation, the *routing.rsc* file includes an `INTERSITE` token, which lists all site pins a net is connected to. Line 9 of Listing 4.5 shows an example `INTERSITE` specification. Site pins are important for recreating the routing structure of nets in external tools because they can be used to (a) build the intrasite portions of a net and (b) determine if a net is fully routed (nets that route to all their site pins are fully routed by definition). In Vivado, the site pins attached to a net can be determined with the Tcl command `[get_site_pins -of $net]`.

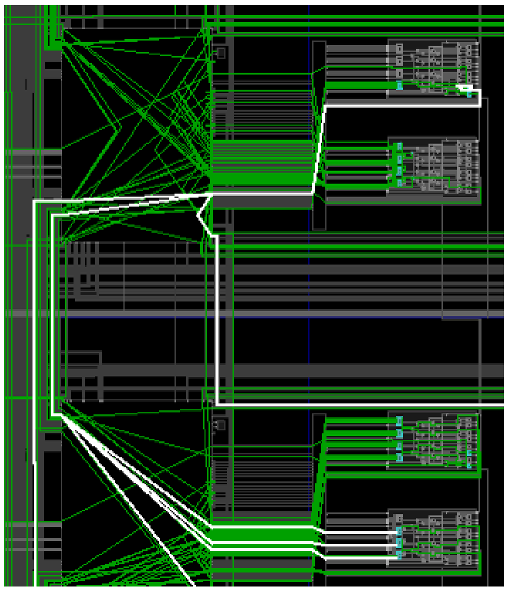


Figure 4.8: Vivado Intersite Net (highlighted in white)

During the routing stage of implementation, all site pins of a net are connected together through the general routing fabric of the FPGA. At this point the net is considered fully routed. The physical structure of a fully routed net can be expressed three different ways in Vivado: (1) the ROUTE property on the net, (2) the wires used in the net, or (3) the device PIPs used in the net. For reasons discussed in section 4.3, device PIPs are the best option for external tools. The *routing.rsc* file uses the token `ROUTE` for specifying the PIPs of a routed net. Line 10 of Listing 4.5 shows an example `ROUTE` specification. The Tcl command `[get_pips -of $net]` can be used to get a list of all PIPs being used in a Vivado net. Chapter 5 gives an algorithm that can be used to reconstruct a net in external tools based on its PIPs.

VCC and GND Nets

In general, a design can have more than one VCC net in the logical netlist. Each of these nets should have their own unique physical information once the design is routed (based on which wires they use), but this is not the case with Vivado. Instead, Vivado reports that they each use the same set of wires which correspond to a combination of *all* VCC nets. The same applies to GND nets. There are two possible solutions to resolve this discrepancy. The first is to merge all logical VCC (or GND) nets in the netlist into a single net, matching the merged physical routing.

The second is to partition the physical information so that each logical net has unique physical properties.

For simplicity, RSCPs choose to implement the first option. Lines 12-15 of Listing 4.5 show how VCC and GND nets are represented in a *routing.rsc* file. The `VCC` and `GND` tokens serve the same purpose as `ROUTE`, they contain the used PIPs for all VCC and GND nets. The `START_WIRES` token gives a list of all wires connected to VCC and GND sources. Starting at the specified start wires, VCC and GND nets can be reconstructed in external tools. The Tcl command `[get_nets -filter {TYPE==POWER}]` is used to find all VCC nets in a design, and `[get_nets -filter {TYPE==GROUND}]` is used to find all GND nets in a design. The start wires for VCC and GND nets are found by parsing their corresponding `ROUTE` property strings.

4.2 Design Import

RSCPs are only used to represent exported Vivado designs and cannot be directly imported back into Vivado. They must first be converted to **Tincr Checkpoints** (TCP) in an external tool or framework. TCPs are a fileset proposed in the original `Tincr` distribution, and serve as an alternate representation of Vivado designs which can be imported into Vivado through the VDI interface. Each TCP contains the following four files:

- netlist.edf
- placement.xdc
- constraints.xdc
- routing.xdc

As can be seen, TCPs are largely based on Vivado XDC files, which allow a subset of Tcl commands to set constraints on a variety of Vivado objects. For example, cells can be placed and nets can be routed with XDC constraints. The XDC format is desirable for two reasons in particular: (1) a single Vivado Tcl command `[read_xdc]` can parse and apply XDC constraint files and (2) `[read_xdc]` is much faster than regular Tcl scripts executing the same commands.

Each file in a TCP represents a specific aspect of a Vivado FPGA design, and is described in the following subsections as a review for the reader. The *netlist.edf* and *contraints.xdc* file will not be reviewed because they are identical to their RSCP counterparts presented in section 4.1. The `Tincr` function used to parse TCPs, `[tincr::read_tcp]`, has been updated in a variety of ways to support fully importing Vivado designs. These modifications, and the limitations of XDC, are described more in section 4.3

4.2.1 Placement.xdc

Listing 4.6 shows an example *placement.xdc* file within a TCP. Lines 1-3 of the listing show how a cell is placed in Vivado using XDC commands. As can be seen, the first command sets the `BEL` property on the cell to the corresponding `SITE_TYPE.BEL_NAME`. The second command sets the `LOC` property on the cell, which specifies the actual site location. The ordering of these commands relative to one another is crucial, because if the `LOC` property is set before the `BEL` property an error will be thrown in Vivado. For cells of group `LUT`, `INV`, or `BUF` the cell pin to BEL pin mappings must be set manually by the user. This can be done by setting the `LOCK_PINS` property on the cell as shown on line 3, and is most often used for LUT cells. Line 4 of Listing 4.6 shows how a top-level port is mapped to a FPGA package pin.

Listing 4.6: Sample placement.xdc

```
1    set_property BEL SLICEM.H6LUT [get_cells {u3/angle.Ao[16]_i_3}]
2    set_property LOC SLICE_X50Y30 [get_cells {u3/angle.Ao[16]_i_3}]
3    set_property LOCK_PINS { I0:A3 I1:A1 } [get_cells {u3/angle.Ao[16]_i_3}]
4    ...
5    set_property PACKAGE_PIN AD10 [get_ports {Rout[16]}]
6    ...
```

4.2.2 Routing.xdc

During placement, the intrasite routing structure of each site is automatically configured in Vivado based on what cells are placed onto the corresponding site. Therefore, the only necessary information in the *routing.xdc* file is the intersite routing specification for each net. Specifically, the physical structure of a net is specified in Vivado by setting its `ROUTE` string property (also called a directed routing string) as shown in Listing 4.7. A directed routing string represents the tree structure of a physical route by using nested brackets ("{") to represent branching. As an

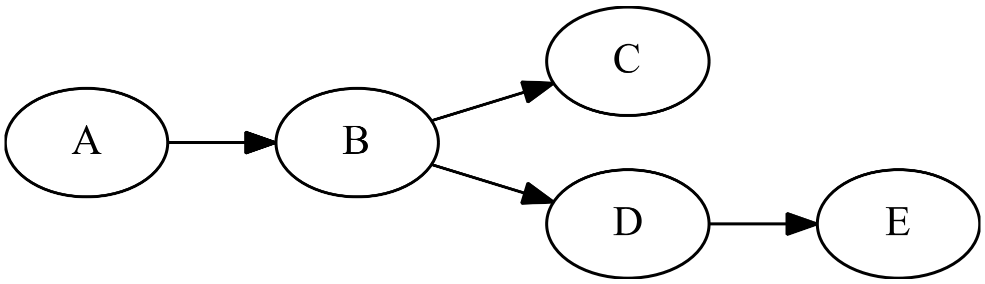


Figure 4.9: Sample Route (A, B, C, D, and E represent device wires)

example, take the hypothetical route shown in Figure 4.9. One valid `ROUTE` string associated with this route is `{ A B { D E } C }`. Another possible `ROUTE` string is `{ A B { C } D E }`, they both represent the route shown in the figure. `ROUTE` strings can be formatted to either include the tile of each wire (line 2 of Listing 4.7), or use only relative wire names (line 3 of Listing 4.7). The advantage of including tile names is to remove any possible ambiguity for a route. The advantage of using relative wire names is that they result in smaller *routing.xdc* files. It is considered best practice, however, to format `ROUTE` strings using tile names to ensure the correctness of the route when it is imported into Vivado.

Listing 4.7: Sample routing.xdc

```
1   ...
2   set_property ROUTE { CLEL_R_X36Y8/CLE_L_SITE_0_COUT ... } [get_nets {Zo}]
3   set_property ROUTE { CLE_L_SITE_0_COUT ... } [get_nets {Zo}]
4   ...
```

4.3 Vivado Tcl Interface Challenges

Vivado's Tcl interface supplies the necessary functionality to generate and parse external design representations. VDI uses the interface to define the formats for RSCPs and TCPs as described in the previous sections. There are, however, a variety of challenges associated with using Tcl commands to generate RSCPs and parse TCPs. Broadly, these challenges can be grouped into four distinct categories.

1. Vivado design import rules: External designs need to be imported into Vivado in a very specific way, or else import can fail.

2. Logical to physical mismatches: When a netlist is implemented on a device, most physical components match to a corresponding logical component (i.e. a BEL maps to a cell). But, there are some aspects of an implemented design that aren't represented in the logical netlist.

3. Tcl objects that are incomplete, ambiguous, or return incorrect results when queried.

4. Parts of a device or design that cannot be configured with Tcl commands

The following subsections document each issue associated with generating RSCPs or parsing TCPs, with their appropriate workaround. In some cases, it is up to external tools to provide a solution. As previously mentioned, each of these issues is specific to Vivado 2016.2, and may be fixed in future tool versions.

4.3.1 Ambiguous ROUTE Strings

As described in subsection 4.2.2, the `ROUTE` property on a net contains its physical routing structure. By default, Vivado uses relative wire names (no tile information) when building these `ROUTE` strings. This is problematic for external tools because it can lead to wire ambiguities. Specifically, it is possible for a given wire in a Xilinx FPGA to connect to more than one wire of the same name. The `BRAM_L_X6Y170/BRAM_CASCOUT_ADDRBWRADDRU6` wire in the Artix7 part `xc7a100tcsg324-3` is an example. This wire connects to two different wires named `BRAM_ADDRBWRADDRU6` through PIP connections. The first wire is located in tile `BRAM_L_X6Y165` and the second wire is located in tile `BRAM_L_X6Y175`. Listing 4.8 shows how Vivado distinguishes these two connections within `ROUTE` strings.

Listing 4.8: Ambiguous ROUTE string example

```
// Tile BRAM_L_X6Y165
{ ... BRAM_CASCOUT_ADDRBWRADDRU6 BRAM_ADDRBWRADDRU6 ... }
// Tile BRAM_L_X6Y175
{ ... BRAM_CASCOUT_ADDRBWRADDRU6 <1>BRAM_ADDRBWRADDRU6 ... }
```

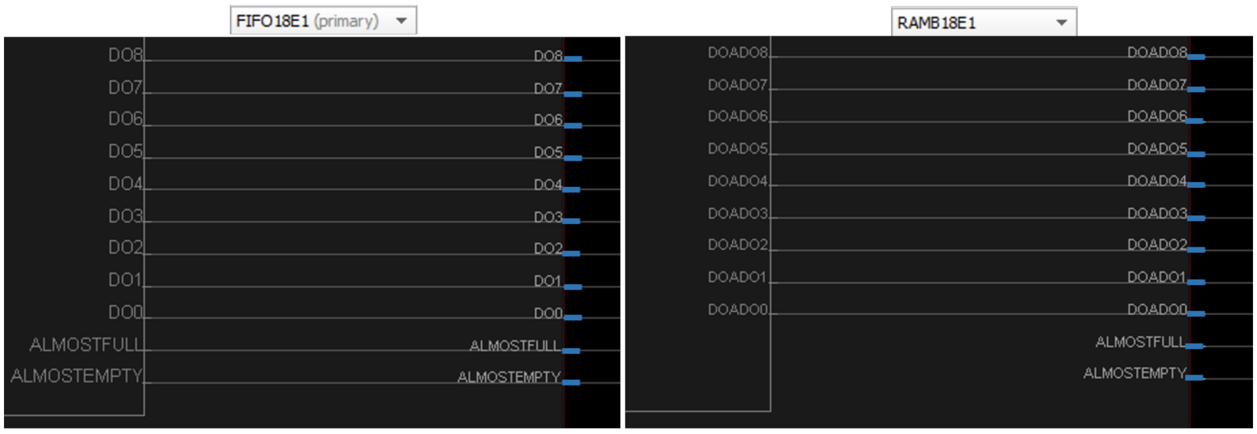
As can be seen, the token "<1>" is used to differentiate the two wires. Vivado most likely has an internal data structure that uses the token to choose the correct sink wire. External tools, however, don't have access to this data structure, meaning the only option is to guess which wire is actually taken (clearly not an acceptable solution). Ambiguous `ROUTE` strings are the primary reason why RSCPs use PIPs to represent routing, and two different external formats are required.

4.3.2 Alternate Site Pins

As discussed in subsection 3.2.2, the Vivado Tcl command `[get_site_pins -of $site]` does not return the correct result for alternate sites on Series7 devices[1]. An example of this behavior is shown in Figure 4.10. The site pin names clearly change in the GUI after the site is changed to be of type `RAMB18E1`, but the Tcl interface is not updated accordingly. RSCPs depend on exporting a correct set of site pins for each net, but the reported site pins will be incorrect for alternate sites if the above command is used.

VDI fixes this issue with an assumption about single-BEL sites in Series7 devices. As can be seen in Figure 4.10, each BEL pin within a single-BEL site connects to an identically named site pin. Using this assumption and a customized site pin getter, the VDI command `[tincr::write_rscp]` reports the correct site pins for each net connected to an alternate site. Of

[1]UltraScale devices always return the correct result



TCL call to [get_site_pins]:
RAMB18_X0Y32/DO0 RAMB18_X0Y32/DO1 ...

TCL call to [get_site_pins]:
*RAMB18_X0Y32/**DO0** RAMB18_X0Y32/**DO1** ...*

Figure 4.10: An example of [get_site_pins] returning incorrect results in Vivado

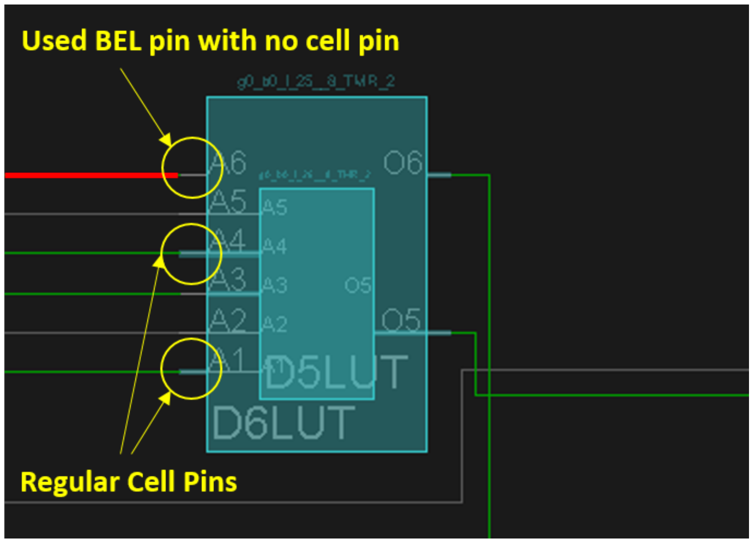


Figure 4.11: An example of VCC routing to an unused BEL pin (A6)

course, this is only valid if all alternate types with changing site pins have a single BEL. Through experimentation with several different designs, this appears to be true. As more tests are run on Series7 devices and designs, the conclusion will be further tested for validity.

4.3.3 VCC/GND BEL Pins

Most nets in a Vivado design connect to a set of cell pins, and are routed to the corresponding BEL pins of those cell pins. VCC and GND nets, however, can route directly to BEL pins that don't have a connecting cell pin. An example is shown in Figure 4.11. As the figure shows, VCC is routed to the `A6` pin of the `D6LUT`, but there is no cell pin mapped to `A6` (the input pins of the cell placed at the LUT have been mapped to `A1` and `A4`). The fact that VCC connects to the `A6` pin of this LUT is not represented in the logical netlist, and is purely an implementation detail of the design. Lacking this information is particularly challenging when developing routing algorithms in external tools. How will the algorithm know to route to VCC/GND BEL pins when they are not explicitly represented in the netlist? Unfortunately, these BEL pins cannot currently be determined with Vivado Tcl commands, and so are not included in RSCPs. It is up to external tools to understand the placement configurations that require VCC and GND to be routed to individual BEL pins.

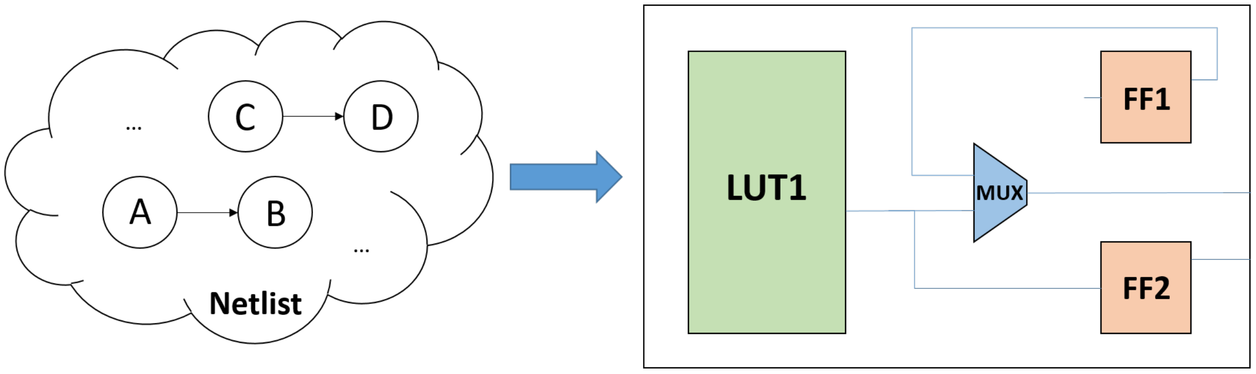


Figure 4.12: Routing Contention Example

4.3.4 SLICE Placement Order

When a cell is placed in Vivado using Tcl API calls, Vivado automatically configures the routing resources inside of the corresponding site based on the cell's connections. Because of this behavior it is possible to have a valid cell placement, but to place the cells in an order that causes an illegal routing configuration (i.e. a routing mux is optionally being used by a net, but is required by a different net of a cell that has just been placed). This primarily affects sites of type SLICEL and SLICEM, due to their more complex internal routing.

Figure 4.12 shows a simplified example of why routing contention can happen within SLICE sites. In this hypothetical scenario, the netlist in the left of the figure is placed onto the site in the right of the figure. Assume the cells of the netlist are placed in the following order: (1) `A` $\rightarrow$ LUT1, (2) `C` $\rightarrow$ FF1, and (3) `B` $\rightarrow$ FF2. Cell `A` will first be placed onto the LUT1 BEL which will use the mux to leave the site because its downhill cell (`B`) has not yet been placed. When cell `C` is placed on the FF1 BEL, an error will be thrown because the net connected to `C` needs to be routed outside of the site, but the routing mux is already in use and there is no other way to leave the site. At this point placement has failed. If instead the placement order is (1) `B` $\rightarrow$ FF2, (2) `A` $\rightarrow$ LUT1, and (3) `C` $\rightarrow$ FF1, no routing contention will happen because the site mux will be available once cell `C` is placed.

For Series7 devices, experimentation has shown that the proper placement order for groups of cells mapped to SLICE sites is as shown in Figure 4.13. The D6LUT needs to be placed first for distributed memory cells. If the required placement order is not followed when recreating a design, internal routing conflicts will occur that Vivado will be unable to resolve. It is the responsibility of

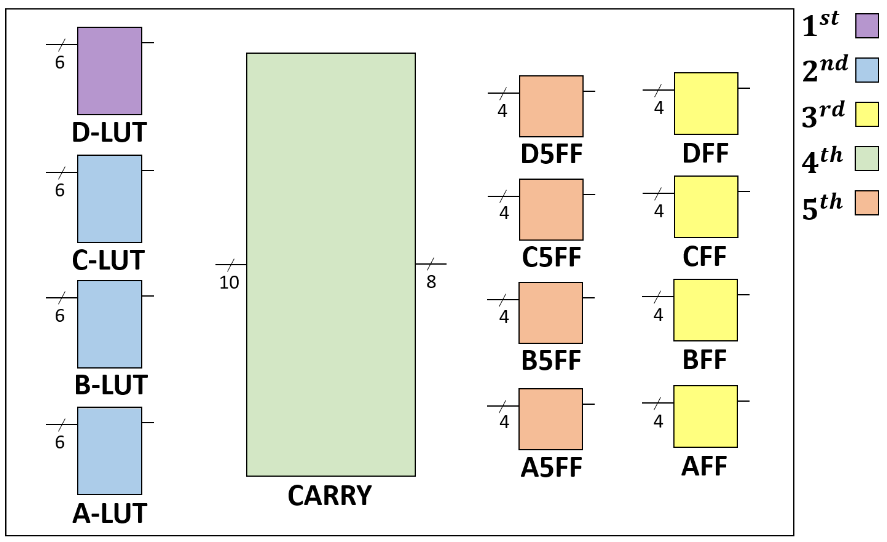


Figure 4.13: Required placement order for Series7 SLICE sites. The figure shows a simplified representation of a SLICEL site.

external tools to sort the cells of a design in the proper order shown in Figure 4.13 when creating the *placement.rsc* file of a TCP.

It is important to note that the SLICE architecture of UltraScale devices is significantly different than the Series7. Specifically, SLICE sites in UltraScale have 8 LUTs (H-A), 16 FFs, and a CARRY8 instead of a CARRY4. In initial testing, it appears that the placement order for UltraScale designs is less strict. The only requirement is that the H6LUT (top-most LUT in the SLICE) be placed first for distributed memory.

4.3.5 Macro Placement

As described in subsection 3.4.4 fully-flattened Vivado netlists may include macro primitives. Section 4.1 describes how macros are exported from Vivado in RSCP files. Importing macro primitives back into Vivado using TCP files, however, is a little more complicated. This is because internal cells to a macro cannot be placed with XDC Tcl commands (setting the property of an internal cell is an unauthorized operation in Vivado). Therefore, external tools have two options when generating a TCP with macros:

1. Before creating the TCP, completely flatten the design netlist so that all macros are completely removed from the design.

2. Support relatively placed macros (RPM), and output the placement information for just the RPM (not the internal cells) to the TCP. RPMs are macro cells that are assigned an anchor BEL placement location, and internal cells to the macro are placed *relative* to the anchor.

In general, the easier of the two solutions is to flatten the design netlist completely, but either solution is valid with VDI.

4.3.6 LUT Routethroughs

As described in subsection 4.1.6, the `CONFIG.EQN` property on a LUT BEL can be used to identity it as routethrough. On design export, all LUTs in a design whose configuration equation matches that of a routethrough are marked in the generated RSCP. It would stand to reason that when importing a design back into Vivado, routethroughs could be recreated by setting the corresponding `CONFIG.EQN` of the BEL. `CONFIG.EQN`, however, is a **read-only** property, meaning the value can be queried but not modified. Explicitly configuring a BEL's properties using the Tcl command `[set_property]` will throw an error in Vivado. These properties can only be modified internally during the placement or routing stage of implementation. Due to these limitations, it is currently not possible to configure a BEL LUT as a routethrough in Vivado.

Therefore, it is the responsibility of external tools to replace all LUT routethroughs in a design with a Xilinx `LUT1` cell before generating a TCP. The general method for this is shown in Figure 4.14.

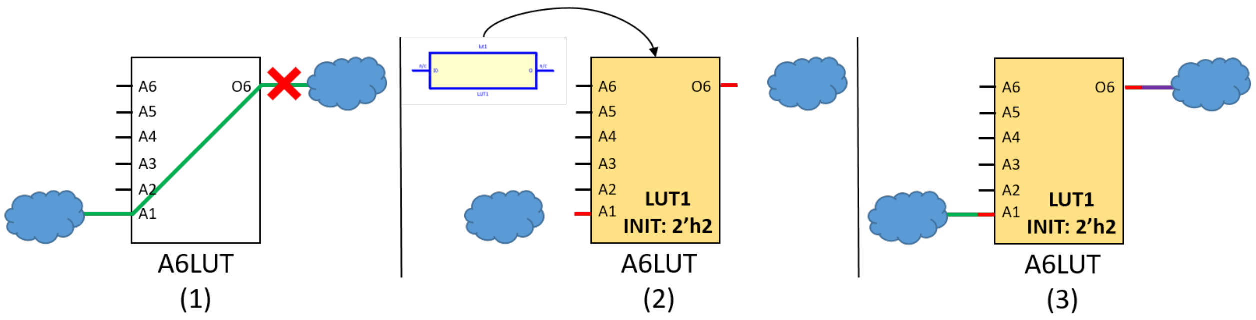


Figure 4.14: Visualization for how to replace a LUT routethrough with a LUT1 Xilinx cell.

First, disconnect the routethrough net from all downhill sink cell pins. Next, create a new `LUT1` Xilinx library cell with a unique name and set the cell's `INIT` property to "2'h2" (this configures the cell to be a passthrough LUT). Place the cell onto the BEL that was formerly being used as a routethrough, and map the `I0` input pin of the cell to the corresponding routethrough BEL pin (`A1` in the case of the example). Rewire the netlist and connect the original routethrough net to pin `I0` of the new cell. The final step is to create a new net with a unique name, and connect it to the output pin of the new cell and the downhill logic disconnected earlier. Once this is complete, the netlist now explicitly represents the routethrough, and valid TCPs can be generated.

4.3.7 Routing Differential Pairs

Xilinx devices support differential input signals for design input. An example differential pairing is shown in Figure 4.15. The net highlighted in white connects the two differential signals

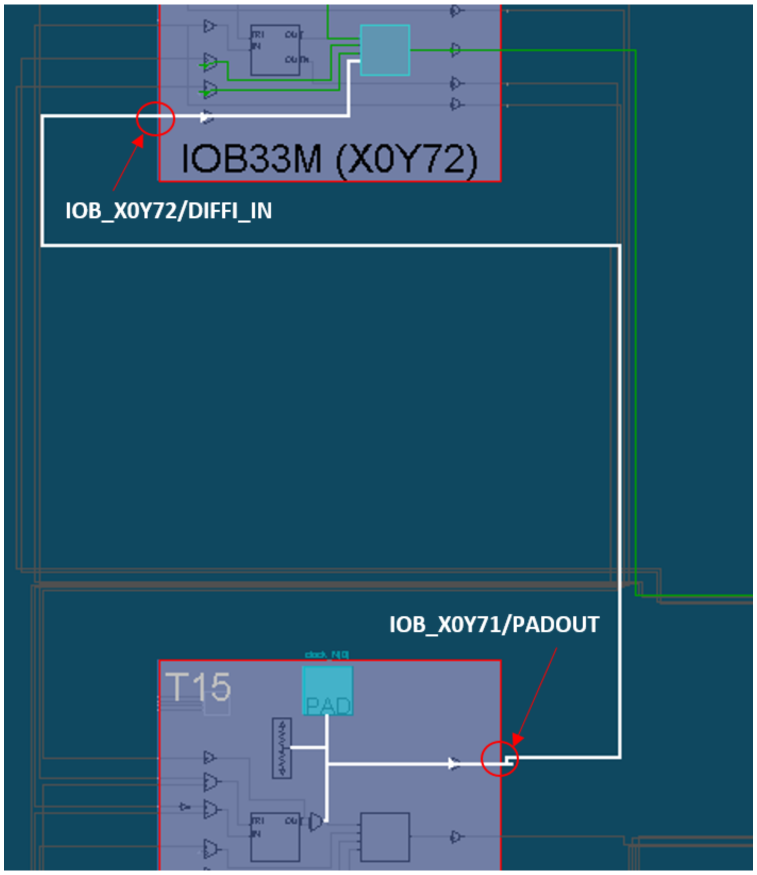


Figure 4.15: Differential Pair Net

together for Xilinx to resolve to a single value. As can be seen, the physical route of the net is very simple, with only a single PIP connection. As a TCP is being imported, however, Vivado is unable to correctly process the `ROUTE` strings for these nets. Specifically, when the `ROUTE` string for a differential net is specified, the following error message is given: "`ERROR: [Designutils 20-949] No driver found on net netname`".

The function `[tincr::read_tcp]` provides a workaround by first routing all other nets in the design. Once all other nets have been routed, the Tcl code shown in Listing 4.9 is run. The differential nets are identified using the Tcl command on line 1, and each net is individually routed by Vivado's router. Since there is only one possible route between the source and the sink pin, the differential nets will be routed correctly. The downfall to this approach is that it can take up to 1 minute to finish routing all differential nets. It is important to note that this is only an issue for Series7 devices, and is not needed for UltraScale.

Listing 4.9: Tcl script to route differential nets in Vivado

```
1 set diff [get_nets -of [get_ports] -filter {ROUTE_STATUS != INTRASITE} -quiet]
2
3 if {[llength $diff] > 0 } {
4    route_design -quiet -nets $diff
5 }
```

4.4 Why Two Design Checkpoint Formats?

So far this chapter has described in great detail RSCPs, TCPs, and the associated Vivado Tcl challenges with both. There is, however, one more question to consider: "Why can't a single design format be used for external tools *and* Vivado"? The purpose of this section is to give insight into four reasons why both RSCPs and TCPs are required with VDI.

1. Some Vivado design representations cannot be used with external tools. This is best shown with Vivado `ROUTE` strings, which are used to represent the physical route of a net. Due to the ambiguities of `ROUTE` strings as described in subsection 4.3.1, they are not suitable for design export (PIPs must be used instead). However, `ROUTE` strings are required when importing a

design back into Vivado (subsection 4.2.2). This indicates that a different format is needed for exporting and importing the physical structure of a net.

2. When an external design is imported into Vivado, some physical implementation details of the design are automatically configured. For example, site PIPs, static source LUTs, and intrasite nets are all configured during placement import based on how the cells of a design are placed. These design aspects do not need to be explicitly represented in a TCP. External tools don't have the luxury of automatic configuration, and so these details do need to be explicitly represented in a RSCP. Otherwise, a design could only be partially represented.

3. RSCPs and TCPs are designed for different purposes. TCPs are designed to import a design into Vivado as **quickly as possible**. The XDC format allows for this due to the dedicated Tcl command `[read_xdc]`. RSCPs are designed to be **easily parsable**, and use a simple token-based scheme with whitespace separated values.

4. Vivado offers a native checkpoint format (DCP) that can be used to save and restore the progress of an implemented design. Due to this available checkpoint format, there is no reason to support importing RSCPs directly back into Vivado. Users that want this capability can use DCPs, allowing RSCP checkpoints to be more customized.

4.5 Conclusion

VDI is the first open-source interface capable of completely representing Vivado designs at the subsite level (i.e. at the level of cells and BELs). Two different external design formats are specified by VDI. RapidSmith Checkpoints represent designs exported from Vivado, and Tincr Checkpoints represent designs that can be loaded back into Vivado. Several Vivado Tcl issues have been identified, and resolved to provide a complete import/export solution. Also, the required steps for external tools to create valid TCPs have been laid out. The hope is that researchers can use the external representation of Vivado designs presented in this chapter to implement CAD tools in their language of choice. A CAD tool framework based on VDI is presented in the next chapter.

CHAPTER 5. INTEGRATING RAPIDSMITH2 AND VDI

VDI enables the creation of custom CAD tools capable of targeting Vivado devices. However, there are variety of challenges when working with devices and designs produced from Vivado Tcl commands (reference section 4.3). A framework that can abstract the low-level details and challenges of Xilinx FPGAs away from CAD tool developers is valuable. RapidSmith2 (introduced in subsection 2.6.1) has been updated and integrated with VDI to provide an open-source and intuitive way of manipulating Vivado designs. Specifically, RapidSmith2 now supports importing RSCPs and exporting TCPs with the design flows shown in Figure 5.1. All possible design flows for Vivado **Series7 and UltraScale designs** are supported.

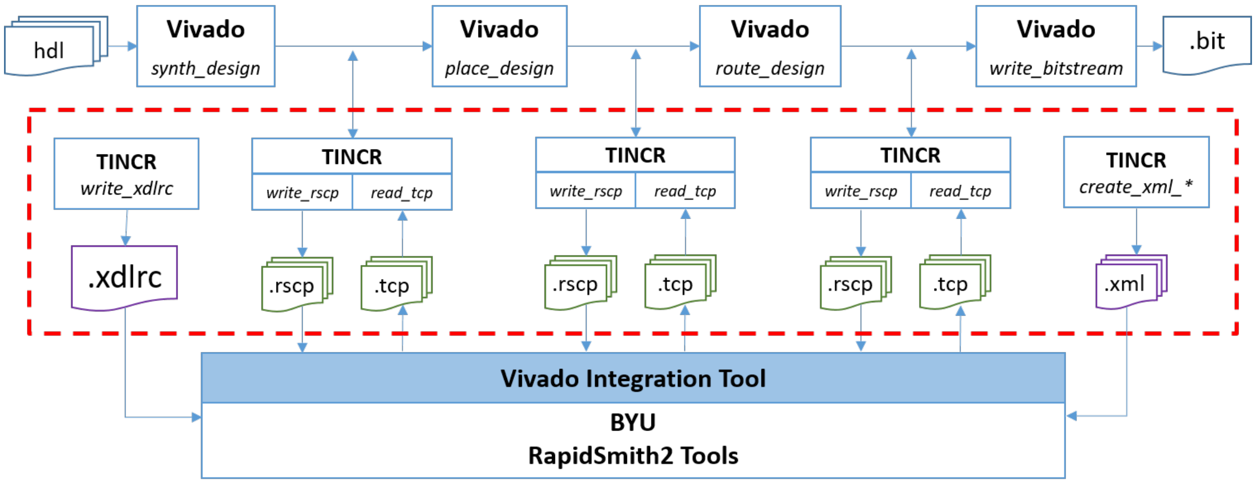


Figure 5.1: RapidSmith2 Supported Design Flows (the red box represents VDI)

RapidSmith2 is publicly available for download and collaboration at `https://github.com/byuccl/RapidSmith2`, and is introduced in greater detail in Appendix A. It is encouraged for the reader to review this appendix to gain a general understanding of the available data structures and APIs. This chapter assumes a working knowledge of RapidSmith2 and instead focuses on

VDI integration. Several new interface classes have been added to support importing RSCPs and exporting TCPs:

- **VivadoInterface.java**: Top-level class to parse RSCPs and generate TCPs.
- **EdifInterface.java**: Class to parse and generate EDIF files
- **XdcPlacementInterface.java**: Class to parse and generate placement files
- **XdcRoutingInterface.java**: Class to parse and generate routing files

The first two sections of this chapter discuss at a high-level how these classes are used to convert RSCPs into RapidSmith2 `CellDesign` objects, and how `CellDesigns` are converted back into TCPs. The final section of this chapter lists the contributions to RapidSmith2 related to the work in this thesis.

5.1 Importing RSCPs

RSCPs are imported into RapidSmith2 by the function `VivadoInterface::load_tcp()`. Listing 5.1 shows the order in which files of a RSCP are processed using the interface classes introduced above. The actual Java commands for each step of the process are given. As can be seen, the first step is to augment the existing `CellLibrary` object with the macros included in the *macros.xml* file. This allows the `CellLibrary` to have a complete picture of all primitives in a design when parsing the EDIF netlist in step 2. After parsing the EDIF, a RapidSmith2 `CellDesign` is created. Once the design is created, the XDC constraints found in *contraints.xdc* are loaded into the `CellDesign` in the form of `XdcConstraint` objects. The final steps are to apply the placement and routing information to the `CellDesign` by parsing the *placement.rsc* and *routing.rsc* respectively. The following subsections detail important aspects of importing RSCP files. Not all steps shown in Listing 5.1 are discussed below.

Listing 5.1: Steps to parsing a RSCP

```java
public void load_tcp (String rscp, Device device, CellLibrary libCells) {
  // 1.) Add macro cells found in "macros.xml" to the cell library
  libCells.loadMacroXML(Paths.get(rscp, "macros.xml"));
```

```

// 2.) Parse the EDIF and create the RS2 netlist
String edifFile = Paths.get(rscp, "netlist.edf").toString();
CellDesign design = EdifInterface.parseEdif(edifFile, libCells);

// 3.) Parse the XDC constraints
parseConstraintsXDC(design, Paths.get(rscp, "constraints.rsc").toString());

// 4.) Apply the placement information to the design
String placementFile = Paths.get(rscp, "placement.rsc").toString();
XdcPlacementInterface pI = new XdcPlacementInterface(design, device);
pI.parsePlacementXDC(placementFile);

// 5.) Apply the routing information to the design
String routingFile = Paths.get(rscp, "routing.rsc").toString();
XdcRoutingInterface rI = new XdcRoutingInterface(design, device,
    pI.getPinMap());
rI.parseRoutingXDC(routingFile); }
```

5.1.1 Approach to Parsing RSCP Files

Most files in a RSCP are whitespace delimited to make parsing very easy. The general approach to parsing these files in RapidSmith2 is shown in Listing 5.2. As can be seen, a RSCP file is read line by line, parsing the line using the Java regular expression `\\s+`, and then processing the individual tokens. It is important to note that the parsing function uses a `Pattern` object and not the built in Java `String` function `split()`. The reason for this is that `split()` internally creates and compiles a new `Pattern` every time it is called. Using a single `Pattern` instead is a minor optimization to design import. A *try-with-resources* block is used to guarantee that the file stream is closed correctly even if an exception occurs within the code. The *constraints.xdc*, *placement.rsc*, and *routing.rsc* files are parsed in this manner.

Listing 5.2: General skeleton for parsing space-separated RSCP files

```
public void parse (File file) {
   // compile the whitespace pattern
   Pattern whitespacePattern = Pattern.compile("\\s+");

   // try-with-resources to guarantee no resource leakage
   try (BufferedReader br = new BufferedReader(new FileReader(file))) {
      String line;
      while ((line = br.readLine()) != null) {
         String[] toks = whitespacePattern.split(line);
         // process tokens
      }
   }
```

5.1.2 Netlist.edf

The BYU EDIF tools [20] offer a comprehensive set of APIs that allow EDIF netlists to be parsed into equivalent Java data structures where they can be analyzed and modified. RapidSmith2 uses this tool to parse the *netlist.edf* file generated from Vivado. Specifically, the *netlist.edf* is first loaded into BYU EDIF data structures. The BYU EDIF representation is then translated into a RapidSmith2 `CellDesign` object using the `EdifInterface.java` class as shown on line 7 of Listing 5.1. For the most part this translation is straightforward, but there are a few noticeable exceptions that are listed below.

- To match the physical representation of VCC and GND nets within a RSCP as described in subsection 4.1.6, all VCC nets are combined into a single VCC net while translating the EDIF to a `CellDesign`. The same applies for GND nets. The API calls `CellDesign::getVccNet()` and `CellDesign::getGndNet()` can be used to obtain a handle to each.

- EDIF files represent top-level bus ports using the naming convention "port[7:0]." RapidSmith2 breaks bus ports into individual ports (i.e. "port[0]" - "port[7]").

- Vivado EDIF files can contain ports with offsetting indexes (i.e. “port[3:2]”). BYU EDIF tools do not represent the start and end index of offsetting ports, so RapidSmith2 must determine them during the translation. To do this, the name of bus ports are parsed using the Java regular expression ”`{(.*)\\[.+:(.+)\\]}`”. The first group of the regular expression matches the relative port name and the second group matches the port’s starting index. Using this information RapidSmith2 can correctly recreate the ports in a `CellDesign` (“port[3]” and “port[2]” for the example above).

The `EdifInterface.java` class has only been tested with EDIF files produced from Vivado, and is not guaranteed to work with other EDIF sources.

5.1.3 Routing.rsc

Due to the complex routing structure of a Xilinx FPGA, routing is the most difficult portion of a Vivado design to recreate in external tools. RSCPs represent this complex routing via *routing.rsc* files. As the example *routing.rsc* file given in Listing 4.5 demonstrates, there are many components to a routed FPGA design. RapidSmith2 processes these many components with the newly created `XdcRoutingInterface.java` class, and converts them into equivalent RapidSmith2 data structures. A detailed description of the routing import algorithm will not be given here. Rather, the most important routing concepts are highlighted, followed by a description of the algorithm used to create a routing tree based on the used PIPs of a net. Appendix section A.4 goes into greater detail about routing in RapidSmith2.

- The used site PIPs of each site are parsed and stored in the top-level `CellDesign`. The function `CellDesign::getUsedSitePipsAtSite(site)` can be used to retrieve the used PIPs for a given site. During routing import, these PIPs are used when reconstructing the *intrasite* portions of a net.

- BEL routethroughs in a design are stored into corresponding `BelRoutethrough` objects. A `BelRoutethrough` contains the BEL, input pin, and output pin for the corresponding routethrough. Researchers can use this information in their CAD Tools when modifying a design. Similarly, all static source BELs are recorded in a `List`.

- While recreating **fully-routed designs**, RapidSmith2 can recognize the VCC/GND BEL pin issue described in subsection 4.3.3. As mentioned in that section, these BEL pins are not represented in the logical netlist. To support a more complete netlist view, the routing importer creates a new cell pin for each discovered VCC/GND BEL pin. These cell pins, called `PseudoCellPins`, are added to the global VCC/GND net, attached to the cell placed at the corresponding BEL, and then mapped to the BEL pin. `PseudoCellPins` are described in more detail in section A.2.1.

- The intersite route status of each net is computed during routing import. Possible values include `FULLY_ROUTED` (all site pins are routed to), `PARTIALLY_ROUTED` (some but not all site pins are routed to), and `UNROUTED` (no site pins are routed to).

- As described in subsection 4.1.6, RSCPs use PIPs to represent routing. Vivado PIP names can be in one of three forms: "`tilename/wire1->>wire2`", "`tilename/wire1->wire2`" or "`tilename/wire1<<->>wire2`." The routing import algorithm uses the Java regular expression "`(.*)/.*\\.([^<]*)((?:<<)?->>?)(.*)`" to isolate the tile, source wire, and sink wire of a PIP in each of these formats.

Route Reconstruction Algorithm

As described in subsection 4.1.6, the physical structure of each net in a RSCP is represented by a list of used device PIPs. This PIP list, however, is not immediately useful to external tools because it does not represent branching wires. The route of a net is more accurately represented with a tree structure where branches in the route can be explicitly represented. RapidSmith2 uses `RouteTree` objects to represent the tree structure of nets. A detailed description of routing in RapidSmith2, including the structure and API of `RouteTrees`, is given in section A.4. The reader is encouraged to review and understand this material before continuing.

Algorithm 3 shows the basic pseudocode for converting a list of PIPs into a `RouteTree` object in RapidSmith2. The inputs to the procedure include (1) a net to route, (2) the start wire of the route, and (3) the set of used PIPs for the net (from the RSCP). As can be seen, the algorithm is a queue-based, breadth-first search of the device beginning at the specified start wire. During each iteration of the while loop starting at line 6, a wire (and its corresponding `RouteTree`) is removed

Algorithm 3 Algorithm to recontructing routing in RapidSmith2

```
1: procedure RECREATE_ROUTING_NETWORK($net, startWire, usedPipSet$)
2:     $searchQueue \leftarrow \varnothing$                                ▷ Initialize the search
3:     $visitedSet \leftarrow \varnothing$
4:     $start \leftarrow$ createRouteTree($startWire$)
5:     $searchQueue$.add($start$)
6:     while $searchQueue$ is not empty do             ▷ Breadth-first search through device wires
7:         $currentRoute \leftarrow searchQueue$.dequeue()
8:         $currentWire \leftarrow currentRoute$.getWire()
9:         for each $connection \in currentWire$.getDownhillWireConnections() do
10:            $sinkWire \leftarrow connection$.getSinkWire()
11:            if $sinkWire \in visitedSet$ then                ▷ Skip wires we have already visited
12:                continue
13:            end if
14:            $isNonPip \leftarrow connection$.isPip()==0
15:            $isUsedPip \leftarrow connection$.isPip() and ($connection$.getPip() $\in usedPipSet$)
16:            if $isNonPip$ or $isUsedPip$ then
17:                $nextRoute \leftarrow$ createRouteTree($sinkWire$)
18:                $currentRoute$.addConnection($nextRoute$)
19:                $searchQueue$.enqueue($nextRoute$)
20:                $visited$.add($sinkWire$)
21:            end if
22:        end for
23:    end while
24:    $net$.addRouteTree($start$)                    ▷ After the search, attach the route to the net
25: end procedure
```

from the queue and each of its downhill connections are examined. If a downhill connection is either (a) a non-PIP connection in RapidSmith2 or (b) a PIP connection **AND** the corresponding PIP is a member of the used PIP set, then the sink wire of the connection is deemed to be part of the route. A new `RouteTree` is created based on the sink wire, connected to the current `RouteTree`, and then added to the queue to be examined later. This process is repeated until the queue is empty and the complete route has been constructed. An identical algorithm can be used to route *intrasite* nets by using the enabled site PIPs of the corresponding site.

5.2 Exporting TCPs

After a CAD algorithm is run in RapidSmith2, the modified design can be re-imported back into Vivado by generating a TCP. The RapidSmith2 function `VivadoInterface::write_tcp()`

can be used to generate a TCP from a `CellDesign`. Listing 5.3 shows the internal steps of `VivadoInterface::write_tcp()` with the corresponding Java code for each step. A detailed description for each step of the TCP generation process is not given here because the steps are fairly straightforward. The user is referred to the RapidSmith2 source code for more details. Instead, only a few key aspects of TCP export are presented below.

- As described in subsection 4.3.6, LUT routethroughs need be replaced with `LUT1` library primitives to create a valid TCP. Before exporting any information to a TCP, all BEL routethrough connections in RapidSmith2 are replaced with `LUT1` cells using the method shown in Figure 4.14. Specifically, routethroughs are determined by iterating over a net's `RouteTree` and looking for BEL routethrough connections. The resulting netlist will be structurally different, but functionally equivalent.

- The BYU EDIF tools are also used to convert a RapidSmith2 `CellDesign` into a *netlist.edf*. As described in subsection 4.3.5, macro cells need to be flattened before generating a TCP. When an EDIF file is created using the BYU EDIF tools, RapidSmith2 macro cells are ignored, and only the internal cells are included in the resulting netlist. As long as the internal cell properties are set correctly, this modification results in a functionally equivalent netlist.

- As described in subsection 4.3.4, cells mapped to Series7 SLICEL or SLICEM sites need to be placed in a very specific order to prevent routing conflicts. Figure 4.13 shows the required placement order. Before generating a *placement.xdc*, RapidSmith2 sorts the cells of a design using a bin sorting algorithm. Specifically, every different color shown in Figure 4.13 represents a different bin. The bins are concatenated together in the correct order, and then the *placement.xdc* is generated. Sorting the cells using a bin placement algorithm is linear in complexity and scales well with the size of a design.

- The `RouteTrees` of all routed nets in a design are converted to Vivado `ROUTE` properties. To prevent any wire ambiguity, tile names are included for each wire in the route. Only source wires of PIP connections are included in the final `ROUTE` string.

Listing 5.3: Steps to create a TCP in RapidSmith2

```java
public void writeTCP(String tcpDirectory, CellDesign design, Device device) //
      1.) Replace routethroughs with LUT1 cells
      LutRoutethroughInserter inserter = new LutRoutethroughInserter(design);
      inserter.execute();

      // 2.) Write EDIF netlist
      String edifOut = Paths.get(tcpDirectory, "netlist.edf").toString();
      EdifInterface.writeEdif(edifOut, design);

      // 3.) write constraints.xdc
      String xdcOut = Paths.get(tcpDirectory, "constraints.xdc").toString()
      writeConstraintsXdc(design, xdcOut);

      // 4.) Write placement.xdc
      String placementOut = Paths.get(tcpDirectory,
          "placement.xdc").toString();
      XdcPlacementInterface pI = new XdcPlacementInterface(design, device);
      pI.writePlacementXDC(placementOut);

      // 5.) Write routing.xdc
      String routingOut = Paths.get(tcpDirectory, "routing.xdc").toString();
      XdcRoutingInterface rI = new XdcRoutingInterface(design, device, null);
      rI.writeRoutingXDC(routingOut, design);
}
```

5.3 Contribution List

One of the major contributions of this thesis is the integration of RapidSmith2 with VDI and Vivado. However, there are a number of other contributions to RapidSmith2 due to the work presented in this thesis. Many of the contributions are documented in Appendix A, but all contributions are highlighted in this section as an organized list.

- **Port Cells**: Top-level port and cell objects are distinguished from one another in Vivado netlists. RapidSmith2 removes this distinction and treats top-level ports as cells. There are three types of port cells: `IPORT`, `OPORT`, `IOPORT` (one for each direction).

- **Site PIPs**: A list of used site PIPs for each site is now stored in the top-level `CellDesign` object of a RapidSmith2 netlist.

- **Macro Support**: Both Xilinx primitive and user-defined macros are supported in RapidSmith2 cell libraries and netlists. Custom macros can be added to a `CellLibrary` by using the command `CellLibrary.loadMacroXml()`.

- **BEL Routethrough Connection**: Routethrough connections included in the *familyInfo.xml* (described in section 3.2) are now created and added to the `Device` data structure. Intrasite routing algorithm can use these routethrough connections just like any other wire connection.

- **Configurable Cell Properties**: The configurable cell properties for each library cell in Vivado is now parsed and loaded into RapidSmith2 from the *cellLibrary.xml*. When a new `Cell` object is created in RapidSmith2, the cell will have the same default properties as are shown in Vivado.

- **Three-part Routing**: The physical routing of RapidSmith2 nets have been updated to the three-part representation described in subsection A.4.5. This makes it easier to distinguish between the intrasite and intersite portions of a net.

- **Extensible RouteTrees**: The original `RouteTree` class included with RapidSmith2 had an integer cost code for routing algorithms. `RouteTrees` have been updated to be extensible, allowing a routing algorithm to use whatever customized cost function desired.

- **Pseudo Cell Pins**: Support for `PseudoCellPins` have been added to resolve the issue described in subsection 4.3.3. In general, `PseudoCellPins` are cell pins that can be created at runtime and attached to any `Cell` object.

- **Package Pins**: A list of `PackagePin` objects have been added to the `Device` class. Specifically, a `PackagePin` contains mappings between BELs to device package pins and vice versa. Package pins that access the global clock routing network of the chip are marked so placement algorithms can place clock ports accordingly.

- **XDC Contraints**: XDC constraints on a Vivado design are now parsed and loaded into `XdcConstraint` objects. Users can view/modify these constraints as they run CAD algorithms on a design.

- **Dot File Visualization**: To help visualize design netlists and cell placements, a `DotFilePrinter` class has been added. The `DotFilePrinter` can create valid DOT files for a design netlist, a placed design, or placement information for a single site in the device. This can be useful for debugging CAD algorithms.

- **Vivado Console**: The `VivadoConsole` is a Java class capable of starting a new instance of Vivado, sending Tcl commands to the Vivado instance, and reading back the results through a Java API. This class is discussed more in the next chapter.

- **Device Info Parsing**: Device file generation has been updated to parse the information found in *deviceInfo.xml* files introduced in section 3.3.

- **Family Info and Cell Library Updates**: The original specification for the *cellLibrary.xml* and *familyInfo.xml* does not include many of the things introduced in chapter 3. The RapidSmith2 parsing logic for these files has been updated accordingly.

- **Public Release**: RapidSmith2 has been released publicly on GitHub at `https://github.com/byuccl/RapidSmith2`. The GitHub repository includes Travis-CI build testing and support for Docker containers.

- **Unit Test Framework**: A unit test framework based on JUnit5 has been added to RapidSmith2. Most logical design data structures (`Cells`, `CellNets`, etc.) have been well tested using these unit tests.

- **UltraScale Support**: UltraScale designs have been successfully imported into RapidSmith2. They have also been successfully exported from RapidSmith2 back into Vivado with error. This enables the creation of UltraScale CAD tools for the first time.
- **Example Programs**: An example Series7 simulated annealing placer has been created and added to the RapidSmith2 repository as a reference placer. Other sample programs have also been created including a design printer, import/export demonstration, and a basic A* router.

5.4 Conclusion

This chapter presents the integration of RapidSmith2 with VDI, providing a framework to create Vivado FPGA CAD tools for next generation devices. Specifically, RapidSmith2 now supports parsing designs from RSCPs and translating internal designs to TCPs. Using the available Java APIs, algorithms to analyze and modify Vivado designs can be easily written. This is especially true when compared against the Tcl CAD tool alternative. Designs targeting **Series7 and UltraScale** architectures are supported with this framework, but support for UltraScale+ devices is still underway. The testing process used to demonstrate the correctness of RapidSmith2 and VDI is presented in the next chapter.

CHAPTER 6. VDI TESTING

To show the validity of VDI, several different experiments and tests were run using RapidSmith2. Table 6.1 shows the benchmark designs used in these experiments. Most of the benchmarks were obtained from the open-cores project located at https://opencores.org/, and range in size and FPGA resource usage. It is important to have designs for both Series7 and UltraScale architectures which use SLICEs, BRAMs, DSPs, and IOBs to test the major components of a Xil-

Table 6.1: VDI Benchmarks

Benchmark Name	**Architecture Name**	**Cell Count**	**Net Count**	**SLICE Util.**	**BRAM Util.**	**DSP Util.**	**IOB Util.**
bramdsp	Series7	230	430	0.28%	1.11%	0.42%	23.33%
canny7	Series7	7445	11830	5.74%	12.96%	0%	9.52%
cordic7	Series7	2420	3362	1.91%	0%	0%	35.24%
count16	Series7	75	121	0.03%	0%	0%	16.67%
hilbert	Series7	432	664	0.32%	0%	0%	12.86%
leon3 [21]	Series7	12391	13461	15.16%	51.85%	1.67%	1.43%
shaTmr	Series7	68376	68451	84.63%	0%	0%	32.86%
simon	Series7	894	1224	0.96%	0.37%	0%	37.14%
ssd	Series7	59	80	0.04%	0%	0%	17.62%
superCounter	Series7	15832	15844	15.09%	0%	0%	1.43%
viterbi7	Series7	52865	55431	84.85%	0%	0%	3.33%
cannyU	UltraScale	7128	11287	3.09%	4.86%	0%	6.41%
cordicU	UltraScale	2350	4011	1.02%	0%	0%	23.72%
counterTmr	UltraScale	11507	16522	5.78%	0%	0%	26.28%
ethmac	UltraScale	5153	5675	2.91%	0.56%	0%	67.63%
flappy	UltraScale	4607	7088	3.00%	0%	0%	13.14%
msp430	UltraScale	3398	5295	1.86%	0%	0.09%	78.21%
reed	UltraScale	1256	1656	.30%	0%	0%	94.55%
sha	UltraScale	9477	9619	4.76%	0%	0%	28.53%
vga	UltraScale	1732	2419	0.67%	0.14%	0%	58.01%
viterbiU	UltraScale	7194	13796	3.88%	0%	0%	10.58%

inx FPGA. All benchmarks in the Series7 category target the Artix7 part `xc7a100tcsg324`, and all benchmarks in the UltraScale category target the Kintex UltraScale part `xcku025-ffva1156`.

This chapter presents the testing process for VDI and RapidSmith2. Section 6.1 and 6.2 describe three different tests used to verify RSCPs and TCPs. Section 6.3 describes a case study in which a Series7 simulated annealing placer is implemented in RapidSmith2. Section 6.4 discusses the performance of VDI import and export.

6.1 RSCP Import Testing

The `VivadoConsole` is a Java class included in the RapidSmith2 distribution that is capable of creating a new instance of Vivado, sending Tcl commands to the instance, and reading back the results of the command. This flow is demonstrated in Figure 6.1. Sending Tcl commands in Java is a very useful utility that can used for a variety of tasks including: (1) generating RSCPs directly from Java, (2) comparing a RapidSmith2 design with the original Vivado design, and (3) loading and analyzing Vivado checkpoints. It can even be used for bitstream generation with the Vivado command `[write_bitstream]`.

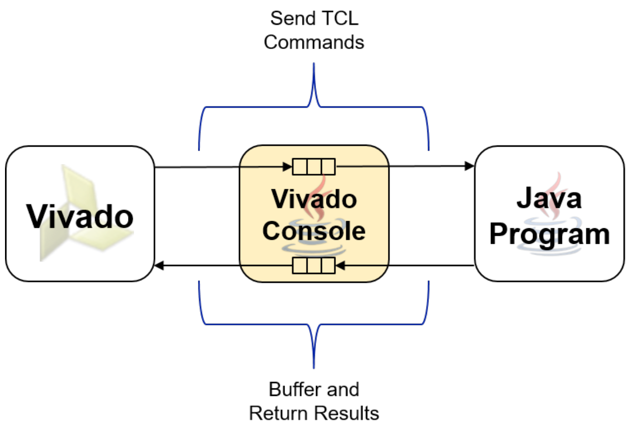


Figure 6.1: Vivado Console

The primary downfall to using the `VivadoConsole` is that the longer it runs, the slower the I/O between Java and the Vivado process becomes. For example, in one test the I/O rate between Java and Vivado started at around 10 MegaBytes/sec. After sending only 3000 Tcl commands, the I/O rate had diminished to 13 KiloBytes/sec. For this reason the `VivadoConsole` is not applicable for general design manipulations or loading designs directly into RapidSmith2. However, a few techniques can be used to avoid diminishing I/O rates. The first is to use batch commands instead of single commands. Instead of looping over each net of a design in RapidSmith2 and sending the Vivado command `[get_property ROUTE_STATUS $net]`, it is better to request this information for *all* nets at once using the Tcl command `[get_property ROUTE_STATUS $nets]`. The returned list can be parsed in Java and then processed without significantly affecting the I/O rate. It appears the I/O rate lowers based on the *number of commands sent*, and not the number of bytes returned from the command. Inevitably though, the I/O rate will decrease until it is too slow. Once this happens the second option is to restart the `VivadoConsole`, and the new Vivado instance will have high I/O rates again.

RapidSmith2 uses the `VivadoConsole` to verify that a design imported through a RSCP is still functionally equivalent to the original design in Vivado. Specifically, automated unit tests written in JUnit5 have been added to RapidSmith2 which operate as shown in Figure 6.2. The first step in the testing process is to load a design into both RapidSmith2 (through RSCPs) and Vivado (through DCPs, Vivado's native checkpoint format) in parallel. Information about the Vivado design is extracted through Tcl commands sent through the `VivadoConsole`, and compared to the RapidSmith2 representation. For example, the Tcl command `[get_bel -of ‘‘myCell’’]` will return the name of the BEL object that "myCell" is placed on in Vivado. This value is compared against where RapidSmith2 thinks "myCell" is placed, and an error thrown if there is a mismatch. All logical and physical aspects of a design are tested in a similar manner. To reduce the run time

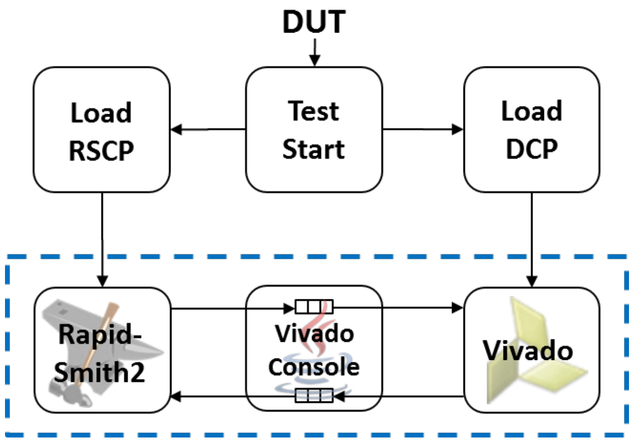


Figure 6.2: RapidSmith2 Design Input Testing Flow

of these tests, the two `VivadoConsole` tricks described above are employed. All benchmarks in Table 6.1 passed this design verification.

6.2 TCP Export Testing

When exporting a TCP from RapidSmith2, two things need to be tested: (1) the generated TCP can be successfully imported back into Vivado without errors, and (2) the design is functionally correct. An extensive number of designs for both Series7 and UltraScale devices were tested against criteria (1). The testing process is shown in Figure 6.3. As the figure shows, the test starts with a directory full of native Vivado checkpoint files (.dcp) for *fully-routed* designs. A Tcl script loads each test design into Vivado and generates a corresponding RSCP into another directory. A Java class (`TcpExportTest`) loads each RSCP and then immediately exports the design to a corresponding TCP. This step is important because RapidSmith2 performs several netlist modifications before exporting a TCP (described in section 5.2). Once a directory of TCPs has been generated, the `VivadoConsole` is used to load each TCP back into Vivado and perform two automated checks:

1. All cells in the design are placed. This is checked by using the Vivado Tcl command `[ get_cells -filter {STATUS == UNPLACED}]`. If the only returned cells are VCC and GND, then the test passes.

2. All nets in the design are *fully* routed. This is checked by using the Vivado Tcl command `[get_nets -filter ROUTE_STATUS != {FULLY_ROUTED}]`. The test passes if no nets are returned.

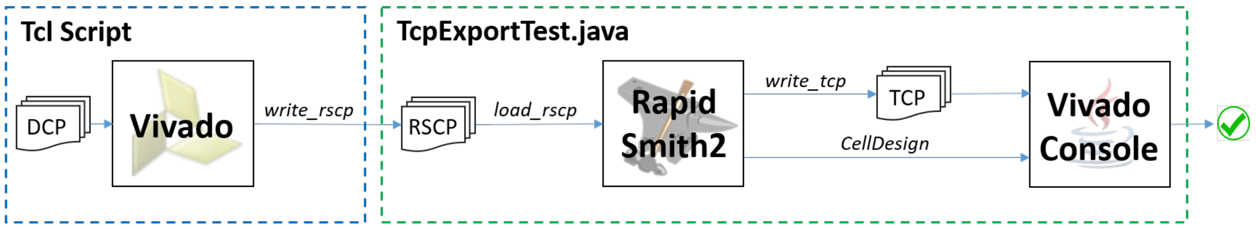


Figure 6.3: Test flow to verify that RapidSmith2 TCPs can be exported into Vivado without errors.

If a Vivado design passes both checks, the TCP is declared correct. All benchmarks listed in Table 6.1 passed TCP verification.

Criteria (2) was tested against a set of simple FPGA designs shown in Table 6.2. These designs are taken from the BYU course ECEn 320 - Digital System Design.

Table 6.2: Nexys4 Test Designs

Benchmarks	
Switch Even Detector	Seven Segment decoder
Seven Segment Timer	VGA controller
VGA Pong Animations	BRAM Character Generator
UART TX	UART RX
SRAM Controller	Frame Buffer

Figure 6.4 shows the on-board testing process. As can be seen, a design is first implemented and run on a Digilent Nexys4 DDR evaluation board (Artix7 FPGA). Once the design works on the evaluation board, it is exported from Vivado, imported into RapidSmith2, and directly exported from RapidSmith2 with no modifications. The resulting TCP is then imported back into Vivado for bitstream generation, and loaded onto the evaluation board again. It is important to note that before a bitstream can be generated from a TCP, the Vivado Tcl commands show in Listing 6.1 must be executed (it appears that Vivado needs to set internal implementation flags before generating a bitstream). If the circuit still works as expected on the board, the new bitstream is declared functionally equivalent to the original bitstream. All designs listed in Table 6.2 ran successfully on the

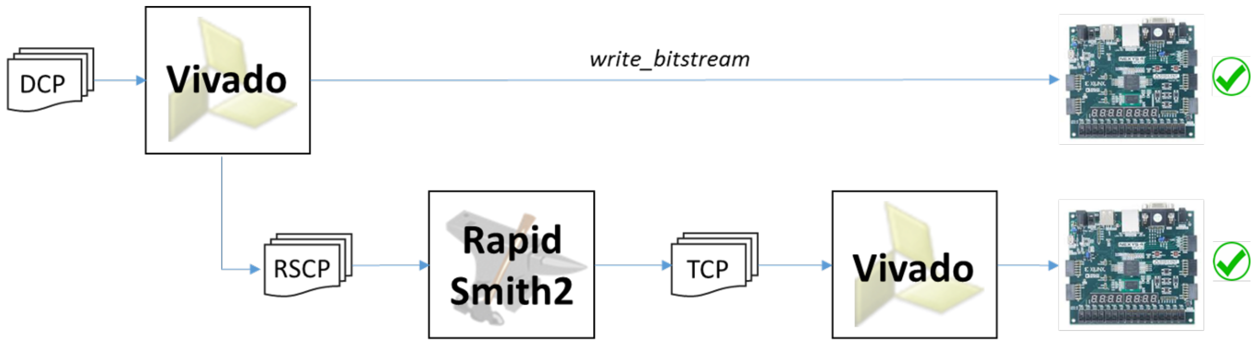


Figure 6.4: Test flow to verify that RapidSmith2 TCPs are still functionality equivalent to the original Vivado design.

Nexys4 after being passed through RapidSmith2, indicating that RapidSmith2 netlist modifications result in a functionally equivalent netlist.

Listing 6.1: Workaround to generate a TCP bitstream

```
lock_design -level Routing ; // Lock the design so it won't be modified
place_design ; // force set placement implementation flag
route_design ; // force set routing implementation flag
write_bitstream ; // generate bitstream
```

On-board hardware testing is a promising first step in verifying VDI and RapidSmith2, but future work will create a more complete verification by using simulation models and automated test benches.

6.3 Case Study: Series7 Simulated Annealing Placer

To demonstrate VDI, a simulated annealing site-level placer targeting Series7 devices has been implemented in RapidSmith2 as a case study. The purpose of this case study is four-fold.

1. Provide an example CAD tool that operates on Vivado designs through VDI.
2. Show that VDI contains useful and sufficient information for external frameworks to utilize.
3. Demonstrate the productivity advantages of using VDI with an external framework.
4. Demonstrate the complexities of writing CAD tools targeting real hardware.

Due to the lack of a working packer in RapidSmith2, a design is first packed in Vivado and given an initial placement. The placed design is exported to a RSCP, where it is translated into a RapidSmith2 `CellDesign`. Before running the placer, the design is completely randomized across the device to give an initial placement state. The simulated annealing algorithm shown in Appendix C takes the initial state and tries to find an optimal placement configuration. Once a final placement is determined, the design is re-imported into Vivado for routing and bitstream generation. Source code for the placer is available in the RapidSmith2 distribution repository under the `examples.placerDemo` package. The following subsections go into more detail about the case study goals as listed above. Preliminary results are also presented.

6.3.1 Utilizing VDI Information

One important goal of VDI is to export useful information about Xilinx devices and designs. The hope is that external frameworks can use the information to create more complete and efficient CAD tools. RapidSmith2's sample placer uses a variety of information exported through VDI including:

- **Compatible Types**: When doing a site-level placement, groups of cells can generally be placed onto more than one site type. For example, a group of cells targeting a `SLICEL` site can also target a `SLICEM` site. It is important for a site-level placer to understand which site types are compatible to create a complete list of candidate site placements. Compatible type information is obtained in RapidSmith2 with the function `Site::getCompatibleTypes()`.
- **Alternate Types**: Once a final placement is determined, the type of some sites may need to be changed before marking the placement as complete. For example, a Xilinx `BUFG` cell can be placed onto a `BUFGCTRL` site, but the site type must first be configured to its alternate `BUFG`. RapidSmith2 offers the functions `Site::getPossibleTypes()` to find all valid alternate types of a site, and `Site::setType()` to set a site to an alternate type.
- **Pin Mappings**: When a cell is placed, each cell pin is mapped to a corresponding BEL-pin. RSCPs include these mappings for each cell in a design. This is particularly useful for a placement algorithm because cells are constantly being swapped between different sites (and BELs). On design import, the sample placer caches the cell pin to BEL-pin mappings in a RSCP so that they can be restored before the final placed design is exported to Vivado.

Other important pieces of information that are used by the placer include the data structures, site types, and two-dimensional layout of a device.

6.3.2 Productivity Advantages

Besides using RapidSmith2 and VDI, the only open-source alternative to creating a Vivado placement algorithm is to use the Tcl interface and `Tincr`. While it would be theoretically possible to write a simulated annealing placer with the same structure of the one shown in Appendix C, it would be quite challenging. For one, Tcl does not natively support object oriented programming constructs. As the algorithm in Appendix C shows, there are several different Java class objects defined to represent parts of the algorithm at higher levels of abstractions. These classes would need to be mapped to Tcl `list` or `map` objects, which are the primary two data structures supported in Tcl. Other object oriented constructs such as inheritance and polymorphism are used in the Java algorithm to simplify the code as well.

A second challenge to writing a placer in Vivado's Tcl interface is handling the many Tcl issues introduced in chapter 3 and 4. For example, `IPAD` sites report an incorrect list of alternate types through the Tcl interface (subsection 3.2.2). These invalid alternate types need to be ignored which requires specialized code and can be error prone. A Tcl placer also needs to reconstruct parts of VDI that are not directly available with Vivado Tcl commands (such as compatible types). RapidSmith2's placement algorithm does not have to worry about these issues since they are handled within VDI.

The performance of a Java-based placer is significantly faster than an equivalent Tcl version. Java, a managed run-time system, eventually compiles a program to native machine code which can be run directly on hardware. Tcl is an interpreted language. This means the commands are never executed directly in hardware but rather by an interpreter program. Due to the abstractions and advantages listed above, it is clear that a Java-based framework for CAD algorithms is much preferable to Tcl.

6.3.3 Real Device Challenges

When creating a placement algorithm for *real* Xilinx devices, there are several special cases that need to be considered in order to generate a valid placement. These cases are listed below.

- `CARRY4` cells connected through a dedicated **carry chain** need to be placed vertically to one another. The same applies for `DSP48E1` cells. An example valid placement is shown in Figure 6.5. Series7 placement algorithms need to be careful to preserve this configuration while placing a design.

- Similarly, `BRAM` cells connected through cascade pins also need to be placed vertically to one another.

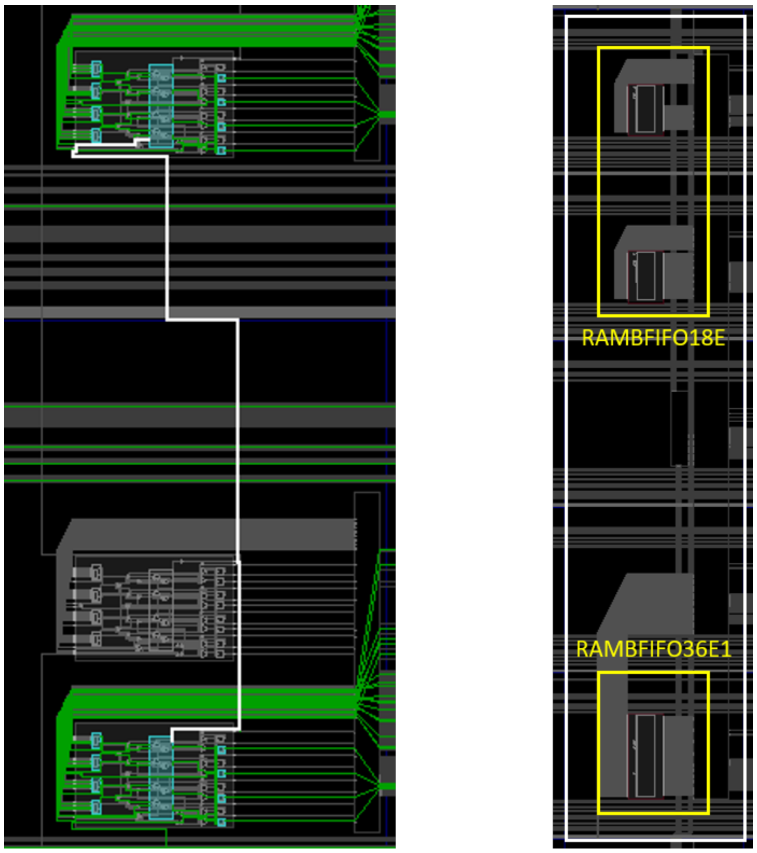


Figure 6.5: Example Carry Chain (left) and BRAM Tile (right) in Vivado.

- `BRAM36` and `BRAM18` cells cannot be placed in the same tile as one another. `BRAM` tiles require that either (a) the single RAMBFIFO36E1 site is used **or** (b) the two RAMBFIFO18E1 sites are used, not both. Figure 6.5 shows the layout of a `BRAM` tile. The most likely explanation is that sites in a `BRAM` share the same silicon and so cannot physically be used at the same time.

These special cases demonstrate one of the primary challenges to writing CAD tools that target commercial devices; they need to be handled and cannot be abstracted away. However, CAD tools that are forced to face these challenges are often more complete and realistic.

6.3.4 Interactive Mode

The RapidSmith2 sample placer can also be run in *interactive* mode. Instead of placing the entire design at once, interactive mode pauses the placer at a list of pre-defined annealing temperatures. During each pause, the current progress of the placer is exported to a Vivado instance using a `VivadoConsole` object, and displayed in the device browser. These visuals can be used to better understand and improve the annealing schedule of the algorithm. It also makes for an interesting demo.

6.3.5 Placer Results

The simulated annealing placer described so far in this section was implemented as a case study for VDI and RapidSmith2. It has not been extensively tested with a variety of benchmarks to ensure that it creates valid placement results for all possible designs. However, some preliminary placer results are available. Table 6.3 shows the results of RapidSmith2's placer vs. Vivado's

Table 6.3: Series7 Placer Results

Benchmark Name	**Swaps Per Second**	**Routed In Vivado?**	**RS2 Max Path Length**	**Vivado Max Path Length**
SSD	948,458	✓	14.304ns	14.047ns
UART TX	819,466	✓	7.124ns	7.183ns
FIR Filter	622,615	✓	4.977ns	5.220ns
DiffEq	238,711	✓	16.335	17.031
CORDIC	222,041	✓	6.211ns	5.554ns
Leon-3	119,806	✓	13.105ns	8.95ns

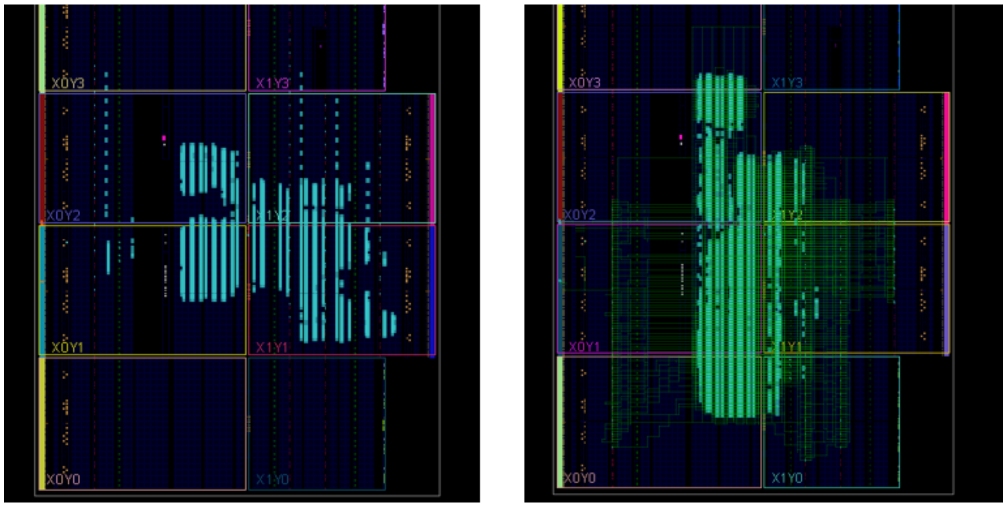


Figure 6.6: Leon-3 soft processor placed by Vivado's placer (left) and RapidSmith2's simulated annealing placer (right)

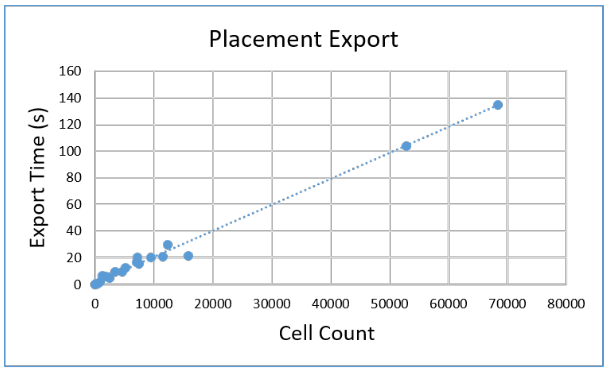
proprietary placer for a variety of benchmarks. Figure 6.6 compares the final placement solutions for the Leon-3 soft processor in Vivado's device browser.

6.4 VDI Performance

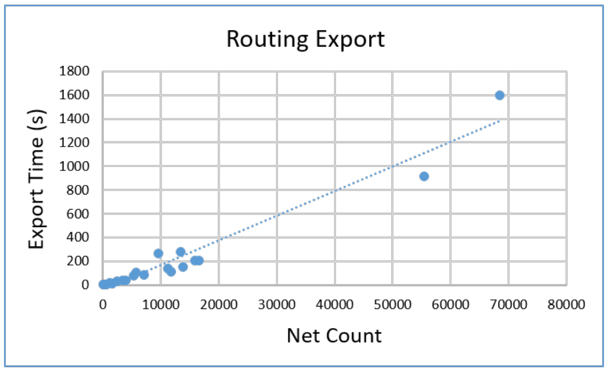
Although VDI focuses on functionality, an important aspect of any Xilinx design interface tool is performance. Table 6.4 shows the import and export times of VDI for all design benchmarks given in Table 6.1. As the table shows, Vivado import and export times are relatively slow. A design that uses 85% of an Artix7 part (*shaTmr*) takes almost 30 minutes to export from Vivado and 2.5 hours to import back into Vivado (keep in mind these numbers were obtained on a less-powerful machine). Routing takes much longer than placement, and contributes to approximately 90% of the total export time and 97% of the total import time across all benchmarks. The table also shows

Table 6.4: VDI Import/Export Times

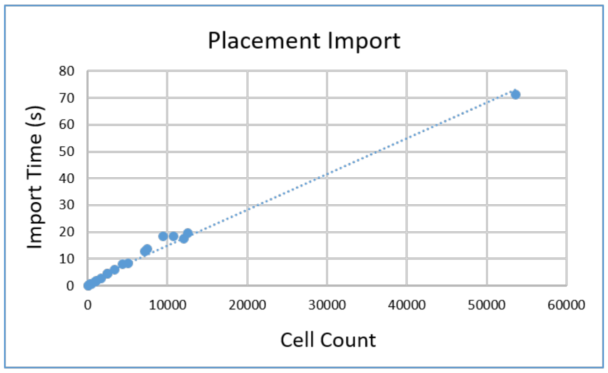
Benchmark Name	**Placement Export**	**Routing Export**	**Export Total**	**Placement Import**	**Routing Import**	**Import P+R**
bramdsp	0.66s	6.22s	8.33s	0.51s	4.63s	5.14s
canny7	15.66s	112.61s	130.12s	13.53s	131.165s	144.70s
cordic7	4.83s	35.24s	41.66s	4.35s	23.89s	28.24s
count16	0.15s	2.37s	3.96s	0.15s	0.69s	0.84s
hilbert	0.84s	7.17s	9.44s	0.67s	3.99s	4.66s
leon3	29.59s	280.45s	312.16s	19.63s	496.17s	515.80s
shaTmr	135.26s	1598.66s	1739.07s	324.43s	7785.35s	8110.79s
simon	2.12s	18.035s	21.63s	1.55s	11.62s	13.16s
ssd	0.15s	2.61s	4.25s	0.10s	0.82s	0.92s
superCounter	21.48s	203.20s	226.74s	17.51s	189.73s	207.24s
viterbi7	104.37s	917.16s	1024.73s	71.34s	4638.92s	4710.27s
cannyU	16.44s	140.05s	156.99s	12.65s	262.6s	275.25s
cordicU	5.65s	40.64s	47.88s	4.59s	59.52s	64.11s
counterTmr	20.87s	205.34s	228.24s	18.54s	380.44s	398.98s
ethmac	12.45s	103.66s	117.82s	8.28s	220.71	228.99s
flappy	9.54s	85.97s	97.25s	7.94s	165.11s	173.05s
msp430	9.76s	75.07s	87.26s	6.10s	170.57s	176.67s
reed	6.68s	13.93s	22.16s	1.79s	57.92s	59.71s
sha	20.04s	262.50s	284.60s	18.5047s	707.288s	725.79s
vga	5.85s	31.88s	39.64s	2.84s	59.42s	62.26s
viterbiU	20.46s	152.38s	174.82s	13.036s	275.533s	288.57s
	9.27%	**90.01%**		**2.7%**	**97.2%**	



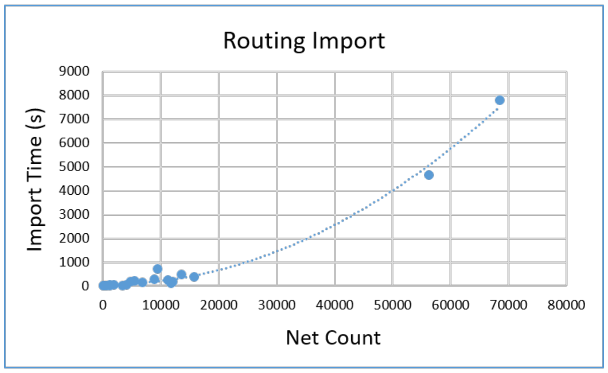
(a)



(b)



(c)



(d)

Figure 6.7: VDI Import/Export Trends

that importing a design back into Vivado is generally longer than exporting the same design from Vivado.

Figure 6.7 shows the performance trends of VDI based on the size of a design. Subfigures (a) and (c) plot the placement export and import times versus the number of cells. Subfigures (b) and (d) plot the routing export and import times versus the number of nets. Ideally, there would be more data points in the 20,000-40,000 count range, but finding open-source benchmarks that can fit on a device without using all the available I/O is difficult. These graphs demonstrate that all import/export times appear to be linear *except for* routing import. Routing import seems to be quadratic complexity and explains why design import is slower than design export. One hypothesis for the quadratic behavior is that Vivado performs design rule checks for the entire design after each physical constraint is imported. This suggests that after each wire is assigned to a net from a XDC file, the **entire routing structure** is re-checked for illegal configurations. These observations agree with the observations of Brad White made in [1]. Another possible reason may relate to the `VivadoConsole` diminishing I/O bandwidth discussed in section 6.1, but this is less likely.

In many use cases the design import/export time is not critical to the final application. For example, consider custom placement and routing algorithms that aim to increase the reliability of a FPGA circuit. The ultimate goal of these CAD tools is to physically implement a circuit so that it fails less often due to radiation and other effects. The amount of time it takes to import a placed and routed design back into Vivado is much less significant than a functionally correct (and hopefully more reliable) bitstream. A key aspect of VDI is that it makes tools such as this **possible** to implement in a friendly programming environment where there is no other alternative. It is also important to note that the time it takes to import a design *without routing* is significantly faster.

Although a variety of useful tools can be implemented through VDI, the observed slow import performance can lead to several design challenges. One such challenge is debugging. Because routing import times are slow, the productivity of CAD designers trying to debug their algorithm can decrease for large designs. The design cycle time may be too long while waiting for designs to be imported into Vivado and tested. Future work will focus on improving the performance of VDI to make it applicable for a wider range of CAD tools. Possible options to improve the performance are discussed in the next chapter.

6.5 Conclusion

Since VDI is a newly defined interface, it is important to verify that the interface is valid. This chapter has presented several methods and experiments that have been used to demonstrate that VDI can be used as a general purpose import/export method into Vivado. However, one major drawback of VDI is its performance. Section 6.4 shows several examples of long import and export times for Vivado designs. For the first iteration of VDI presented in this thesis, these times are acceptable because the main goal was simply to **create** a Vivado interface. Future work will focus on improving these design import and export times to make VDI applicable with a wider range of applications.

CHAPTER 7. CONCLUSION

This thesis introduces the Vivado Design Interface (VDI): an open-source interface to extract device and design specific information from Xilinx's Vivado tool suite. Chapter 3 describes the file formats defined by VDI to represent Vivado FPGA devices. Chapter 4 introduces RapidSmith Checkpoints (RSCP) and Tincr Checkpoints (TCP), the file formats used to export and import Vivado designs. Chapter 5 describes how the RapidSmith2 CAD tool framework was integrated with VDI to support Vivado. Chapter 6 demonstrates the many ways in which VDI was tested to verify it as a complete and accurate interface. This chapter concludes the thesis in three ways. It first includes a discussion on the sustainability of VDI and RapidSmith2 moving forward. It then presents two lists. The first is a list of contributions from this work, and why they are important. The second is a list of future work to improve VDI and RapidSmith2.

7.1 Sustainability

One of the major questions concerning this work is how sustainable VDI and RapidSmith2 are with future releases of Xilinx device families. Specifically, how will the Tcl interface and Xilinx device representation change with future versions of Vivado? Unfortunately, this question is impossible to answer without knowing how Vivado will be updated, but some insight can be given based on the transition from Series7 to UltraScale devices. While adding support for UltraScale, two notable changes between UltraScale and Series7 were observed: (1) UltraScale devices have several architecture differences (which are described in section 3.1), and (2) Vivado's Tcl interface is easier to work with (there are fewer Tcl bugs for UltraScale). These two observations resulted in several updates to `Tincr`, VDI, VSRT, and RapidSmith2, but ultimately all new device challenges were overcome to support an external CAD tool flow. It took an estimated one month to fully support UltraScale devices and designs while making changes to the four pieces of software previously mentioned. It is the opinion of this thesis, based on the experience with UltraScale, that

VDI, VSRT, and RapidSmith2 can be easily updated to support new Vivado device families for the foreseeable future.

7.2 Contributions

The work presented in this thesis is the first successful attempt to provide an open-source tool-flow that can export designs from Vivado, manipulate them with external CAD tools, and re-import valid design representations back into Vivado. The main contributions of this thesis are listed below:

- It provides insight into the many low-level details of a fully implemented Xilinx design, and how to best represent those details in an external format. It also documents how to successfully reconstruct a design in Vivado.
- It introduces the Vivado Design Interface (VDI), an extension of *Tincr*. VDI uses Vivado Tcl commands to define an organized framework for interfacing with Vivado designs and devices. Specifically, VDI allows users to extract complete design and device information from Vivado to be used with external tools. It also allows users to import modified designs back into Vivado.
- It describes the integration of VDI with RapidSmith2, an external CAD tool framework. This integration enables a variety of CAD tools to be written for Vivado designs.
- It demonstrates the correctness of VDI and RapidSmith2 through two methods of verification: (1) Java unit tests which verify that the RapidSmith2 design representation matches the corresponding Vivado design, and (2) On-board hardware tests which verify that designs are still functionally correct after being passed through RapidSmith2 and back into Vivado.
- It presents the implementation and results of a simulated-annealing placer that has been implemented in RapidSmith2.
- It introduces support for UltraScale devices in RapidSmith2. CAD tools targeting UltraScale devices can now be created, which was previously impossible.

- It documents the open-source release of RapidSmith2 at GitHub to allow FPGA CAD researchers across the country and world to develop Vivado CAD algorithms.

One other important contribution to note is that the work presented in this thesis has also been formatted into a conference paper that was accepted into Field Programmable Logic and Applications (FPL) 2017. The title of the FPL paper is "Vivado Design Interface: An Export/Import Capability for Vivado FPGA Designs," and presents many of the same concepts introduced in this thesis.

7.3 Future Work

RapidSmith2 and VDI provide a starting point for supporting CAD tools that can target Vivado devices. There are several updates that can be made to greatly enhance both in future revisions. A possible list of tasks is given here to suggest where the work in this thesis may lead.

- **Improve Design Import Times**: As discussed in section 6.4, VDI design import is very slow due to routing. A method to speed up the routing import would be valuable. One possible solution is to break a design into several sub-designs, each with their own individual placement and routing. Each sub-design can be imported into Vivado using out-of-context checkpoints, and then stitched together to create a complete design. Preliminary attempts to achieve this have been made using Vivado `PBlock` objects. The initial work can be found in the *tincr_checkpoint.tcl* file of the `Tincr` distribution.

- **TCP Testbench Verification**: The functionality of TCPs produced from RapidSmith2 have been tested by running a set of simple FPGA designs on a Nexys4 evaluation board. A more complete verification would be to create a Vivado simulation model of a design exported from RapidSmith2, and run it through an automated test bench that passed for the initial Vivado design. This would give more confidence that TCPs are functionally correct.

- **RSCP Compression**: For large designs, RSCPs can grow to be very large (100MB). To reduce the size of these checkpoints, compression can be applied. External tools can decompress the archive before parsing the relevant design information. A preliminary test has shown that a simple compression can be used to reduce the size of RSCPs by 10X.

- **General Hierarchy Support**: VDI and RapidSmith2 currently support fully-flattened Vivado netlists (macro primitives included). A more general support for hierarchy would be useful for CAD tools that want to preserve hierarchy in their tools.
- **UltraScale+ Support and Testing**: RapidSmith2 provides support for Series7 and UltraScale designs. Support for UltraScale+ is currently being developed, but integration and testing with RapidSmith2 is still required.
- **Complete CAD flow in RapidSmith2**: The simulated annealing placer discussed in this thesis was simply a case study on VDI and RapidSmith2. A general purpose and complete CAD flow for RapidSmith2 (including a packer, placer, and router) would be of great value to the research community. This is especially true if the CAD flow supports UltraScale designs.

The hope is that the work started in this thesis (and theses before) can be continued to create a robust and useful CAD tool flow with Vivado.

REFERENCES

[1] B. White. (2014) Tincr: Integrating custom cad tool frameworks with the xilinx vivado design suite. [Online]. Available: http://scholarsarchive.byu.edu/etd/4338/ ix, 6, 92

[2] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, ser. FPL '97. London, UK: Springer-Verlag, 1997, pp. 213–222. 2

[3] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz, "VTR 7.0: Next Generation Architecture and CAD System for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 2, pp. 6:1–6:30, June 2014. 2

[4] C. Wolf and M. Lasser, "Project icestorm," http://www.clifford.at/icestorm/. 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 *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications*, ser. FPL '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 349–355. [Online]. Available: http://dx.doi.org/10.1109/FPL.2011.69 2, 15

[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

[7] A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, 2011, pp. 228–235. 2

[8] A. Otero, E. de la Torre, and T. Riesgo, "Dreams: A tool for the design of dynamically reconfigurable embedded and modular systems," in *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*. IEEE, 2012, pp. 1–8. 2

[9] C. Lavin, B. Nelson, and B. Hutchings, "Impact of hard macro size on fpga clock rate and place/route time," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–6. 2

[10] A. Love, W. Zha, and P. Athanas, "In pursuit of instant gratification for fpga design," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–8. 2

[11] A. Das, S. Venkataraman, and A. Kumar, "Improving autonomous soft-error tolerance of fpga through lut configuration bit manipulation," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–8. 2

[12] M. Wirthlin, J. Jensen, A. Wilson, W. Howes, S.-J. Wen, and R. Wong, "Placement of repair circuits for in-field fpga repair," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2013, pp. 115–124. 2

[13] E. Hung, F. Eslami, and S. Wilton, "Escaping the Academic Sandbox: Realizing VPR Circuits on Xilinx Devices," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, April 2013, pp. 45–52. 2

[14] B. White and B. Nelson, "Tincr A Custom CAD Tool Framework for Vivado," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*. IEEE, Dec. 2014. 3, 12

[15] T. Haroldsen, B. Nelson, and B. Hutchings, "Rapidsmith 2: A framework for bel-level cad exploration on xilinx fpgas," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 66–69. 3, 17

[16] Xilinx. (2016) Vivado design suite 7 series fpga and zynq-7000 all programmable soc libraries guide. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug953-vivado-7series-libraries.pdf 10

[17] ——. (2016) Ultrascale architecture libraries guide. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug974-vivado-ultrascale-libraries.pdf 10

[18] C. Laven. (2010) Rapidsmith technical report and documentation. [Online]. Available: http://rapidsmith.sourceforge.net/docs.html 13, 15

[19] T. Haroldsen, B. Nelson, and B. Hutchings, "Packing a modern xilinx fpga using rapidsmith," in *ReConFigurable Computing and FPGAs (ReConFig), 2016 International Conference on*. IEEE, 2016, pp. 1–6. 17

[20] M. Wirthlin. (2017) Byu edif tools. [Online]. Available: http://reliability.ee.byu.edu/edif/ 71

[21] C. Gaisler, "Leon3," http://www.gaisler.com/index.php/products/processors/leon3. 80

[22] S. Hauck and A. Dehon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, 2008. 175

APPENDIX A. RAPIDSMITH2 DOCUMENTATION

This appendix includes portions of the current RapidSmith2 tech report found at: `https://github.com/byuccl/RapidSmith2/tree/master/doc`. Relevant portions of the RapidSmith2 documentation are included for two reasons:

1. One of the main contributions of this thesis is the integration of RapidSmith2 with VDI. Specific details about RapidSmith2 that are not discussed in the thesis body are introduced here in great detail for the reader.
2. This appendix serves as a snapshot for RapidSmith2 at the time of writing. Future updates to RapidSmith2 can be appropriately credited to the authors of the changes.

It is important to note that not all sections of the tech report are included in this appendix. Those that are irrelevant, or contain concepts already discussed in the thesis are omitted. All package paths in this appendix are relative to the *src/main/java* folder of the RapidSmith2 repository.

A.1 Devices

A.1.1 Device Data Structures

In the original RapidSmith, the device architecture stopped at the site level. A site was considered a black box who could be configured using string attributes, but the actual internal components were unknown. RapidSmith2 extends the device architecture to include all components **within** a site as well. Figure A.1 shows the new data structure hierarchy, which can be found in the package *edu.byu.ece.rapidSmith.device*. The classes and interfaces within *edu.byu.ece.rapidSmith.device* are named to reflect the terminology used by Xilinx. Many classes that exist in Vivado's Tcl interface have a direct map to a class in RapidSmith2 (such as a `Tile`). Because of this, most RapidSmith2 data structures represent a straightforward part of a Xilinx FPGA. The

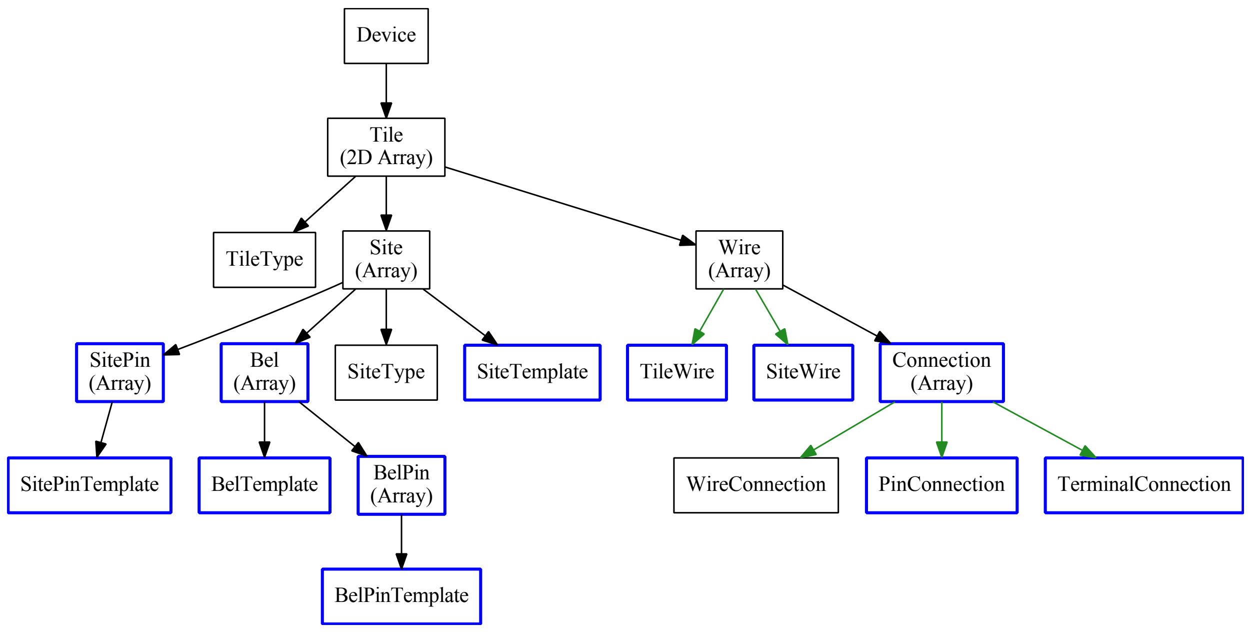


Figure A.1: RapidSmith2 Device data structure tree. Green arrows represent inheritance, and black arrows represent association. Classes and Interfaces bolded in blue are new to RapidSmith 2.

`DeviceBrowser` and `DeviceAnalyzer` example programs illustrate how to load and browse a device with `Tile` and `Site` data structures, and Listing A.1 shows basic device usage in RapidSmith2. The remainder of this section details important aspects of RapidSmith2 devices.

Listing A.1: Basic device function calls

```
Device device = getDevice();

// Get device components by name
Tile tile = device.getTile("CLBLL_R_X27Y130");
Site site = tile.getSite("SLICE_X44Y130");
Bel bel = site.getBel("D6LUT");

// Get all device components
device.getTiles();
tile.getSites();
site.getBels();
```

Templates

As Figure A.1 shows, there are several template classes in RapidSmith2. Template classes are used to specify the configuration of a device structure only once, where the configuration can be reused across identical objects. The usefulness of templates is best shown with an example. In an Artix-7 *xc7a100tcsg324* part, there are 11,100 sites of type SLICEL. Each of these SLICELs have 215 internal components (BELs, pins, and PIPs). To save memory, RapidSmith2 lazily creates site objects based on the template only when a handle to a SLICEL site is requested. The alternative would be to create each of the objects when a device is loaded, but this would require more memory. Template classes should generally not be used by the regular user. When creating algorithms using RapidSmith's API, use the non-template version of classes instead.

WireEnumerator

Wires with the same name and function can occur several times throughout a Xilinx FPGA. For example, the wire `CLBLL_L_C2` exists in every tile of type `CLBLL_L` in a Series7 device. To make RapidSmith2 device files small, each uniquely named wire is assigned an integer enumeration and stored in a `WireEnumerator` class. The `WireEnumerator` has methods to convert an integer to the corresponding string wire name and vice versa.

In previous versions of RapidSmith, the `WireEnumerator` was used extensively while building CAD tools. RapidSmith2 has changed this, largely abstracting the `WireEnumerator` away in favor of more convenient methods that return `Wire` objects. For example, the name or enumeration of a wire can now be obtained with the function calls in the `Wire` interface `getWireName()` and `getWireEnum()` respectively. A handle to the `WireEnumerator` still exists in the `Device` class for those who want to use it, but this is not recommended.

TileWire and SiteWire

Wires in RapidSmith2 are uniquely identified not only by their name (or enumeration), but also by the tile or site in which they exist. RapidSmith2 introduces the `TileWire` and `SiteWire` classes to encapsulate this information for the user. Many functions in RapidSmith now return a

`TileWire` or `SiteWire` object (wrapped in a generic `Wire`) instead of an integer wire enumeration. `Wires` are connected through `Connection` objects as described in section A.4.

Package Pins

Vivado maps all *bonded* IOB sites to corresponding package pins. Top-level ports of a design can be mapped to these package pins to communicate with external components. A RapidSmith2 `Device` object represents Vivado package pins with `PackagePin` objects. Each `PackagePin` contains (1) the name of the package pin (i.e. M17), (2) the PAD BEL of the package pin, and (3) a boolean flag to mark clock package pins. Clock package pins are those that can access the global clock routing structure of the FPGA for low skew signals. Listing A.2 shows some available package pin method calls.

Listing A.2: RapidSmith2 package pin functions

```java
// Get a list of all package pins
device.getPackagePins();

// Get a list of clock package pins
device.getClockPads();

// Get an individual package pin based on the BEL
Bel bel = device.getSite("IOB_X0Y10").getBel("PAD");
PackagePin pin = device.getPackagePin(bel);

// Package pin functions
pin.getName();
pin.getSite();
pin.getBel();
pin.isClockPad();
```

A.1.2 Loading a Device

Listing A.3 demonstrates how to load a supported device into RapidSmith2. The first function call will only load the device into memory if it has not yet been loaded. If it has been loaded, then the cached `Device` data structure will be returned. The second function call will reload the device from disk, creating a separate `Device` data structure. This is useful when implementing multi-threaded code that targets the same part. **NOTE:** When a Vivado design is loaded into RapidSmith2 via a RSCP, the corresponding device is also loaded.

Listing A.3: Loading a Device

```java
Device device = RSEnvironment.defaultEnv().getDevice("xc7a100tcsg324");
//or
Device reload = RSEnvironment.defaultEnv().getDevice("xc7a100tcsg324", true);
```

A.1.3 Supported Device Files

RapidSmith2 includes two device files on installation: (1) an Artix7 `xc7a100tcsg324`, and (2) a Kintex UltraScale `xcku025-ffva1156`. These device files have been well tested and are a good starting point for new users looking to implement Vivado CAD algorithms. RapidSmith2, however, has general support for the following families:

- Artix7
- Virtex7
- Kintex7
- Zynq
- Kintex UltraScale
- Virtex UltraScale

Section A.6 describes how to create new device files for these families and add them to RapidSmith2.

A.2 Designs

A.2.1 RS2 Netlist Data Structures

RapidSmith2 netlists are modeled closely after Xilinx netlists. In fact, much of the terminology between the two are identical or very similar. For those that are familiar with Vivado designs, this should make the transition to RapidSmith2 straightforward. The data structures that constitute a RapidSmith2 netlist can be found in the package *byu.edu.ece.rapidSmith.design.subsite*. The package hierarchy is shown in Figure A.2.

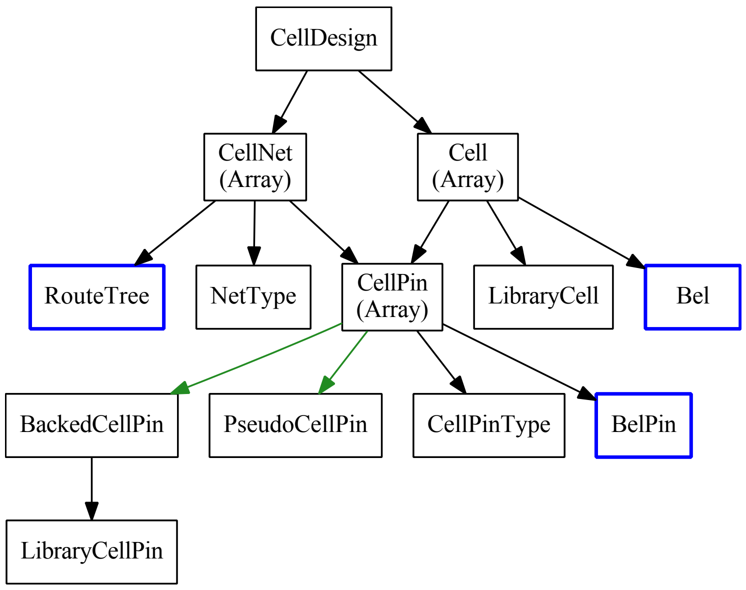


Figure A.2: RapidSmith2 design data structure tree. Black arrows represent composition, green arrows represent inheritance, and blue boxes are physical implementation components of the netlist.

As can be seen, a `CellDesign` is the top-level design object in RapidSmith2. It consists of a collection of `Cell` objects, interconnected by `CellNets`. RapidSmith2 `Cells` are equivalent to Xilinx cells and RapidSmith2 `CellNets` are equivalent to Xilinx nets. Each `Cell` has a template `LibraryCell`, which represents a Xilinx library primitive (i.e. a LUT). They also have a collection of connected `CellPin` objects. Placing and routing these logical design elements is described in section A.3 and A.4 respectively. The best way to learn how to utilize the classes shown in Figure A.2 is to generate and read through the JavaDocs, but important aspects of each class is included in the following subsections.

CellDesign

As previously mentioned, the `CellDesign` class is the top-level netlist object in RapidSmith2. An instance of a `CellDesign` contains the following:

- A list of cells
- A list of nets
- Global GND and VCC nets
- Cell placement information (i.e. where each cell is placed)
- The used site PIPs of each site
- A list of XDC constraints imported from Vivado. See section A.5 for more information about XDC constraints and how they are represented in RapidSmith2.

The `CellDesign` class has a variety of methods to retrieve and manipulate the cells and nets of a design, place cells onto physical BELs, configure sub-site routing, and perform several other tasks.

Cell

`Cell` objects are the building blocks of RapidSmith2 netlists. This section details some important aspects of `Cells`.

- A `Cell` always contains a reference to a backing `LibraryCell` object. A `LibraryCell` is equivalent to a Xilinx primitive cell (described in section 2.2), and serves as a template for instantiated `Cell` objects. The template is used to save memory when creating several `Cells` of the same type. Whenever a new `Cell` object is created, a corresponding `LibraryCell` must be specified in the constructor. Listing A.4 demonstrates how to create new cells in RapidSmith2 and filter cells based on their type.

Listing A.4: How to create new cells in RapidSmith

```
// You need to have a cell library to create a cell
```

```
CellLibrary libCells = new CellLibrary(RSEnvironment.defaultEnv()
            .getPartFolderPath("xc7a100t-csg324")
            .resolve("cell_library.xml"));

// How to create a new cell. The first argument is the name of the cell, the
// second argument is the library cell
Cell cell = new Cell("myCell", libCells.get("LUT6"));

// Get all cells in a design of a certain type
CellDesign design = getCellDesign();
Stream<Cell> cells = design.getCellsOfType(libCells.get("FDRE"));
```

- The method call `CellDesign::getCellsOfType(String,CellLibrary)` can be used to get all cells in the current design with a specific type.
- The methods `Cell::getPins()`, `Cell::getInputPins()`, and `Cell::getOutputPins()` can be used to get a handle to the pins of a `Cell`. If more pins are needed on a `Cell`, `PseudoCellPin` objects can be attached (see section A.2.1).
- `Cells` can be placed onto `Bel` objects of the current `Device`. See section A.3 for more information about cell placement.
- Top-level `Ports` in Vivado (design input/output) are represented as Port `Cells` in RapidSmith2. Specifically, there are three types of port cells: IPORT, OPORT, and IOPORT. The method `CellDesign::getPorts()` can be used to iterate through the ports in a design, and `Cell::isPort()` can be used to determine if a given `Cell` is actually a port.
- RapidSmith2 supports both Xilinx macro and leaf cells. More information about these is given in section 2.2 and subsection 3.4.4 respectively.

CellPin

CellPins in RapidSmith2 are attached to Cell objects and are equivalent to the cell pins found in Vivado. Each CellPin has an associated CellPinType and PinDirection. Table A.1 displays the possible values for both of these properties.

Table A.1: Cell Pin Types and Directions

Property	Values
CellPinType	CLEAR CLOCK ENABLE PRESET RESET REUSED SET SETRESET WRITE_ENABLE DATA PSEUDO
PinDirection	IN OUT INOUT

The CellPinType can be used to find all RESET pins in a design, determine if a net is a clock net (it connects to pins of type CLOCK), and help with other useful functions. Cell pins of type PSEUDO are a special case, and described in section A.2.1. The PinDirection field is typically used to filter a list of pins on a cell by their direction. It is especially useful for finding INOUT pins.

CellNet

CellNets are used to wire components of a logical netlist together. Specifically, a CellNet connects an output CellPin to several input CellPins with the purpose of transferring a signal from one Cell to another. Listing A.5 shows the basic usage of CellNet objects in RapidSmith2, and the remainder of this section details other important aspects about CellNets.

Listing A.5: Basic CellNet functions

```java
CellDesign design = getCellDesign();

// creating a new cell
CellNet net = new CellNet("myNet", NetType.Wire);
design.addNet(net);

Cell cell1 = design.getCell("cell1");
Cell cell2 = design.getCell"cell2");

// connecting nets to cell pins
net.connectToPin(cell1.getSourcePin());
net.connectToPin(cell2.getpin("a"));
```

- `CellNets` are routed using `RouteTree` data structures. RapidSmith2 routing is described in more detail in section A.4.

- All `CellNets` have a `NetType` enumeration. Possible values for `NetType` include VCC, GND, and WIRE. VCC is reserved for power nets, GND is reserved for ground nets, and WIRE represents all other nets in the design.

- The suggested approach to working with static nets in RapidSmith2 is to have only one VCC and GND net in a `CellDesign`. In general, this representation is much easier to work with and the special nets can be obtained with the functions `CellDesign::getVccNet()` and `CellDesign::getGndNet()`. When a design is imported from Vivado through a RSCP, all VCC and GND nets are collapsed automatically. Having multiple VCC and GND nets, however, is still supported if desired.

- Most nets have a single driver, but some can be sourced in multiple locations. Figure A.3 shows an example for a GND net. RapidSmith2 handles this oddity by allowing `CellNets` to have more than one `RouteTree` object associated with it. In the case of Figure A.3 the net would have two `RouteTrees`, one for each source TIEOFF.

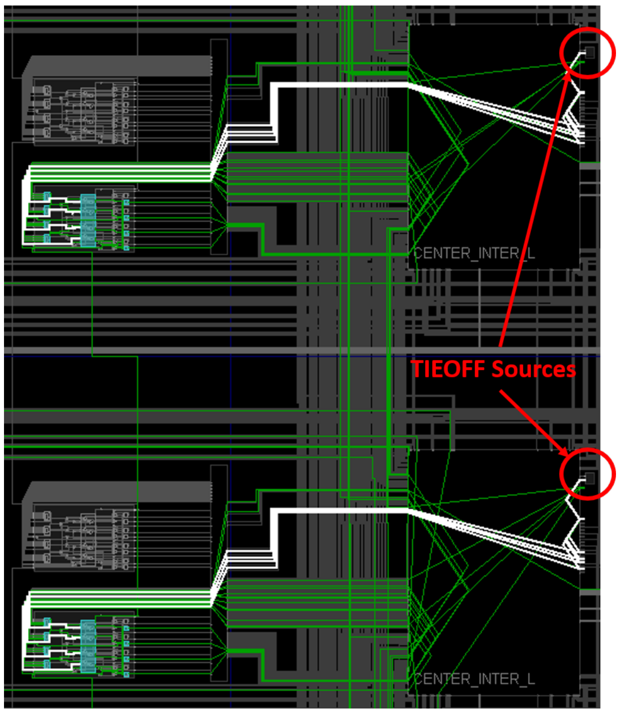


Figure A.3: Example of a Vivado net with multiple sources. The highlighted wires in white are all part of the same net.

- Figure A.4 shows a bidirectional net in Vivado. As can be seen, the highlighted net can be driven by both the OBUF output, and from an external source via the PAD BEL. RapidSmith2 supports bidirectional nets, and a list of possible drivers can be obtained with the function call `CellNet::getAllSources()`.

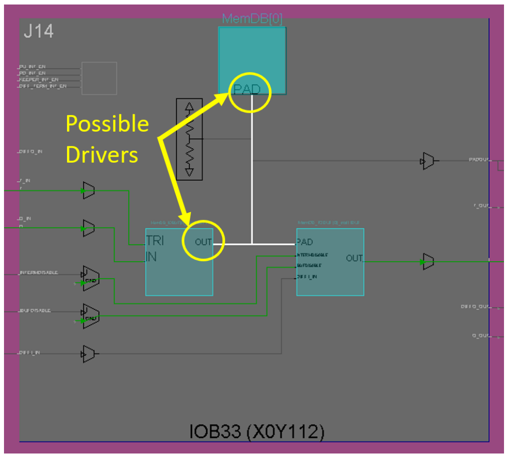


Figure A.4: Bidirectional Net

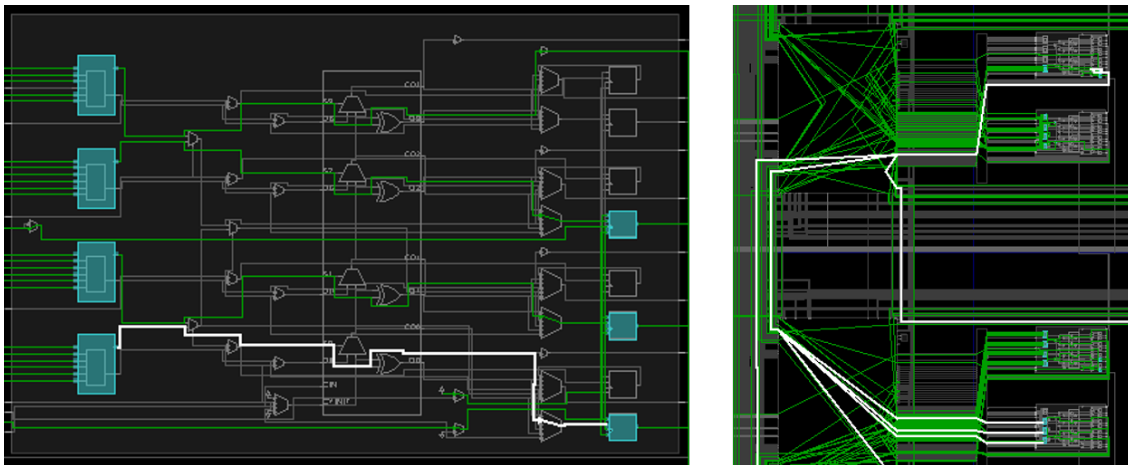


Figure A.5: Example INTRASITE Net (left) and INTERSITE Net (right)

- After a design has been placed, `CellNets` fall into one of two categories: **intrasite** vs. **intersite**. Figure A.5 shows an example of both types of nets. As can be seen, intrasite nets do not cross site boundaries while intersite nets stretch across multiple sites. To determine if a `CellNet` is an intrasite net, the method `CellNet::isIntrasite()` can be used.

Macro Cells

As described in subsection 3.4.4 flattened Vivado netlists can include macro primitives. RapidSmith2 now supports importing macro cells from Vivado and adding them to a `CellDesign`. Listing A.6 gives a brief introduction to using macros in RapidSmith2.

Listing A.6: How to use macros in RapidSmith

```java
// Get a handle to a design and cell library
CellDesign design = getCellDesign();
CellLibrary libCells = getCellLibrary();

// Create a new macro cell and add it to a design
Cell macro = new Cell("myMacro", libCells.get("IOBUF"));
design.addCell(macro);

// Connect the macro cell to a net
```

```
CellPin pin = macro.getPin("I0");
design.getNet("TmpNet").connectToPin(pin);

// Iterate through a list of all cells (macro and leaf cells) of a design
for (Cell cell : design.getCells) {
   if ( cell.isMacro() ) {
      List<Cell> internalCells = cell.getInternalCells();
      List<CellNet> internalNets = cell.getInternalNets();
      List<CellPin> externalPins = cell.getPins();
      // do something with the macro info
   }
   else {
      // do something with a regular leaf cell
   }
}
```

As the code example above shows, macro cells are generally used exactly like regular leaf cells. However, there are a few distinctions between macro cells and leaf cells.

- When a macro cell is added to a design, all internal cells and nets are automatically added to the design as well. Users do not have to worry about adding these themselves. Similarly, when a macro cell is removed from a design, the internal cells and nets are also removed.

- When a macro cell pin is connected to a `CellNet`, RapidSmith2 automatically connects the net to the corresponding internal pins. When a macro cell pin is disconnected from a `CellNet`, the internal cell pins are disconnected. Nets in RapidSmith2 only connect to leaf cell pins (i.e. it is essentially a flattened netlist with macros cell "wrappers").

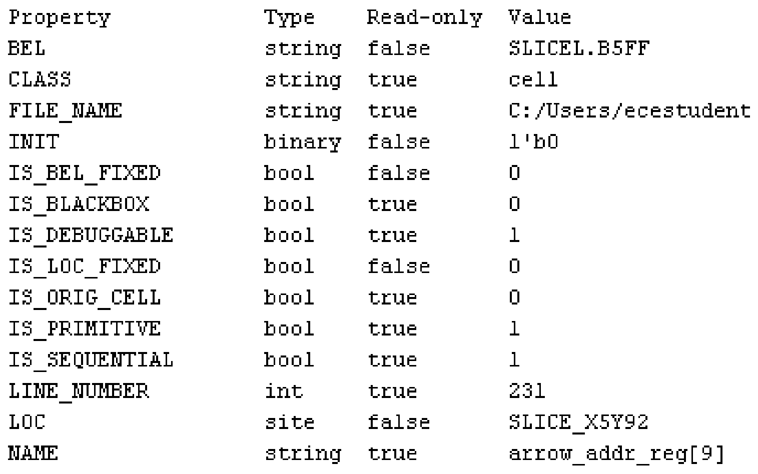
- Internal cells and nets within a macro cannot be individually added or removed from a design. If this is attempted, an exception will be thrown. Instead, the entire macro cell must be added or removed.

- Macros cannot be placed. Rather, the internal cells of a macro should be placed instead.

- When a `CellDesign` is exported from RapidSmith, macro cells are not exported. The design is first flattened, and only the internal cells and nets are exported. This means the macro will not be rebuilt in Vivado.

PropertyList

Most objects in Vivado's Tcl interface have attached properties. These properties can be used to describe attributes of the object (such as name, type, etc.), but they can also be used for configuring the object. Figure A.6 shows a list of properties for a FDCE flip flop cell in Vivado.



```
Property           Type    Read-only  Value
BEL                string  false      SLICEL.B5FF
CLASS              string  true       cell
FILE_NAME          string  true       C:/Users/ecestudent
INIT               binary  false      1'b0
IS_BEL_FIXED       bool    false      0
IS_BLACKBOX        bool    true       0
IS_DEBUGGABLE      bool    true       1
IS_LOC_FIXED       bool    false      0
IS_ORIG_CELL       bool    true       0
IS_PRIMITIVE       bool    true       1
IS_SEQUENTIAL      bool    true       1
LINE_NUMBER        int     true       231
LOC                site    false      SLICE_X5Y92
NAME               string  true       arrow_addr_reg[9]
```

Figure A.6: Properties of a Vivado FDCE Cell

The Tcl command `[report_property $object]` can be used to list all properties for a given Vivado object (cell, BEL, etc.). Cells are the most interesting objects in terms of properties because the function of a `Cell` is determined by how it is configured. For example, the memory width of a BRAM cell in Vivado is configured by setting the `READ_WIDTH` and `WRITE_WIDTH` properties of the cell. Possible values include 1, 2, 4, 9, 18, 36 and 72. The operation of the BRAM is different depending on how this property is set. Another example is a D flip flop cell (FDRE) and its `IS_C_INVERTED` property. This property indicates if the flip flop will be rising-edge or falling-edge triggered. The properties of cells, nets, and the top-level design are included in the output EDIF netlist of a RSCP for non-default values only.

When RapidSmith2 parses the EDIF file of a RSCP, the properties within are stored in a data structure called a `PropertyList`. Each `CellDesign`, `Cell`, and `CellNet` in RapidSmith has an associated `PropertyList` object. The `PropertyList` for each cell in the design also has a list of default configuration properties. Configuration properties for cells are always included in the `PropertyList` even if they are not explicitly set by the user because the functionality of the cell is dependent on how it is configured. Listing A.7 shows some basic property usage.

Listing A.7: Using PropertyLists in RapidSmith

```
// Create a new FF cell with default properties
CellLibrary libCells = getCellLibrary();
LibraryCell libCell = libCells.get("FDRE");
Cell cell = new Cell("myCell", libCell);

// Get a handle to the cells properties
PropertyList properties = cell.getProperties();

// Print the configurable properties of the cell
for(String propName : libCell.getConfigurableProperties()) {
   Property prop = properties.get(propName);
   System.out.println(propName + ":");
   System.out.println("\tDefault -> " + prop.getStringValue());
   System.out.println("\tPossible -> " + libCell.getPossibleValues(propName));
}

// Iterating over a PropertyList
for (Property prop : properties) {
   System.out.println(prop.getKey() " -> " + prop.getStringValue());
}

// Change the FF to be falling edge triggered...this will override the default
property properties.update("IS_C_INVERTED", PropertyType.EDIF, "1’b1");
```

Some additional notes about properties are given below.

- In Vivado, the configurable properties on a cell can be determined by using the Tcl command `[report_property [get_lib_cells $cell]]`. All properties that start with "CONFIG" are configurable properties that can be modified.
- Because EDIF properties only support String, Integer, and Boolean types, any properties imported from the EDIF file will be one of these types. It seems, however, that Vivado always exports its properties as strings [1].
- Only properties of type `PropertyType.EDIF` will be exported from RapidSmith2. When using properties, make sure to mark the type of the property as EDIF. All other properties will be ignored during design export.

Xdc Constraints

In Vivado, XDC constraint files are used to set the target clock frequency of a design, constrain a top-level port to a specific package pin on the device, or specify other physical implementation details. A RapidSmith2 `CellDesign` represents these constraints with `XdcConstraint` objects. Currently, `XdcConstraints` only contain two fields: (1) a command name (such as *set_property*) and (2) the command arguments (combined into a single string). It is the responsibility of the user to parse these XDC constraints if they need to use them in their CAD tools. The function `CellDesign::getVivadoConstraints()` returns a list of constraints currently attached to a design and `CellDesign::addVivadoConstraint()` can be used to add new constraints to a design.

Pseudo Cell Pins

To address the routing issue described in subsection 4.3.3, RapidSmith2 allows users to create and attach `PseudoCellPins` to an existing cell. For example, a `PseudoCellPin` can be attached to the cell shown in Figure A.7. In this case VCC is routed to the `A6` BEL pin of the

[1]RapidSmith makes no attempt to parse the Vivado properties into their corresponding data structures. All Vivado properties are represented using Strings, and it is currently up to the user to parse the properties if they need to

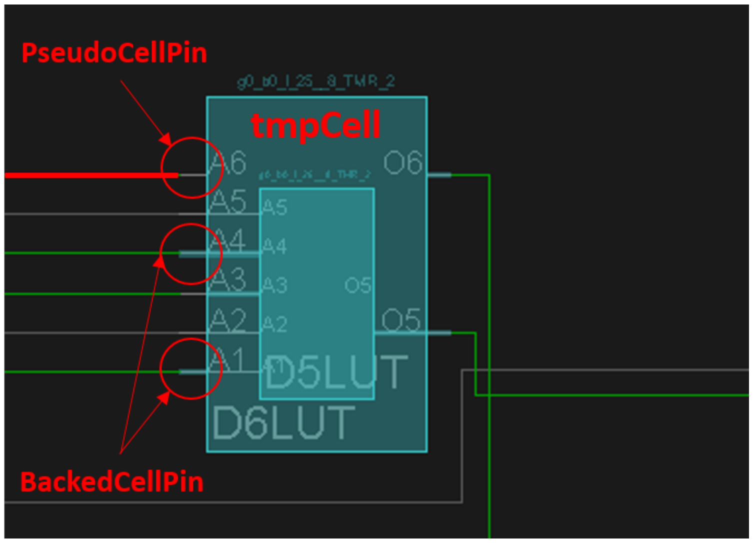


Figure A.7: Pseudo Cell Pin Example

D6LUT, but this is not represented in the design netlist. To explicitly represent this connection in the netlist, a new `PseudoCellPin` can be created, attached to the VCC net of the design, attached to the cell in the figure ("tmpCell"), and then mapped the `A6` BEL pin. Listing A.8 demonstrates how to attach a pseudo pin to a cell in RapidSmith2.

Listing A.8: Required function calls to attach a PseudoPin to a Cell

```
BelPin bp = getBelPin();
Cell cell = design.getCell("foo");
CellPin pseudo = cell.attachPseudoPin("VccTmpPin");
pseudo.mapToBelPin(bp)
```

A.2.2 The Cell Library

RapidSmith2 parses the cell library XML file described in section 3.4 into a `CellLibrary` data structure. A `CellLibrary` is required to perform any type of netlist modification, or to import a design from Vivado. Currently, each `CellLibrary` corresponds to a specific Xilinx part. This means that for each device file in RapidSmith2, a new `CellLibrary` needs to be generated [2]. Listing A.9 shows two ways to load a `CellLibrary` in RapidSmith2.

Listing A.9: Ways to load a CellLibrary

```java
// First way, load a Tincr Checkpoint
TincrCheckpoint tcp = VivadoInterface.loadTcp("checkpoint.tcp");
CellLibrary libCells1 = tcp.getLibCells();

// Second way, directly load the cell library from disk
CellLibrary libCells2 = new CellLibrary(RSEnvironment.defaultEnv()
                    .getPartFolderPath("xc7a100t-csg324-3")
                    .resolve("cellLibrary.xml");
```

Adding Custom Macros to a Cell Library

Customized, user-defined macros can be added to a RapidSmith2 `CellLibrary` if desired. This can be accomplished in two easy steps.

1. Create a XML specification of the macro that follows the format laid out in Listing 3.10.
2. Import the macro into the `CellLibrary` using the API call shown in Listing A.10.

Listing A.10: Function calls to add a new macro to a CellLibrary

```java
// Get a handle to a CellLibrary
CellLibrary libCells = getCellLibrary();

// Add the macros in a XML file.
libCells.loadMacroXML(Paths.get("myMacro.xml");
```

Once this is complete, you can use the custom macro in a `CellDesign` like a normal cell. Potential uses for customized macros include hard-macro design flows, and supporting a single-level of design hierarchy.

[2]This may change to be family-specific in the future, but usually, parts in the same family can use the same `CellLibrary`.

A.3 Placement

In the original RapidSmith, placement occurred at the site level. A collection of cells were grouped together into an *instance*, and the instance was assigned to a compatible site. The actual placement locations for the cells within the site were unknown. Because RapidSmith2 breaks up a site into its individual components, cells can now be placed directly onto physical BELs within a site. This gives Xilinx FPGA CAD developers finer-grained control over the placement of a design, and allows sub-site algorithms (such as packers) to be explored. Listing A.11 demonstrates the basic steps to placing cells in RapidSmith2.

Listing A.11: Steps for placing a Cell in RapidSmith2

```
// Load the device and design
TincrCheckpoint tcp = VivadoInterface.loadTcp("myCheckpoint.tcp");
Device device = tcp.getDevice();
CellDesign design = tcp.getDesign();

// Get a handle to a Cell and Bel. The cell is of type LUT2
Cell cell = design.getCell("MyCell");
Bel bel = device.getSite("SLICE_X40Y137").getBel("D6LUT");

// Place the cell onto the bel
design.placeCell(cell,bel);

// Two ways to map bel pins
CellPin pin1 = cell.getPin("I0");
CellPin pin2 = cell.getPin("I1");
CellPin pin3 = cell.getPin("O")

// First way
pin1.mapToBelPin(bel.getPin("A1"));
pin2.mapToBelPin(bel.getPin("A2"));

```

```
22  // Second way
23  List<BelPin> possible = pin3.getPossibleBelPins(bel);
24  pin3.mapToBelPin( possible.get(0) );
```

As the code listing shows, there are two steps to placing a RapidSmith2 `Cell`. The first is to get a handle to a `Bel` object, and use the method `CellDesign::placeCell(cell, bel)` (line 11). Once a `Bel` has been used, no other `Cell` can be mapped to it. No error checking is performed to ensure that the cell is actually compatible with the BEL. The second step is to map each pin of the `Cell` object to a corresponding BEL pin. This can be done by either (a) specifying the BEL pin name (lines 19-20), or (b) using the function `CellPin::getPossibleBelPins(bel)` (line 23). Most cell pins only map to one BEL pin, but there are two noticeable exceptions to this rule.

1. LUT input pins are permutable. This means that an input cell pin attached to a LUT cell can be mapped to any input pin of a LUT BEL. Figure A.8 shows an example of this functionality in Vivado. In this case, `CellPin::getPossibleBelPins(bel)` will return all input BEL pins of the LUT and the user can decide which ones to use.

2. Logical-to-physical pin mappings can change **based on how a cell is configured**. For example, on a RAMB36E1 cell some data input pins map to different physical BEL pins when the width of the BRAM is set to 72. This is an important concept when performing netlist modifications in RapidSmith2. The function call `CellPin::getPossibleBelPins(Bel)`

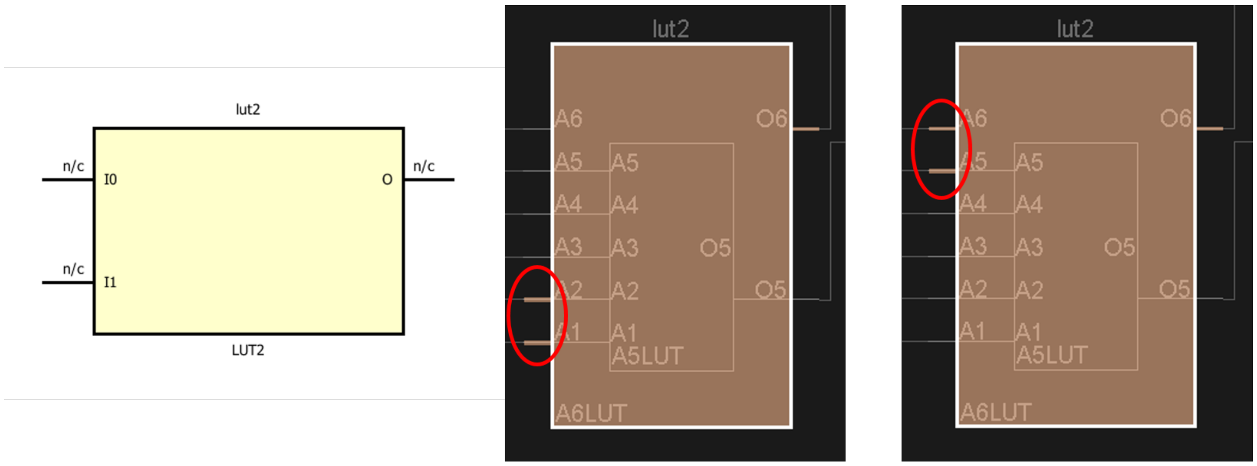


Figure A.8: LUT Pin Permutation Example

only returns the pin mappings for the **default** cell configuration. If new logic is being added to a design it is up to the user to determine the proper pin mappings. Users can determine the pin mappings of a configured cell by using the TCL commands shown in Listing A.12. The correct pin mappings are always used when a RSCP is imported into RapidSmith2.

Listing A.12: TCL script to print all logical-to-physical pin mappings of a cell

```
proc print_pin_mappings{cell} {
   foreach cell_pin [get_pins -of $cell] {
      puts "$cell_pin -> [get_bel_pins -of $cell_pin]"
   }
}
```

Some additional notes about placement are given in the list below:

- VCC and GND cells are not placed when implementing a design in Vivado. This distinction is applied to RapidSmith2 as well. Rather than placing VCC or GND explicitly, `RouteTrees` that are sourced by switchbox TIEOFFs are used to express their placement implicitly (VCC/GND is "placed" on the TIEOFF).

- A list of valid placement options for a cell can be obtained with the function call `Cell::getPossibleAnchors()`. The sample program **CreateDesignExample** demonstrates how to use this function.

- There are several placement rules for a given Xilinx FPGA. One such rule is that CARRY4 cells which are connected through a carry chain need to placed vertically to one another. Another example is that a RAMB36E1 cell cannot be placed in the same tile as a RAMB18E1 cell. If either of these rules are violated, an error will be thrown in Vivado when attempting to import a design. It is the responsibility of the user to determine all relevant placement rules because error checking is not performed on design export.

- Macro cells in RapidSmith2 cannot be placed. The internal cells of a macro should be placed instead.

A.4 Routing

During placement, all cells of a design are mapped to BELs, and all cell pins are mapped to BEL pins. The next (and final) step of the FPGA implementation flow is to physically wire together the used BEL pins. This is known as routing. Routing involves taking each logical net of a design, determining the BEL pins they are connected to (based on the cell pins), and finding a list of physical wires that electrically connect the pins together. This section details how routing algorithms can be implemented in RapidSmith2.

A.4.1 Wires and Wire Connections

Routing in RapidSmith2 is done using `Wire` objects, which are described in subsection 2.1.3. `Wires` are uniquely identified by their corresponding tile and wire name (i.e. "tileName/wireName"), and are connected through `Connection` objects. There are two types of wire connections:

1. **PIP Connections**: Connect two different wires through a Programmable Interconnect Point. Most PIP connections are found in switchbox tiles of a FPGA part (as shown in Figure 2.5). These types of connections are important to FPGA routing, because they dynamically configure the routing network for a given design.

2. **Non-PIP Connections**: Connect the same physical wire across two different tiles. In general, wires stretch across multiple tiles in a device, having a different name in each tile. This is demonstrated in Figure A.9. The example wire shown in the figure spans 5 tiles, but has a different name in each. To save space, only the source and sink wire segments are kept

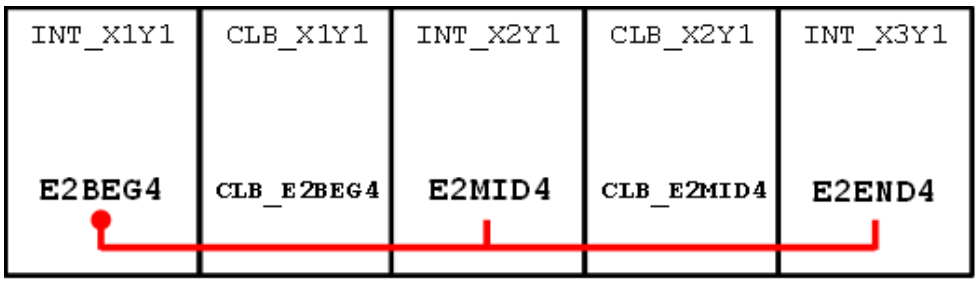


Figure A.9: Multi-Tile Xilinx Wire

in RapidSmith2 data structures (i.e. `INT_X1Y1/E2BEG4`, `INT_X2Y1/E2MID4`, and `INT_X3Y1/E2END4`). The source segment is connected to each sink segment through a non-PIP wire connection. It is also possible to have non-PIP connections within a tile, but this is rare.

A.4.2 Traversing Wire Objects

Traversing through wires in a device is straightforward. Given a handle to a `Wire` object named ”mywire” or a `Connection` object named ”conn”, the following function calls can be used:

- `mywire.getWireConnections()`: Returns a collection of all `Connection` objects whose source is “mywire”. This collection can be iterated over to find all places a specific wire goes (i.e. what wires it connects to).
- `conn.isPip()`: Returns true if the wire connection “conn” is a PIP connection. Returns false otherwise.
- `conn.getSinkWire()`: Returns the sink wire of a wire connection.

In general, these are the only three functions that are needed to search through the wires of a FPGA device. It is important to note however that the first wire in the route must be either (a) created using a `TileWire` constructor, or (b) retrieved from a function call of another object (such as `SitePin::getExternalWire()`). Listing A.13 demonstrates how to iterate over `Connection` objects. To gain a better understanding of how to use `Wires` and `Connections`, see the **HandRouter** example in the RapidSmith2 repository.

A.4.3 Other Types of Connections

Along with PIP and non-PIP wire connections, there are several other types of connections in RapidSmith2. The source of the connection is always a `Wire` object, but the sink object differs. A description of these connections is found below:

- **Site Pin Connections**: Connects a `Wire` to a `SitePin`. The function call `Connection::.getSitePin()` can be used to return a handle to the site pin.

- **Terminal Connections**: Connects a `Wire` to a `BelPin`. The function call `conn.getBelPin()` can be used to return a handle to the BEL pin.

- **Site Routethrough Connections**: Connects an input site `Wire` to an output site `Wire`. A `Site` in Vivado can be configured to pass the signal on an input pin directly to an output pin. These connections are represented as routethroughs in RapidSmith2 and can be determined with the function call `Connection::isRoutethrough()`. **NOTE**: Before using this type of connection when building a routing data structure, make sure the corresponding site is unused.

- **BEL Routethrough Connections**: Connects an input BEL `Wire` to an output BEL `Wire`. LUTs in Vivado can also be configured as routethroughs. A BEL routethrough connection can be used while routing the inside of a site if there is no cell placed on the corresponding BEL.

When traversing through the device data structure, a generic `Connection` object is usually used. This connection can refer to any of the connections described so far in this documentation. Listing A.13 demonstrates how to iterate through different `Connection` types in RapidSmith2.

Listing A.13: How to iterate over Connections in RapidSmith2

```
// Get a handle to a wire
Wire wire = sitePin.getExternalWire();

// Iterate over all WireConnections
for (Connection conn : wire.getWireConnections()) {
  Wire sinkWire = conn.getSinkWire();

  if (conn.isPip()) {
    // Do something with a PIP
  }
  else if (conn.isRoutethrough()) {
    // Do something with a routethrough
  }
```

```
    else {
      // Do something with a regular wire connection
    }
  }

  // Get the site pin connected to a wire
  SitePin pin = wire.getConnectedPin();
  if (pin != null)
    // Do something with the SitePin
  }

  // Get the bel pin connected to a wire
  BelPin pin = wire.getTerminal();
  if (pin != null)
    // Do something with the BelPin
  }

  // Iterate over all connections at the same time
  Iterator<Connection> connIt = wire.getAllConnections().iterator();

  while (connIt.hasNext()) {
    Connection conn = connIt.next();

    if (conn.isPinConnection()) {
      //do something with the site pin
    }
    else if (conn.isTerminal()) {
      //do something with the bel pin
    }
    else if (conn.isPip()) {
      //do something with the pip wire connection
    }
```

```
        else {
            //do something with regular wire connection.
        }
    }
```

A.4.4 RouteTrees

`Wires` and `WireConnections` are the fundamental objects used to specify and explore routing in RapidSmith2, but they need to be organized in a higher-level data structure to give meaning to the route of a `CellNet`. In the original RapidSmith, creating this data structure was up to the user. RapidSmith2 introduces the `RouteTree`, which can be seen in Figure A.10.

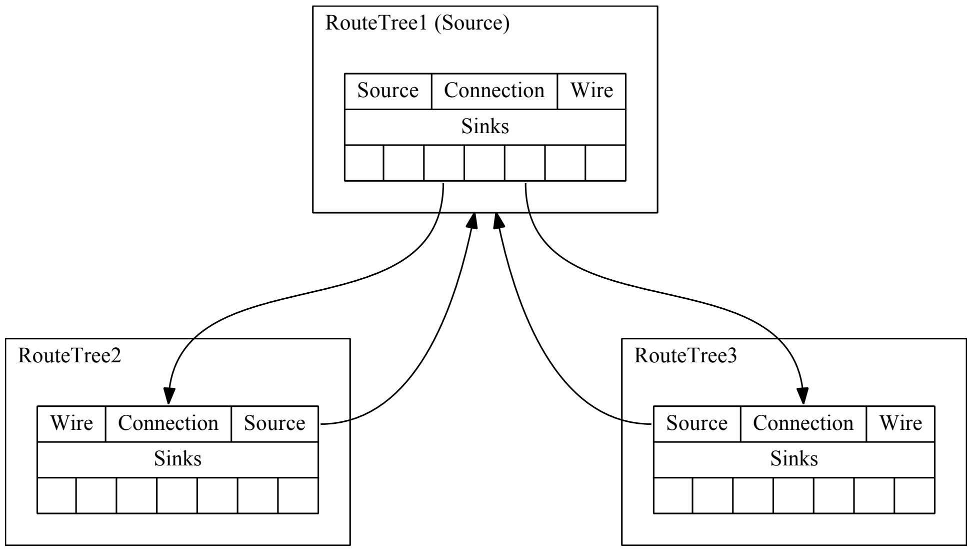


Figure A.10: Visual Representation of a RapidSmith2 RouteTree

As the figure shows, a `RouteTree` is a simple tree data structure. Each node in the tree represents a physical wire in the device, and is connected to other nodes (wires). Edges in the tree represent wire connections (i.e. how one wire connects to another). A `RouteTree` can also be conceptually thought of as a graph, with a single "starting" node and several "sink" nodes. A `RouteTree` node contains the following members:

- **Wire**: The physical `Wire` object that the `RouteTree` node represents. This can be either a `TileWire` or `SiteWire`.
- **Source**: A link to the parent `RouteTree` node
- **Connection**: The `Connection` taken from the **parent** `RouteTree` node to reach the **current** `RouteTree` node. In other words, it is the `Connection` object that was taken from the parent wire to reach the current wire.

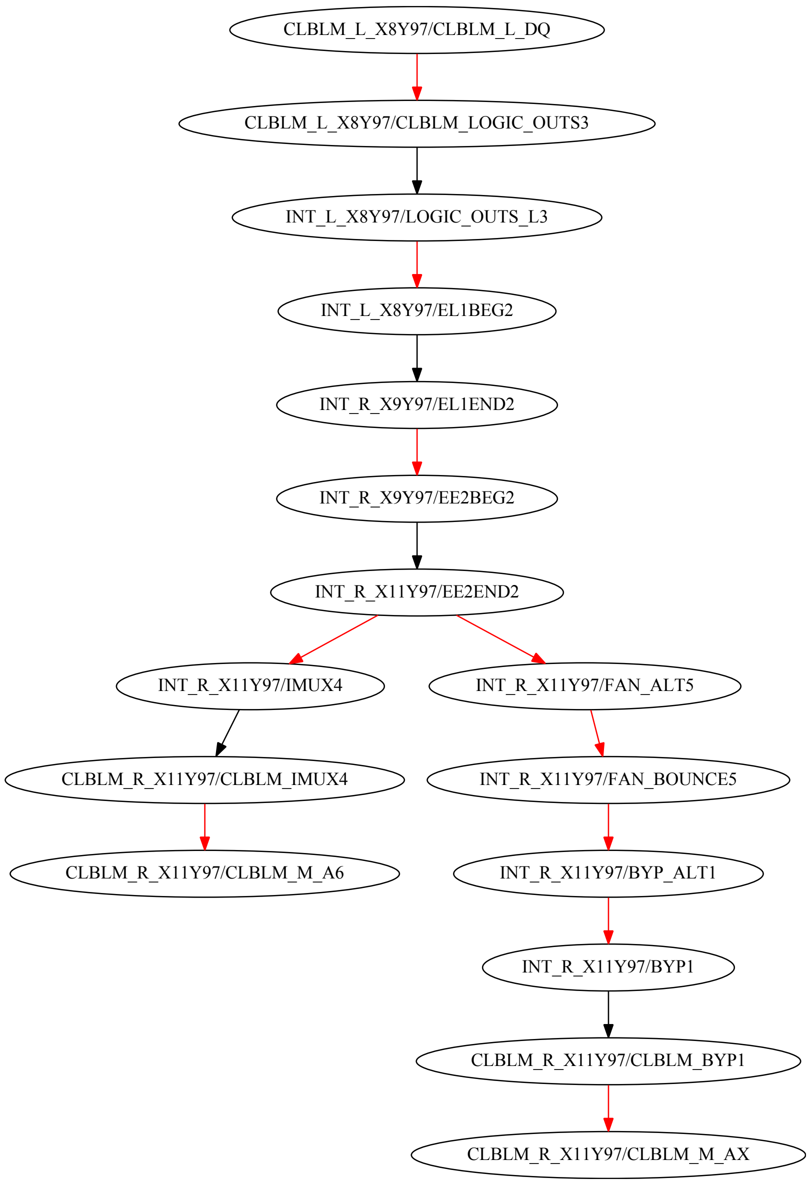


Figure A.11: Sample RapidSmith2 RouteTree (red edges represent PIP connections)

- **Sinks**: A list of child nodes. There is no limit to how many children a `RouteTree` can have.
- **Cost** (not shown): An optional cost field for routers

A complete `RouteTree` specifies how the source of a `CellNet` is physically connected to all of its sinks. Figure A.11 shows an example of a complete `RouteTree` in RapidSmith2. As can be seen, the `CellNet` that is being routed has once source pin, and two sink pins. The source pin is connected to wire `CLBLM_L_X8Y97/CLBLM_L_DQ`, and the sink pins are connected to the wires `CLBLM_R_X1Y97/CLBLM_M_A6` and `CLBLM_R_X11Y97/CLBLM_M_AX`. Starting from the source, wires are traversed downward (via wire connections) until the target wires are reached. Listing A.14 demonstrates the basic usage of `RouteTree`s in RapidSmith2. The **DesignAnalyzer**, **AStarRouter**, and **HandRouter** examples in the RapidSmith2 repository also demonstrate how to traverse and build a `RouteTree`.

Listing A.14: Building a RouteTree

```java
// Find a Wire to start the RouteTree at
Site site = device.getSite("SLICE_X5Y84");
SitePin pin = site.getSitePin("DQ");
Wire startWire = sink.getExternalWire();

// Create the first node in the RouteTree
Queue<RouteTree> rtQueue = new LinkedList<RouteTree>();
RouteTree start = new RouteTree(startWire);
rtQueue.add(start);

// Build up the RouteTree somehow
while (!amDone()) {
   RouteTree current = rtQueue.poll();
   Wire wire = route.getWire();

   for (Connection conn : wire.getWireConnections()) {
      // add qualified connections to the RouteTree
      if (isQualified(wire)) {
            RouteTree tmp = current.addConnection(conn);
            rtQueue.add(tmp);
        }
    }
}
```

When a design is imported from Vivado through a RSCP, the routing information is parsed and loaded into `RouteTrees` for each `CellNet`. On design export, the `RouteTree` for each `CellNet` is traversed and converted into a Vivado `ROUTE` string. Users can use custom data structures to route a design, but they needs to be **converted to an equivalent RouteTree representation** before exporting the design to Vivado.

A.4.5 Three Part Routing

In RapidSmith2, there are three sections to a routed `CellNet`:

1. The portion of the net that starts at the source BEL pin, and is routed to an output site pin. This part of the route exists completely inside of site boundaries.
2. The portion of the net that starts at the output site pin of part (1), and is routed to several sink site pins. This part of the route is called the **intersite** route because it connects sites together. A typical router is responsible for routing this section of the net[3].
3. The portion of the net that starts at the sink site pins from part (2), and is routed to sink BEL pins. Since there can be several sink pins in a `CellNet`, this section of the net can have more than one component. Each component exists completely inside site boundaries.

Figure A.12 shows a visual representation of the three-part routing structure. Each section of a route has a corresponding `RouteTree` object. A source `RouteTree` represents the orange wires in the figure (part 1), an intersite `RouteTree` represents the green wires in the figure (part 2), and a list of sink `RouteTrees` represents the purple wires in the figure (part 3, with a different `RouteTree` object for each site). It is important to note that intrasite nets only have a source `RouteTree`

[3]VCC and GND nets don't follow this pattern. The only difference for VCC and GND is that they can have multiple intersite nets.

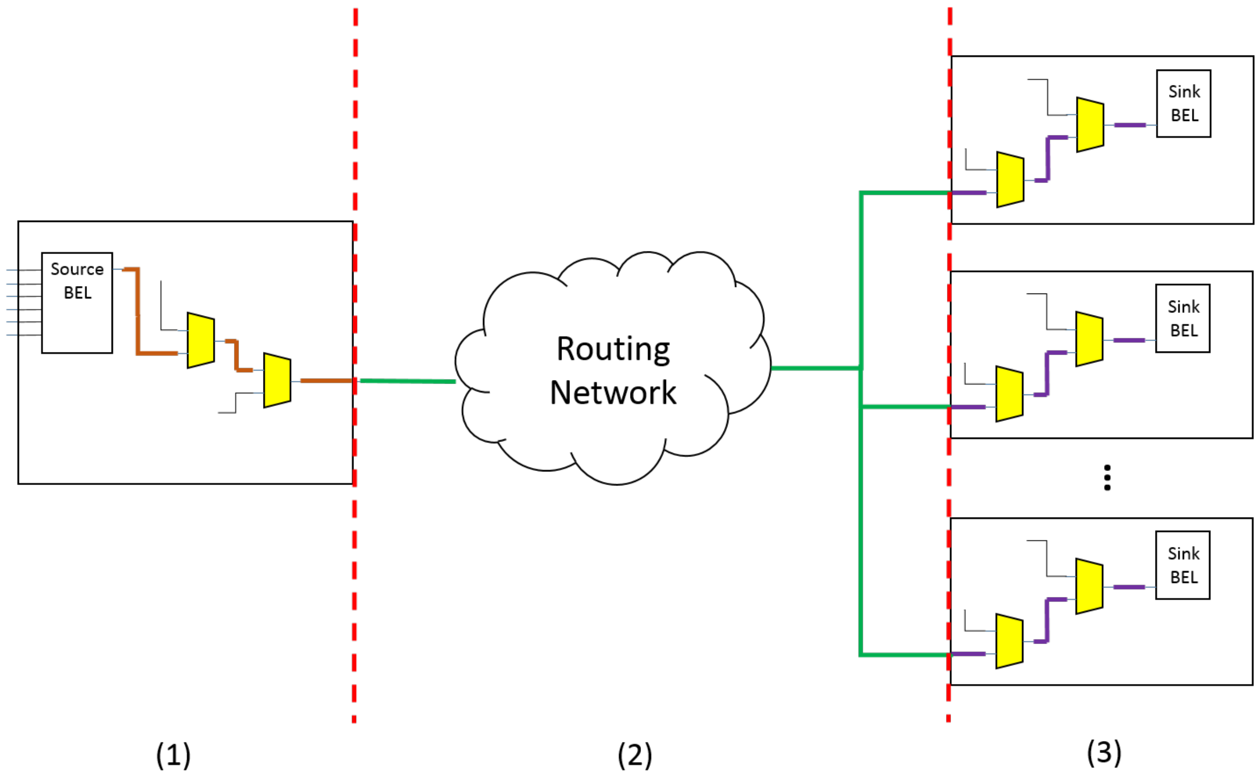


Figure A.12: Three-Part Routing

because they are completely contained within a site. Listing A.15 demonstrates how to utilize three-part routing in RapidSmith2. On design import, the routing sections of each `CellNet` are created automatically.

Listing A.15: Demonstration of three-part routing in RapidSmith2

```
// Get a handle to a routed net in the design
CellNet net = design.getNet("myNet");

// Handling the source RouteTree
RouteTree source = net.getSourceRouteTree();
net.setSourceRouteTree(createSourceRoute());

// Handling the intersite RouteTree
RouteTree intersite = net.getIntersiteRouteTree();
net.addIntersiteRouteTree(createIntersiteRoute());
```

```

// Iterate over a list of sink RouteTrees
for (RouteTree rt : net.getSinkSitePinRouteTrees()) {
   // do something with the RouteTree
}

// Or, get a RouteTree based on a SitePin
for(SitePin sitePin : net.getSitePins()) {
   if (sitePin.isInput()) {
      RouteTree sinkTree = net.getSinkTree(sitePin);
      // do something with the RouteTree
   }
}

// Add a new sink RouteTree that starts at a SitePin
net.addSinkRouteTree(sitePin, createSinkRouteTree(sp));
```

A.4.6 Intrasite Routing

On design import, the site PIP information extracted from Vivado is stored into RapidSmith2 data structures, and used to reconstruct the three-part routing view described in the previous section. This gives the user two options when dealing with intrasite routing in RapidSmith2: (1) use the three-part routing data structures, or (2) use the set of enabled site PIPs stored in the `CellDesign`. It is user preference for which representation to use when writing a CAD tool, but both representations need to be up-to-date before design export. Listing A.16 demonstrates how a set of used site PIPs can be created and added to a site. This step needs to be taken **only when you have modified the intrasite routing** for a site.

Listing A.16: Code to transform a set of SiteWires into Site PIPs

```
// Get a handle to a Design and a Site
CellDesign design = tcp.getDesign();
```

```java
Device device = tcp.getDevice();
Site site = device.getSite("SLICE_X5Y84");

// Get a list of used site wires somehow (this is up to you)
Set<Wire> usedSiteWires = getUsedWires(site);

// Convert the list of wires to their integer enumeration
Set<Integer> usedPipWires = usedSiteWires.stream()
                            .map(w -> w.getWireEnum())
                            .collect(Collectors.toSet());

// Set the used site pips with the design class
design.setUsedSitePipsAtSite(site, usedPipWires);
```

A.5 Design Import/Export

After a Vivado design has been converted to a RSCP using the `Tincr` command `[::tincr::write_rscp]` , the RSCP can be loaded into RapidSmith2 using the code shown on lines 2-5 in Listing A.17.

Listing A.17: How to import and export TCP files to and from RS2

```
1   // Loading a Tincr Checkpoint
2   TincrCheckpoint tcp = VivadoInterface.loadTcp("pathToCheckpoint.tcp");
3   CellDesign design = tcp.getDesign();
4   Device device = tcp.getDevice();
5   CellLibrary libCells = tcp.getLibCells();
6
7   // Insert CAD Tool Here
8
9   // Exporting the modified design to a Tincr Checkpoint
10  VivadoInterface.writeTCP("pathToStore.tcp", design, device, libCells);
```

While a design is being imported into RapidSmith2, several useful additional data structures are built up. To gain access to those data structures, you can pass an additional argument into the `VivadoInterface::loadTCP()`, as shown in. Listing A.18.

Listing A.18: Importing a TCP with additional information

```
// Loading a Tincr Checkpoint with additional info
TincrCheckpoint tcp = VivadoInterface.loadTcp("PathToCheckpoint.tcp", true);
Collection<BelRoutethrough> belRts = tcp.getRoutethroughObjects();
Collection<Bel> staticSources = tcp.getStaticSourceBels();
Map<BelPin, CellPin> belPinToCellPinMap = tcp.getBelPinToCellPinMap()
```

Line 10 of Listing A.17 demonstrates how to export a design from RapidSmith2, which produces a Tincr Checkpoint (TCP). To import the TCP back into Vivado, simply open Vivado in Tcl mode and run the command `[tincr::read_tcp myCheckpoint.tcp]`

A.6 Installing New Device Files

The device files included with the RapidSmith2 installation (listed in subsection A.1.3) have been well-tested, and are great starting points for new users. However, RapidSmith2 also supports installing new device files for parts not listed in subsection A.1.3. The required files that need to be generated from `Tincr` to support a new device in RapidSmith2 is given in chapter 3, and so will not be discussed further here. The remainder of this section documents the required steps to transform the `Tincr` intermediate files into compact device files that can be loaded into RapidSmith2.

A.6.1 Creating New Device Files for Supported Families

Section A.1.3 gives a list of currently supported families in RapidSmith2. If the device to install is **not** within a supported family, see subsection A.6.2 for how to add support for a new family in RapidSmith2. Otherwise, a new device file can be added in four easy steps:

1. Open Vivado in Tcl mode, and execute the `Tincr` command `[::tincr::write_xdlrc]`. An example usage is shown in Listing A.19 for the Artix7 part *xc7a100tcsg324-3*. The "-max_processes" option is used to parallelize the operation so that it will execute faster. This Tcl command can take a very long time to run (more than 24 hours for very large devices). Running the command on a remote machine is good practice. Be aware that these XDLRC files are massive, and 100 GB for the largest XDLRC files is not uncommon. Make sure there is enough space on the hard drive before generating the XDLRC for a device.

Listing A.19: XDLRC Generation Example

```
::tincr::write_xdlrc -part xc7a100tcsg324-3 -max_processes 4 -primitive_defs
    xc7a100tcsg324_full.xdlrc
```

2. Run the `Tincr` command `[tincr::create_xml_device_info]` to create a *deviceInfo.xml* file for the part. Copy the resulting file to the "*devices/family*" directory of the RapidSmith2 installation, where family is replaced with the appropriate Vivado family name (i.e. kintexu).

3. Run the device installer in RapidSmith2 and pass the newly created XDLRC as an argument. An example command line usage is shown in Listing A.20. The device installer creates compact device files that represent a Xilinx device from the XDLRC and *deviceInfo.xml* generated in the previous steps. Notice the two JVM command line arguments used in the command. The first option ("-ea") enables assertions for the code. It is important to include this flag so that device file errors can be caught during parsing. The second option ("-Xmx4096m") sets how much memory the JVM can use while running the installer. Since XDLRC files are quite large, the memory usage of the installer grows very quickly. If the device installer fails with an out of memory exception, you will need to increase the memory and re-run the installer (up to 32 GB of memory may be required).

Listing A.20: RapidSmith2 device installer example usage

```
java -ea -Xmx4096m edu.byu.ece.rapidSmith.util.Installer --generate file
   xc7a100tcsg324_full.xdlrc
```

4. Run the family builder in RapidSmith2 and pass the name of the newly created part as a command line argument. An example usage is shown in Listing A.21 for an Artix7 device. An `Artix7.java` file (or whatever family your device is in) will already exist, but will be updated with new sites and tile types from the newly installed part.

Listing A.21: Family builder example usage

```
java edu.byu.ece.rapidSmith.util.FamilyBuilders xc7a100tcsg324
```

Once the device installer is done executing, the compact devices files are stored in the corresponding family directory of the RapidSmith2 "devices" folder. For example, the device files generated from the example part *xc7a100tcsg324-3* are stored in the "artix7" sub-directory. Listing A.22 shows the two device files that are created after the device installer is run. The file ending in "_db.dat" contains the serialized `Device` data structures for RapidSmith2. The file ending in "_info.dat" contains additional serialized data (such as reverse wire connections) that can be optionally loaded with the device.

Listing A.22: Generated RapidSmith2 device files

```
[ttown523@CB461-EE09968:artix7] ls
cellLibrary.xml familyInfo.xml xc7a100tcsg324_db.dat xc7a100tcsg324_info.dat
```

A.6.2 Supporting New Device Families

Vivado supports implementing FPGA designs on devices for the following families (also called architectures):

- **Artix7 (artix7)**
- **Kintex7 (kintex7)**
- **Virtex7 (virtex7)**
- **Zynq (zynq)**
- **Kintex Ultrascale (kintexu)**
- **Virtex Ultrascale (virtexu)**
- Kintex Ultrascale+ (kintexuplus)
- Virtex Ultrascale+ (virtexuplus)

The name in parentheses is the Vivado Tcl name for the family. Bolded items are families that are currently supported in RapidSmith2 and `Tincr`. To add RapidSmith2 support for another Vivado family, follow the steps listed below.

1. Create the primitive definitions of the family using VSRT. The VSRT user guide is given in Appendix B.
2. Copy the primitive definitions created in step (1) to the directory *tincrPath/cache/family/primitive_defs*, where "tincrPath" is the path to your `Tincr` installation and "family" is the Vivado Tcl name for the family of the primitive defs just generated.
3. Create the *familyInfo.xml*. To do this, open Vivado in Tcl mode and run the command `[ ::tincr::create_xml_family_info]`. An example usage of the command is shown in Listing A.23 for Kintex UltraScale. As the listing shows, there are three arguments to the command:
    - **familyInfo.xml**: The file name to store the generated family info. The file ending ".xml" will be appended if it is not included.

- **kintexu**: The Vivado family name.
- **addedBels.txt** (Optional): The "addedBels.txt" file that was created during step (1). This file contains a list of added VCC/GND BELs for each family.

Listing A.23: Family info example usage

```
::tincr::create_xml_family_info familyInfo.xml kintexu addedBels.txt
```

4. Modify the generated family info with a few hand edits. The required hand edits are broken down between Series7 and UltraScale devices in subsection A.6.3 and subsection A.6.4 respectively.
5. Follow the steps laid out in subsection A.6.1 to generate RapidSmith2 device files.
6. Run the Family Builder in RapidSmith2 (an example usage is shown in Listing A.21). The Family Builder accepts one command line argument: a part name of a device in the family. Using the device files for the specified part, a Java file is created that contains all tile types and site types within the part. For example, the command in Listing A.21 will generate an `Artix7.java` file which can be used to find site and tile types as shown in Listing A.24. Every family Java class includes a classifications section with the header "/* —— CLASSIFICATIONS GO HERE —— */". Below the header, tile and site classifications can be manually added to group similar site types together. The classifications for Artix7 are shown in Listing A.25 for reference.

Listing A.24: How to access SiteTypes and TileTypes in RapidSmith2

```
SiteType siteType = Artix7.SiteTypes.SLICEL;
TileType tileType = Artix7.TileTypes.CLBLL_L;
```

Listing A.25: Device classifications example

```
/* ------ CLASSIFICATIONS GO HERE ------ */
        _CLB_TILES.add(TileTypes.CLBLL_L);
        _CLB_TILES.add(TileTypes.CLBLL_R);
        _CLB_TILES.add(TileTypes.CLBLM_L);
        _CLB_TILES.add(TileTypes.CLBLM_R);

        _SWITCHBOX_TILES.add(TileTypes.INT_L);
        _SWITCHBOX_TILES.add(TileTypes.INT_R);

        _BRAM_TILES.add(TileTypes.BRAM_L);
        _BRAM_TILES.add(TileTypes.BRAM_R);

        _DSP_TILES.add(TileTypes.DSP_L);
        _DSP_TILES.add(TileTypes.DSP_R);

        _IO_TILES.add(TileTypes.LIOB33_SING);
        _IO_TILES.add(TileTypes.LIOB33);
        _IO_TILES.add(TileTypes.RIOB33);
        _IO_TILES.add(TileTypes.RIOB33_SING);

        _SLICE_SITES.add(SiteTypes.SLICEL);
        _SLICE_SITES.add(SiteTypes.SLICEM);

        _BRAM_SITES.add(SiteTypes.RAMB18E1);
        _BRAM_SITES.add(SiteTypes.RAMB36E1);
        _BRAM_SITES.add(SiteTypes.RAMBFIFO36E1);

        _FIFO_SITES.add(SiteTypes.FIFO18E1);
        _FIFO_SITES.add(SiteTypes.FIFO36E1);
        _FIFO_SITES.add(SiteTypes.IN_FIFO);
        _FIFO_SITES.add(SiteTypes.OUT_FIFO);
        _FIFO_SITES.add(SiteTypes.RAMBFIFO36E1);

        _DSP_SITES.add(SiteTypes.DSP48E1);
```

```
    _IO_SITES.add(SiteTypes.IOB33);
    _IO_SITES.add(SiteTypes.IOB33S);
    _IO_SITES.add(SiteTypes.IOB33M);
    _IO_SITES.add(SiteTypes.IPAD);
    _IO_SITES.add(SiteTypes.OPAD);
```

Once these steps are complete, RapidSmith2 will have full support for the generated family. This means that device files for any part within the family can be created.

A.6.3 Series7 Family Info Hand Edits

Due to complications with Vivado's Tcl interface, several hand edits are required to complete Series7 family info files. RapidSmith2 already provides support for all Series7 families, but the required manual edits are documented here in case they need to be regenerated in the future.

1. The first hand edit is to remove invalid alternate types. The only way to determine invalid alternate types in Vivado is to go site-by-site in the family info, select an instance of the site type in Vivado's device browser, and click the site type dropdown box (as shown in Figure 3.7). If there are any site types reported in the family info XML that are not shown in the GUI, they need to be removed from the XML. The Tcl commands shown in Listing A.26 can be used to select a specific site type in Vivado to view its alternate types.

Listing A.26: Tcl commands to select a Vivado Site object

```
Vivado% set site [lindex [get_sites -filter {SITE_TYPE==IPAD}] 0]
Vivado% select $site
```

2. The second hand edit is to add alternate type pin mappings. When a site is changed to one of its alternate types in Vivado, the site pins can be renamed. An example is shown in Figure 3.9 for an IDELAYE3 site that has been changed to the alternate type ISERDESE2. Notice how the "SR" site pin has been renamed to "RST" in the figure. Unfortunately these pin renamings cannot be automatically extracted from Vivado's Tcl interface, and so must

be added manually. Listing A.27 shows how to add pin renamings to the family info XML using the "pinmaps" tag. To determine the actual pin mappings, the first step is to open two instances of the Vivado GUI. For each site in the family info, load the default type in one Vivado instance, load each alternate type in the other instance, and visibly check what pins are renamed in the alternate type (as demonstrated in Figure 3.9). Table A.2 gives a list of all alternate types that rename pins for Artix7 devices.

Listing A.27: Sample pinmaps in a family info file

```
<name>ILOGICE3</name>
<alternatives>
  <alternative>
    <name>ILOGICE2</name>
    <pinmaps>
    </pinmaps>
  </alternative>
  <alternative>
    <name>ISERDESE2</name>
    <pinmaps>
     <pin>
       <name>RST</name>
       <map>SR</map>
     </pin>
    </pinmaps>
  </alternative>
</alternatives>
```

3. The third hand edit is to remove invalid mux corrections. In some cases, BELs might be incorrectly tagged as "routing muxes" or "polarity selectors" even though they are not. This issue has mostly been fixed, but it is still good practice to examine all mux corrections in the family info and verify that they are correct.

Table A.2: Artix7 Alternate Pin Mappings

Default Type	**Alternate Type**
FIFO18E1	RAMB18E1
ILOGICE3	ISERDESE2
IOB33M	IPAD
IOB33S	IPAD
OLOGICE3	OSERDESE2
RAMBFIFO36E1	FIFO36E1, RAMB36E1

4. The final hand edit is to add missing compatible types. Some compatible types can be automatically generated from Vivado, but not all. This means that the missing compatible types must be added manually. The next section describes in more detail how to add compatible types to the Family Info.

A.6.4 UltraScale Family Info Hand Edits

UltraScale and later devices require only a single hand edit: adding missing compatible types (most compatible types can be determined automatically). The XML listing given in Listing A.28, shows the two compatible types that were manually added to complete the Kintex UltraScale family info. Other device families may require additional compatible site hand edits. It is up to the user to determine what compatible sites need to be added through experimentation.

Listing A.28: Manually added compatible sites for UltraScale devices

```
<site_type>
  <name>SLICEL</name>
  <is_slice/>
  <compatible_types>
    <compatible_type>SLICEM</compatible_type>
  </compatible_types>
  ...
</site_type>
<site_type>
```

```
  <name>HRIO</name>
  <is_iob/>
  <compatible_types>
    <compatible_type>HPIOB</compatible_type>
  </compatible_types>
  ...
</site_type>
```

APPENDIX B. VSRT DOCUMENTATION

This appendix contains a complete user guide for the VSRT tool introduced in subsection 3.1.3. VSRT may be updated in the future, and so this appendix serves as a snapshot of the current state of VSRT.

B.1 Introduction

The Xilinx Design Language (XDL) is a command line interface into Xilinx's ISE tool suite. Using a single command `xdl`, both device and design information can be extracted from ISE for external use. Device information (the physical components of the FPGA) is extracted via `XDLRC` files, and design information (the logical netlist, placement, and routing) is extracted via `XDL` files. Many FPGA CAD research projects have built upon the XDL interface, resulting in several useful tools capable of targeting commercial devices. With the release of Vivado, however, Xilinx no longer supports XDL. This makes external tools and frameworks that rely on the interface incompatible with next-generation Xilinx devices (Ultrascale, Ultrascale+, and beyond). Instead, Vivado includes a `Tcl` interface that grants access to Xilinx's internal device and design data structures.

Tincr, a Vivado `Tcl` plugin, defines a set of APIs that provide the same functionality as XDL did for ISE. Device and design information can be extracted from Vivado using the following `Tincr` commands:

- *write_xdlrc*: Creates a `XDLRC` file which describes the physical components of a Vivado FPGA part.
- *write_rscp*: Creates a RapidSmith Checkpoint (RSCP) which describes all relevant aspects of a Vivado design.

Due to limitations of Vivado's Tcl API, the `XDLRC` files generated from *write_xdlrc* are incomplete. They are missing the primitive definition section. A primitive definition (also called primitive def) defines all pins, BELs, and routing muxes inside of a site, and how the components are connected through site wires (section B.3 has a more detailed description of the terminology used here). Using `Tincr` and Vivado Tcl commands, a set of *partial* primitive defs can be created. These partial primitive defs include everything *except for* the **connections** between primitive def components, which is crucial for any external CAD tool.

The Vivado Subsite Routing Tool (VSRT) is a GUI application that helps users add these missing connections to create complete primitive definitions for Vivado devices. The general flow for using the tool is shown in Figure B.1.

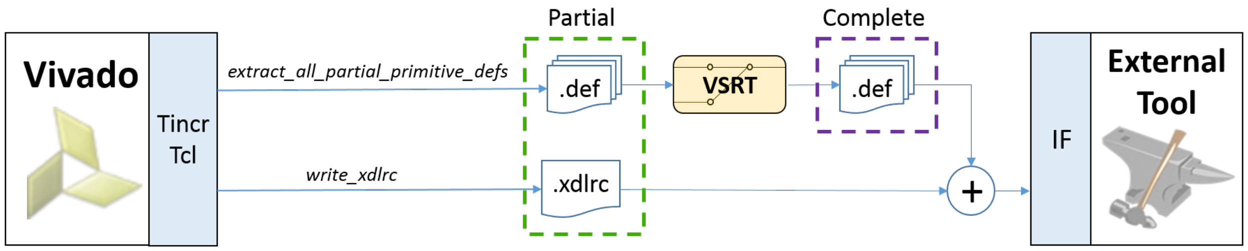


Figure B.1: XDLRC Creation Flow

As the figure shows, `Tincr` first generates a set of partially complete primitive definitions (one for each site in the device architecture). The partial primitive defs are then loaded into VSRT, where a user processes each site one-by-one. To use the tool, a user brings up a picture of the primitive site in both Vivado's device browser and the VSRT GUI. They then use VSRT to manually draw the connections visible in Vivado's device browser. This may seem like a tedious process, but there are two important observations to note:

1. Primitive definitions are at least identical for parts that have the same architecture. For example, any part within the Kintex UltraScale family will use the same set of primitive definitions. You don't need to create new primitive defs for each part. **NOTE**: This is also most likely true across series (i.e. all UltraScale parts regardless of architecture use the same set of primitive defs)

2. Many primitive definition connections can be inferred automatically in Tcl even though the site wires cannot be explicitly accessed. This is especially true for sites that are dominated by a single BEL. This reduces the workload for VSRT users and makes the process of creating primitive defs for new architectures manageable.

The remainder of this document describes important aspects of VSRT. Section B.2 gives installation instructions. Section B.3 describes important terminology for VSRT. Section B.4 gives a user guide of how to use the tool. Section B.5 describes the features of the tool and how to use VSRT most effectively. Section B.6 lists all currently known issues.

B.2 Installation

B.2.1 Tool Facts

- Written in Java 8
- Built on the QtJambi 4.6 GUI framework
- Supported Operating Systems:
  - Linux 32-bit and 64-bit
  - Windows 32-bit and 64-bit
  - MacOSX
- Tested on:
  - Fedora 20
  - Windows 7/8/10

B.2.2 Requirements

- Vivado 2016.2
- Tincr: `https://github.com/byuccl/tincr`
- RapidSmith2: `https://github.com/byuccl/RapidSmith2`
- JDK 1.8 (earlier versions may work, but have not been tested)
- Eclipse or IntelliJ IDE (optional)

B.2.3 Steps

VSRT is a part of the RapidSmith2 repository linked above. To install and run VSRT, simply follow the RapidSmith2 installation instructions. The main VSRT java class can be found under the RapidSmith2 package `edu.byu.ece.rapidSmith.device.vsrt.gui.VSRTool`, and can be run via the command line or an IDE (recommended). To generate the partial primitive definitions, you will need to install Tincr (instructions are in the Tincr link above), and Vivado 2016.2. Once you have Vivado, Tincr, and RapidSmith2 setup correctly, you are ready to use VSRT. Section B.4 gives more information about how to use the tool.

B.3 Terminology

This section introduces important terminology used in VSRT (or links to locations that describes the terminology). If you are comfortable with Xilinx FPGA terminology and the contents of XDLRC files, you can skip subsections B.3.1 and B.3.2.

B.3.1 Vivado Devices and XDLRC files

VSRT uses the same terminology as Vivado to describe FPGA devices. Therefore it is important to understand Xilinx device terminology to fully utilize VSRT. The “Device” section of the RapidSmith2 Tech Report (found at `https://github.com/byuccl/RapidSmith2/tree/master/doc`) describes Xilinx FPGA architecture and XDLRC in greater detail. Readers are referred to that document for more information.

B.3.2 Primitive Definitions

A primitive definition is a textual description of all site pins, BELs, and routing muxes inside of a primitive site, and how they are connected through site wires. The remainder of this section details how the components of a primitive site are represented in a primitive def file. It is important to note that the partial primitive defs generated from `Tincr` do not include any connections (“conn” keyword). The purpose of VSRT is to generates these.

Header

```
(primitive_def BUFHCE 3 8
```

Every primitive definition starts with a header line (as shown above). This line tells three important things about the primitive def: (1) the name of the primitive site that it represents, (2) the number of site pins on the corresponding primitive site, and (3) the number of elements defined in the primitive def. Element is a generic term used to represent a “component” of the primitive site. Essentially, everything within the primitive site ends up being an element in the resulting primitive def (this will be shown in the following subsections).

Site Pins

```
// Site pins first show with the "pin" keyword
(pin O O output)
(pin I I input)
(pin CE CE input)
...
// They also show up later in the primitive def as "elements"
(element O 1
   (pin O input)
   (conn O O <== BUFHCE O)
)
(element I 1
   (pin I output)
   (conn I I ==> BUFHCE I)
)
(element CE 1
   (pin CE output)
   (conn CE CE ==> CEINV CE)
   (conn CE CE ==> CEINV CE_B)
)
```

The pins of a site show up in two places in the primitive def file. The first is the pin section, which appears directly after the header. Each pin is defined with a name and a direction. Site pins are also defined as elements, where the connections of the pin are included. It is important to note that the direction of the "element" pin is the opposite of the direction found in the pin section. The element pin direction represents the pin's direction viewed from the internal components of the site, while the other direction represents the pin's direction external to the site.

BELs

```
(element BUFHCE 3 # BEL
    (pin O output)
    (pin I input)
    (pin CE input)
    (conn BUFHCE O ==> O O)
    (conn BUFHCE I <== I I)
    (conn BUFHCE CE <== CEINV OUT)
  )
```

BELs within a primitive site are represented as elements in the primitive def with the special token "# BEL" included after the name of the BEL. The BEL pins and their connections are defined within the element.

Site PIPs (Routing Muxes)

```
  (element CEINV 3
    (pin OUT output)
    (pin CE input)
    (pin CE_B input)
    (cfg CE CE_B)
    (conn CEINV OUT ==> BUFHCE CE)
    (conn CEINV CE <== CE CE)
    (conn CEINV CE_B <== CE CE)
  )
```

Site PIPs (or routing muxes) are defined exactly the same way as BELs, but they **do not** have the "# BEL" token after the name. Elements that do not have the BEL token, and do not have a corresponding primitive def pin with the same name are distinguished as routing muxes.

Configuration Options

```
  (element CE_TYPE 0
    (cfg SYNC ASYNC)
```

```
)
```

Site-level and BEL-level configuration parameters can also be included in the primitive def file. They are also represented as elements, but they only have a single “cfg” child, which defines the possible values of the configurable parameter. These elements are generally not important for VSRT, and can be safely ignored.

Routethroughs

```
(element _ROUTETHROUGH-I-O 2
   (pin I input)
   (pin O output)
)
```

Routethrough elements in the primitive def represent a configurable connection between an input site pin and an output site pin. These connections can be used in external tools instead of trying to configure the necessary routing muxes of the site to create the routethrough. For VSRT, these elements can be ignored because they can be automatically generated from Vivado’s Tcl interface.

B.3.3 VSRT GUI

Figure B.2 shows the terminology used to describe the VSRT GUI. This terminology will be used through the rest of the document, so it is important to briefly review the figure before continuing.

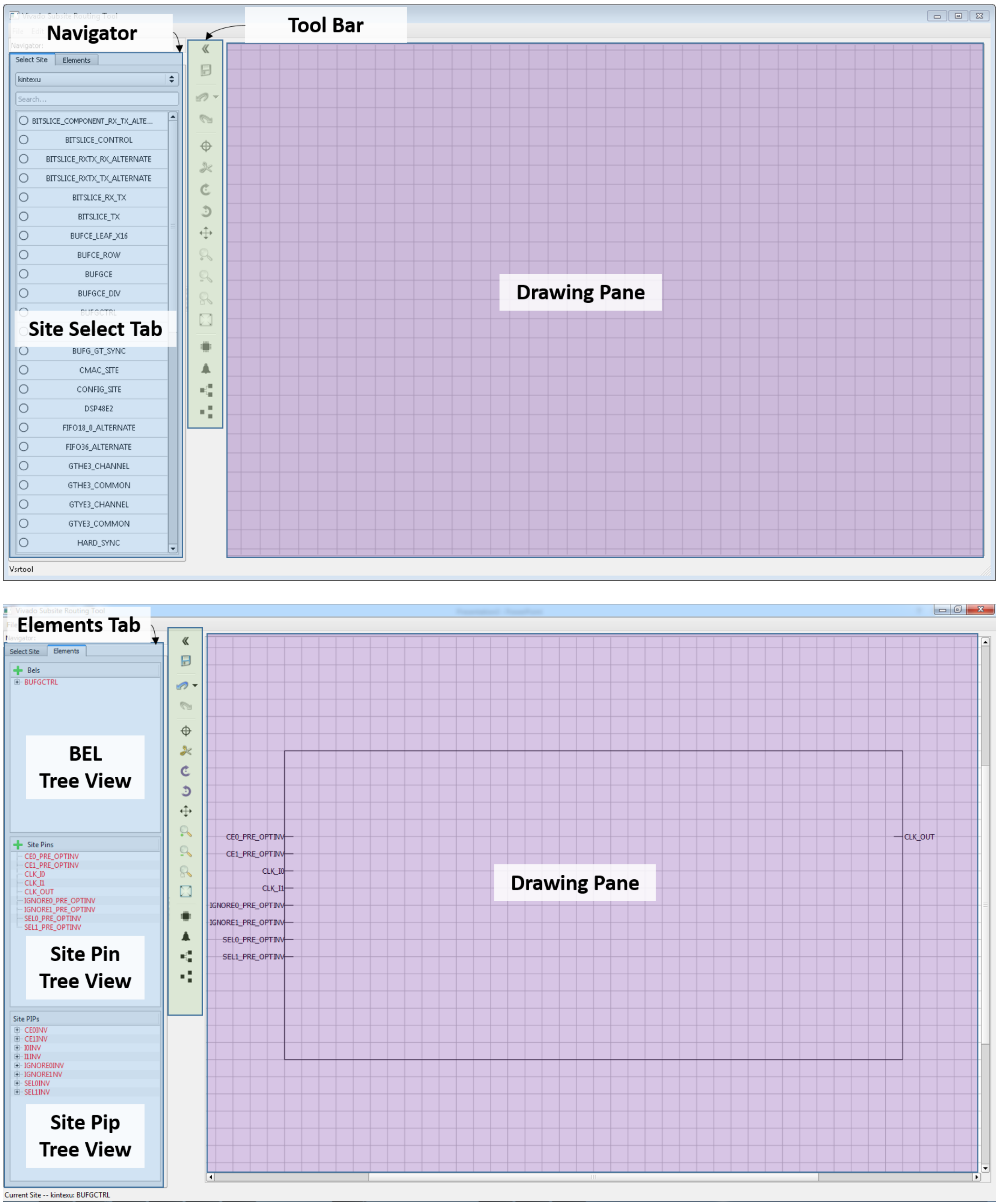


Figure B.2: VSRT GUI Start View (Top) and Primitive Site View (Bottom)

B.4 User Guide

Before running VSRT, you will need to generate the partial primitive definitions from Vivado. To do this, open a new terminal and start Vivado in "Tcl" mode. Make sure the Vivado executable is on your PATH.

```
[ttown523@CB461-EE09968:~] vivado -mode tcl


****** Vivado v2014.2 (64-bit)
  **** SW Build 928826 on Thu Jun 5 17:55:10 MDT 2014
  **** IP Build 924643 on Fri May 30 09:20:16 MDT 2014
    ** Copyright 1986-2014 Xilinx, Inc. All Rights Reserved.


Vivado%
```

Next, run the `Tincr` command `[get_all_partial_primitive_defs]` in the Vivado prompt. The first argument to this command is the directory to store the partial primitive defs (if it does not yet exist, it will be created). The second (optional) argument is the architecture to generate primitive defs for. Possible architectures for Vivado 2016.2 are: `artix7`, `kintex7`, `virtex7`, `zynq`, `kintexu`, `virtexu`, `kintexuplus`, and `virtexuplus`. If no architecture is specified, then the primitive defs for all architectures will be generated. An example usage of this command with sample output is shown below. Replace "primitiveDefs/" with the directory of your choice and "kintexu" with the architecture of your choice. **NOTE**: This command may take a few hours to run if you are generating partial primitive defs for all architectures.

```
Vivado% tincr::extract_all_partial_primitive_defs primitiveDefs/ kintexu
...
Extracting Primitive Sites from part xqku060-rfa1156-1-i (28 out of 31)...
Extracting Primitive Sites from part xqku095-rfa1156-1-i (29 out of 31)...
Extracting Primitive Sites from part xqku115-rld1517-1-i (30 out of 31)...
Extracting Primitive Sites from part xqku115-rlf1924-1-i (31 out of 31)...
Successfully created all .def files in directory primitiveDefs/
```

```
Vivado%
```

Once the command is finished running, navigate to the generated partial primitive def directory and view the newly created files. If you generated the primitive defs for all architectures, it will look something like the directory structure shown below. There is a folder for each architecture that holds all partial primitive defs for that architecture.

```
[ttown523@CB461-EE09968:primitiveDefs] ls
artix7 kintex7 kintexu kintexuplus log.txt virtex7 virtexu virtexuplus zynq
```

Also included in the directory is a *log.txt* file. An example log file is shown below.

```
Extracting Primitive Sites from part xcku025-ffva1156-1-c (1 out of 31)...
   kintexu-SLICEL -> SLICE_X0Y179...
   kintexu-SLICEM -> SLICE_X1Y179...
   kintexu-RAMB181 -> RAMB18_X0Y71...
   ...
   ALTERNATE: kintexu-FIFO18_0 -> RAMB18_X1Y70...
   ALTERNATE: kintexu-RAMB180 -> RAMB18_X2Y70...
```

This file is important to VSRT users because it gives a mapping between a site type, and a specific instance of that site type. This makes it easy to bring up a site in the Vivado device browser when drawing the corresponding connections in VSRT. The *log.txt* file also shows what site types are *alternate-only* sites (they include the keyword ALTERNATE). For these sites, an additional step must be taken in Vivado to show the correct connections. The Tcl commands in Listing B.1 can be used to visualize an instance of any site in Vivado's device browser.

Listing B.1: Tcl commands for viewing a site in Vivado's device browser

```
// open the part and start the device browser (only needs to happen once)
Vivado% link_design -part xcku025-ffva1156-1-c
Vivado% start_gui
```

```
// gain a handle to the site, and zoom to its view
Vivado% set site [get_sites RAMB18_X1Y70]
Vivado% select $site (you may have to press F9 to zoom to the site)

// set the site type (FOR ALTERNATE-ONLY SITES!)
Vivado% set_property MANUAL_ROUTING FIFO18_0 $site
```

Now that the partial primitive defs directory has been created, you are ready to use VSRT. To run VSRT, simply execute the Java class `edu.byu.ece.rapidSmith.device.vsrt.gui.VSRTool` found in the RapidSmith2 repository. VSRT accepts a single command line argument: the partial primitive def directory generated above. You can choose to run the tool on the command line, or in an IDE (which is recommended). Figure B.3 shows the VSRT GUI on startup if everything was setup correctly. VSRT can be run in two different modes: regular mode and single BEL mode. The remainder of this section gives a walkthrough of how to use both modes.

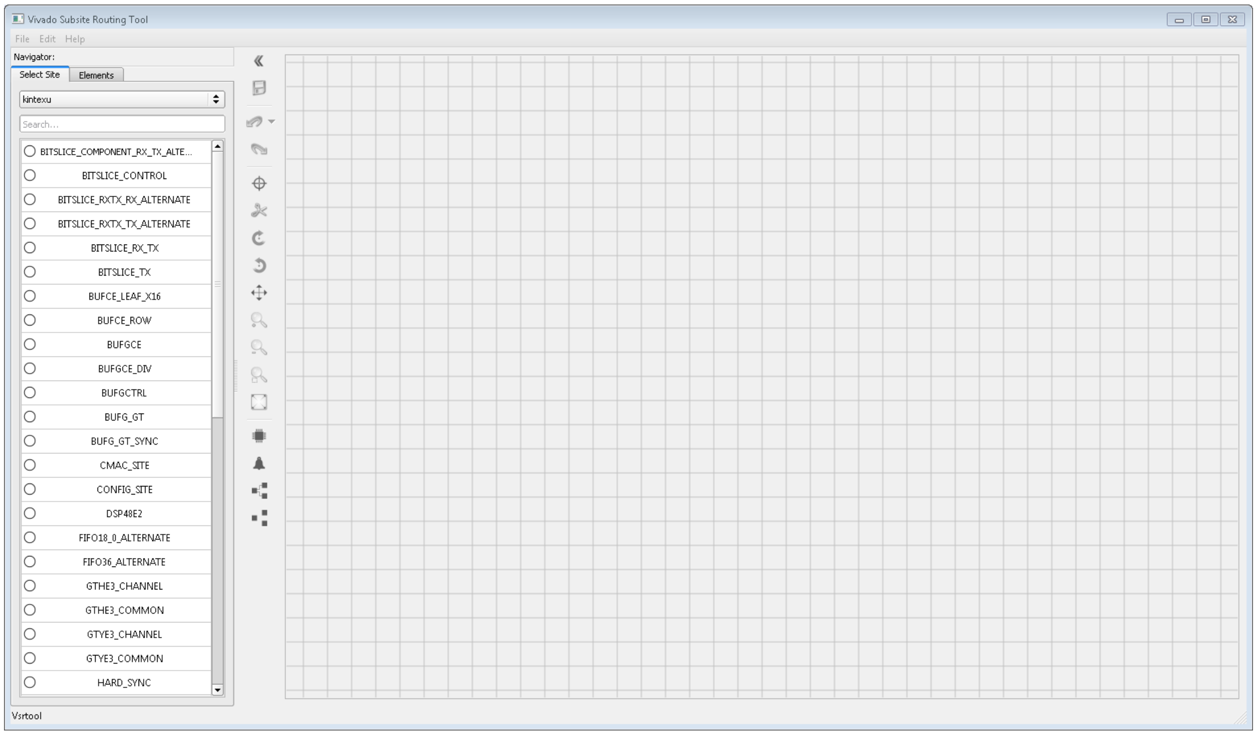


Figure B.3: VSRT GUI Start Screen

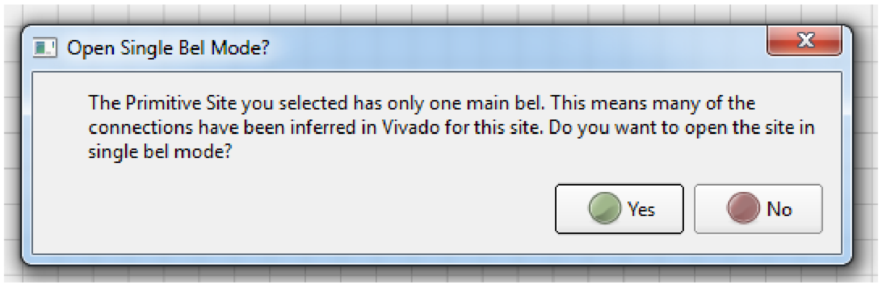
B.4.1 Regular Mode

Regular mode is the default mode for VSRT, and is intended for two use cases:

1. Sites where connections cannot be automatically inferred in Tcl (see subsection B.4.2 for these sites).

2. Small sites that can be wired up very quickly (even if some connections can be automatically generated.)

In this mode, the user is required to create all of the connections of the primitive site by manually drawing wires between components. This is best shown through an example. In the VSRT GUI start screen, select the "kintexu" architecture in the dropdown box and find the BUFGCTRL site in the table. Double click the site, and you should see the prompt shown in Figure B.4:


Open Single Bel Mode?

The Primitive Site you selected has only one main bel. This means many of the connections have been inferred in Vivado for this site. Do you want to open the site in single bel mode?

Figure B.4: User Mode Prompt

Since we want to run the tool in regular mode, select "No" and the BUFGCTRL site will load in VSRT as shown in Figure B.5.

As the screenshot shows, all BELs, site pins, and site pips (routing muxes) of the site are displayed in the "Elements" tab on the left. The red font color indicates that at least one pin attached to the element is unconnected. The drawing pane (gridded area) is initialized to include all site pins attached to the site, and a rectangle representing the site's boundaries. In this case, there is 8 input site pins and 1 output site pin. To add elements to the drawing pane, simply double click the item in the Elements tab. The item will appear in the top-left corner of the drawing pane, where you can then drag it to the location of your choice (shown in Figure B.6). To create the internal site connections, just add elements to the drawing pane, and

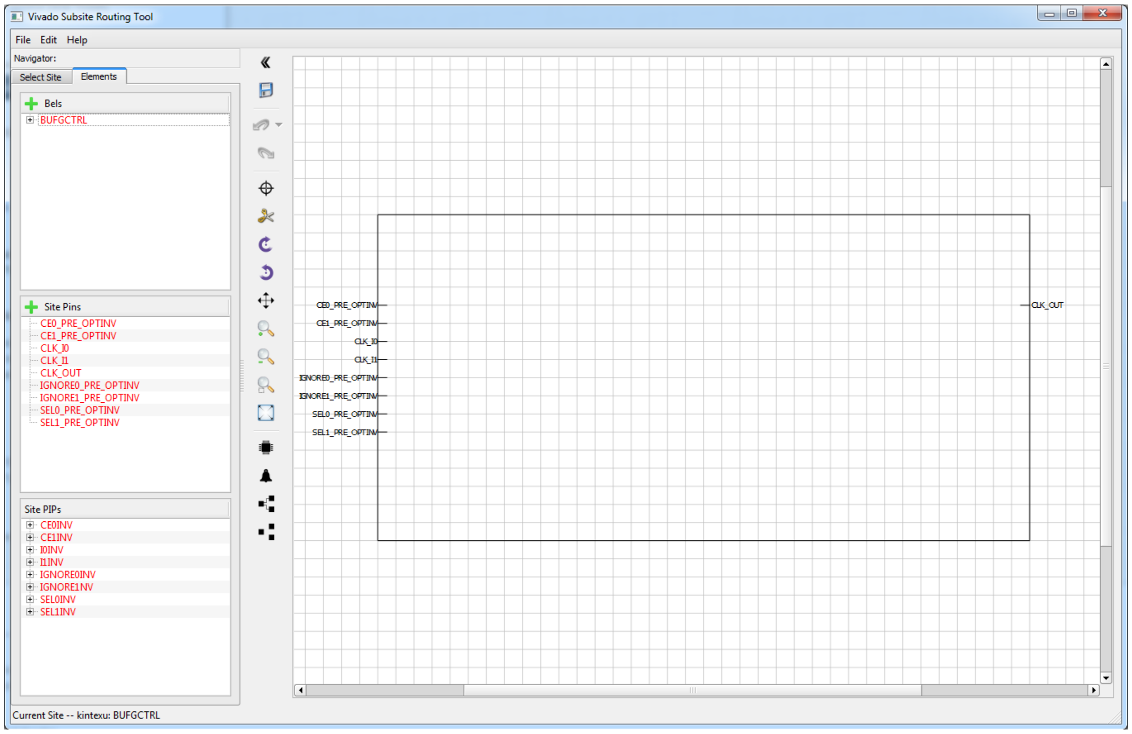


Figure B.5: BUFGCTRL Site in VSRT

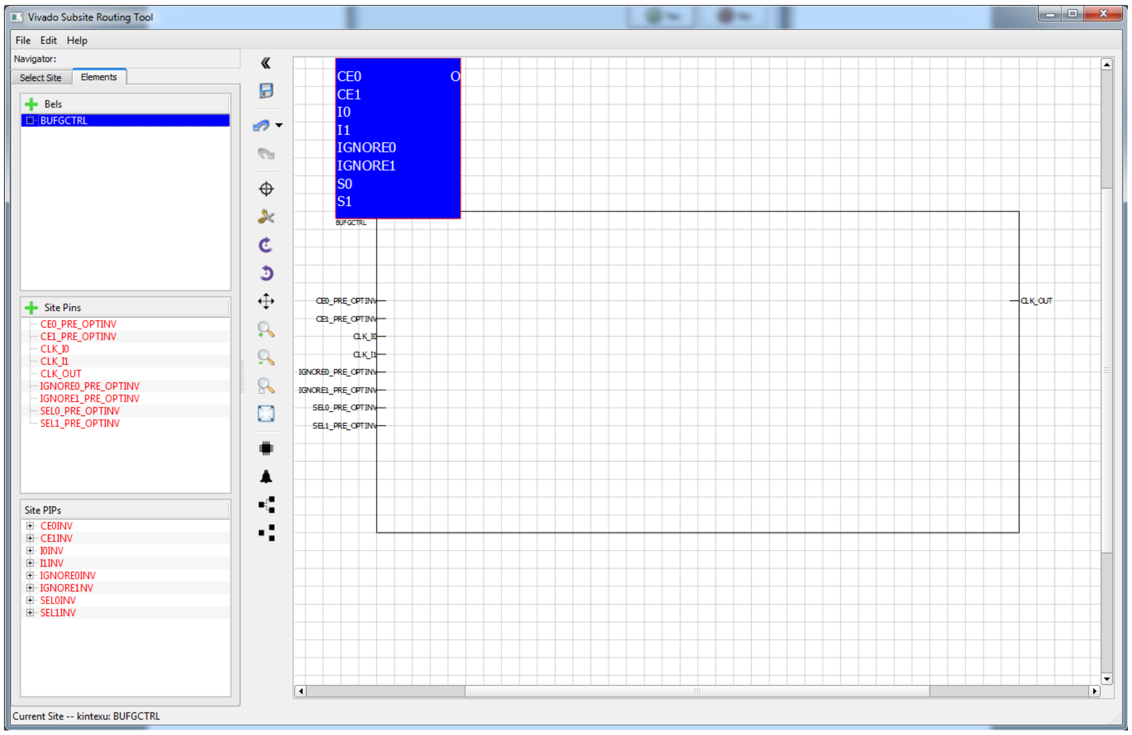


Figure B.6: Adding a BEL to the Drawing Pane

connect elements by drawing wires between their pins. To add a new wire, either press “w” (recommended), or click the draw wire button [⌖] on the tool bar. This will replace the cursor with crosshairs and put the tool in wire drawing mode.

As shown in Figure B.7, several things happen when two pins are connected with a wire. First, the tree view of the pin is updated to show the connection using the same format as it will appear in the primitive def file. Second, the pin in the element tab changes color from red to green (which signifies a connection has been made to the pin). A BEL or site PIP element will turn green once all of its pins have been connected. Third, the outline color of the shapes in the drawing pane will change. A green outline represents a fully connected element, orange represents an element that is partially connected, and red represents an element that is completely unconnected. This color coding makes it easier to determine what remaining connections need to be generated.

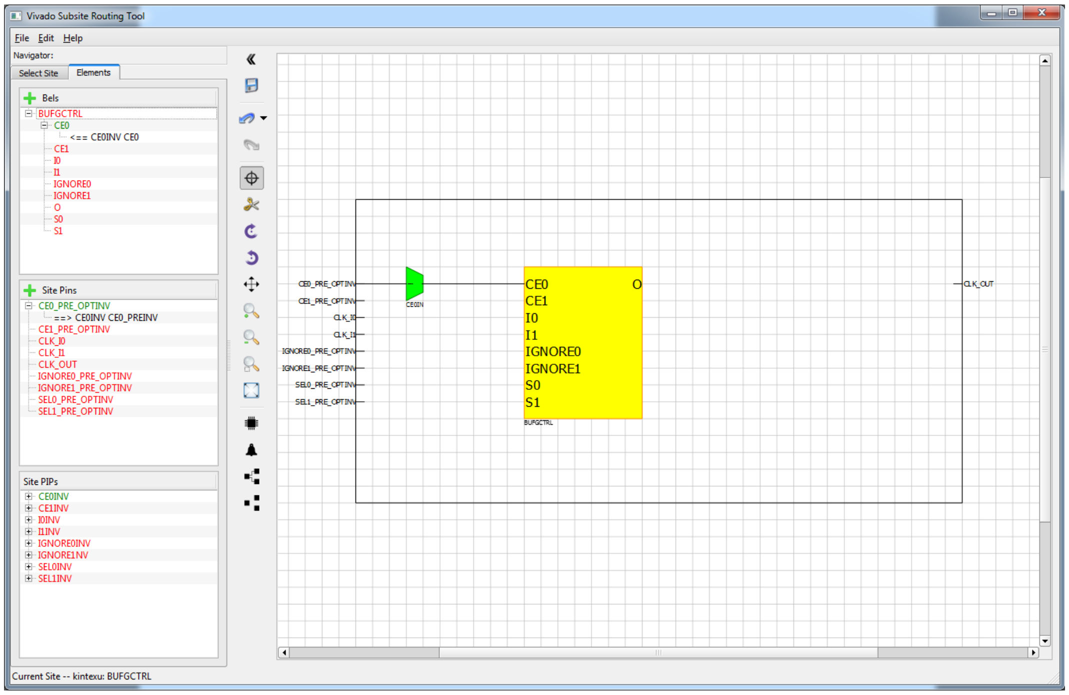


Figure B.7: Wire Connections in VSRT

Now that you know the basics of creating connections in VSRT, let’s go back to the BUFGCTRL example. You will first need to open an instance of a BUFGCTRL site in Vivado’s device browser using the Tcl commands shown in Listing B.1 (replace RAMB18_X1Y70 with the appropriate value found in the *log.txt* file for BUFGCTRL sites). After running the commands, the Vivado device browser should now display a BUFGCTRL site as shown in Figure B.8.

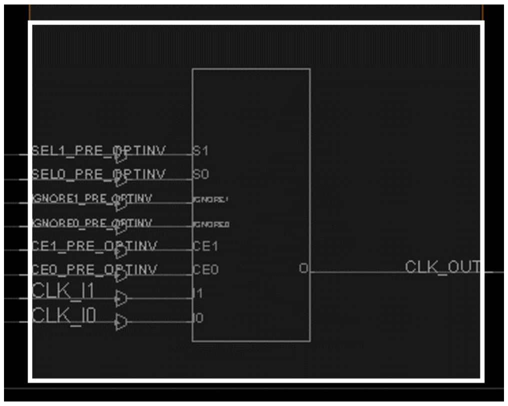


Figure B.8: BUFGCTRL Site in Vivado

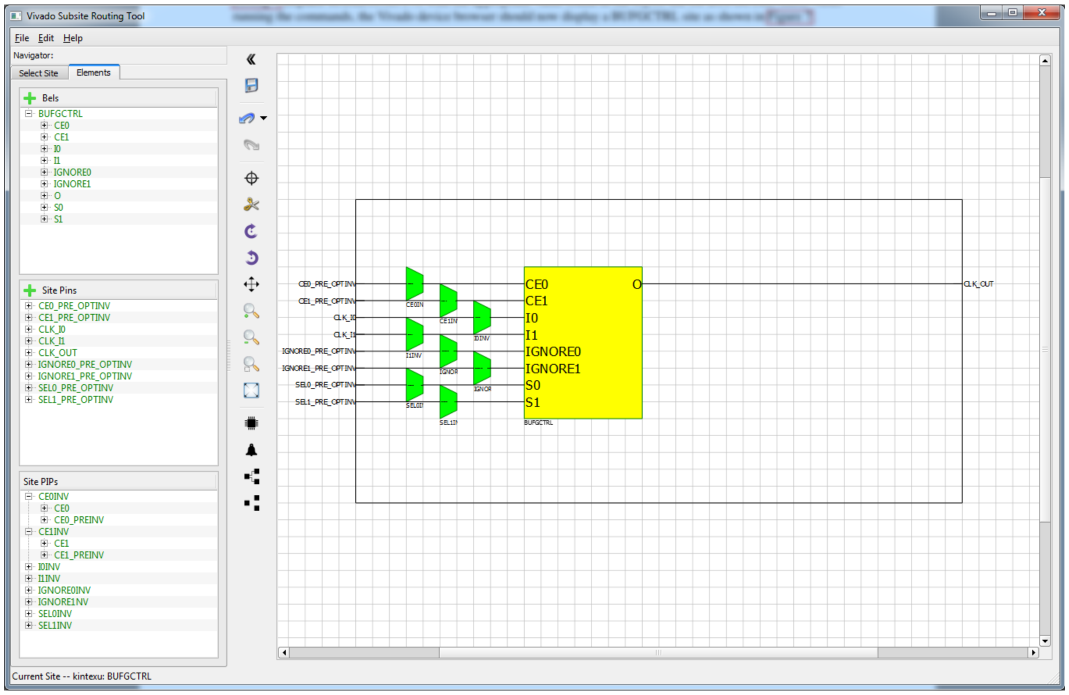


Figure B.9: Completed BUFGCTRL in VSRT

The site in the device browser is interactive, so you can click-on or hover-over elements to view their names. The next step is to recreate the connections you see in Figure B.8 in the VSRT

GUI. As described above, you can do this by adding elements to the drawing pane, rearranging the elements, and drawing wires between element pins. Once you have generated all connections in VSRT for the BUFGCTRL site, it should look similar to Figure B.9. Notice that all items in the "Elements" tab are green, meaning that every pin has at least one connection. This is a good indicator that we are done wiring up the BUFGCTRL site.

To generate the completed primitive def file, click File→Save, and the prompt shown in Figure B.10 will appear.

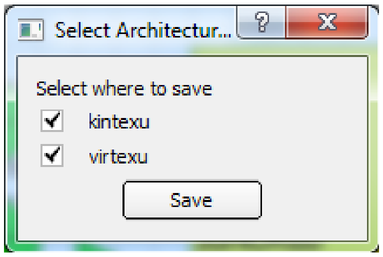


Figure B.10: VSRT Save Prompt

The prompt will display all available architectures to the tool (in this example, there is only two). Since the BUFGCTRL site for `kintexu` and `virtexu` architectures are identical, we can save the completed primitive def to both of those architectures. Click "Save" and navigate back to the partial primitive defs directory that was created from Tincr. You will see a new folder has been created called "CompletePrimitiveDefinitions" as shown below.

```
[ttown523@CB461-EE09968:primitiveDefs] ls
artix7   CompletePrimitiveDefinitions   kintex7  kintexu     kintexuplus
log.txt  virtex7                        virtexu  virtexuplus  zynq
```

All completed primitive definitions will appear in this folder (separated by architecture specific sub-folders). After saving the primitive def, you will be returned to the VSRT start screen. The final step in completing the primitive site is to right click on the BUFGCTRL site you just completed, and tag it as complete (shown in Figure B.11.

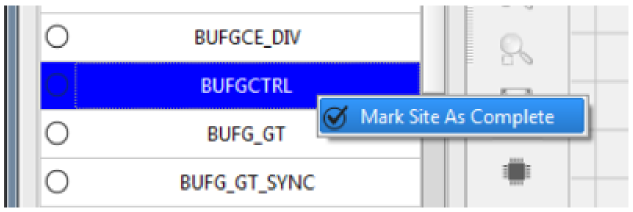


Figure B.11: How to mark a site as complete in VSRT

B.4.2 Single-Bel Mode

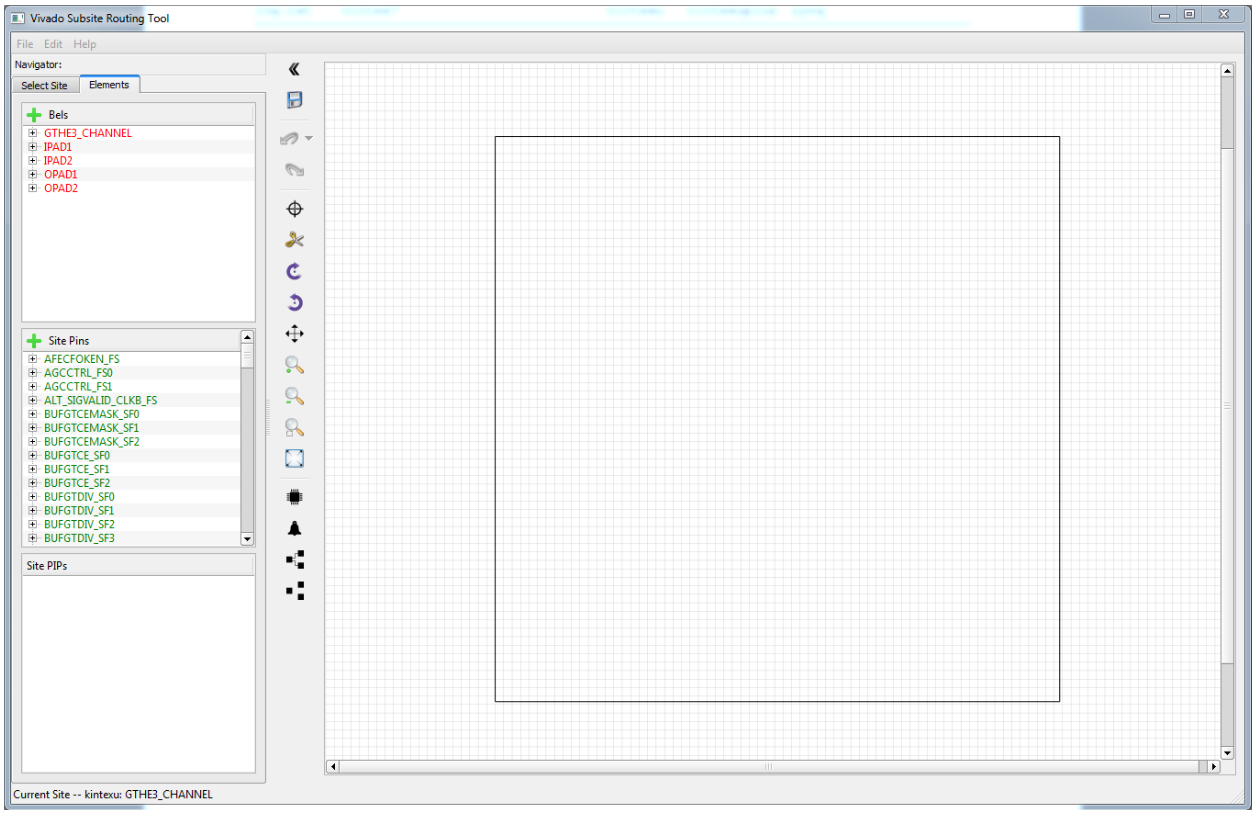
For primitive sites with a single BEL (or one large BEL that takes up most of the site), many of the connections can be automatically inferred in Tcl. However, not all connections can be generated and some may be generated incorrectly. VSRT offers a "Single-Bel Mode" for these sites so that instead of creating every single connection by hand (which could be thousands of connections for large sites), only the connections that Tincr could not infer need to be generated. This is a *significant* time-saver for large primitive sites.

The biggest difference between single-BEL mode and regular mode, is that BEL pins can be individually added to the drawing pane instead of an entire BEL (BELs can still be added if desired). Users can add the BEL pins that are unconnected, and generate connections for those pins only. To show an example of single-BEL mode, we will generate connections for a GTHE3_CHANNEL site. Double-click the GTHE3_CHANNEL site in the "Select Site" tab, and click "Yes" in the popup prompt that appears (Figure B.4). Figure B.12 shows the initial view of the GTHE3_CHANNEL site in VSRT.

There are a few important things to note about sites that have been loaded in single-BEL mode:

1. Site pins are not, by default, added to the drawing pane. Instead, only a generic site boundary is shown. Site pins can be added to the drawing pane by double clicking the site pin in the "Elements" tab. BEL pins can also be added to the drawing pane by double-clicking the corresponding BEL pin.

2. Several site and BEL pins show as green in the "Elements" tab. This indicates that these pins have connections that were automatically generated in Tcl, and usually don't have to be modified in VSRT.



Current Site -- kintexu: GTHE3_CHANNEL

Figure B.12: GTHE3_CHANNEL in VSRT

3. There is more than one BEL in this site, which is normal. The "GTHE3_CHANNEL" BEL is the one where most connections have been automatically generated. The other BELs can be added to the scene like they would in regular mode, where their connections can be generated.

4. If a BEL or pin is added to the drawing pane, any existing connections **will be removed**. This is because the tool will assume that the automatically created connections for those elements were incorrect.

To generate the remaining connections, simply add the unconnected pins to the drawing pane and wire them together (based on the equivalent Vivado GTHE3_CHANNEL image which is not shown in this document). Listing B.2 shows Tcl commands that can isolate a BEL Pin in Vivado.

Listing B.2: Tcl commands to select a BEL pin in Vivado's device browser

```
// First click on the Bel you want in the device browser!
```

```
Vivado% set bel [get_selected]

Vivado% select_objects [get_bel_pins $bel/GTHRXN]
```

Once you have connected all unconnected bel-pins, the drawing pane should look similar to Figure B.13. Of the original 959 bel-pins for the GTHE3_CHANNEL site, only 35 of them (3.65%) need to be connected using VSRT. This demonstrates the productivity advantages of using single-BEL mode. It is *strongly encouraged* for VSRT users to run the tool in single-BEL mode for large sites where applicable.

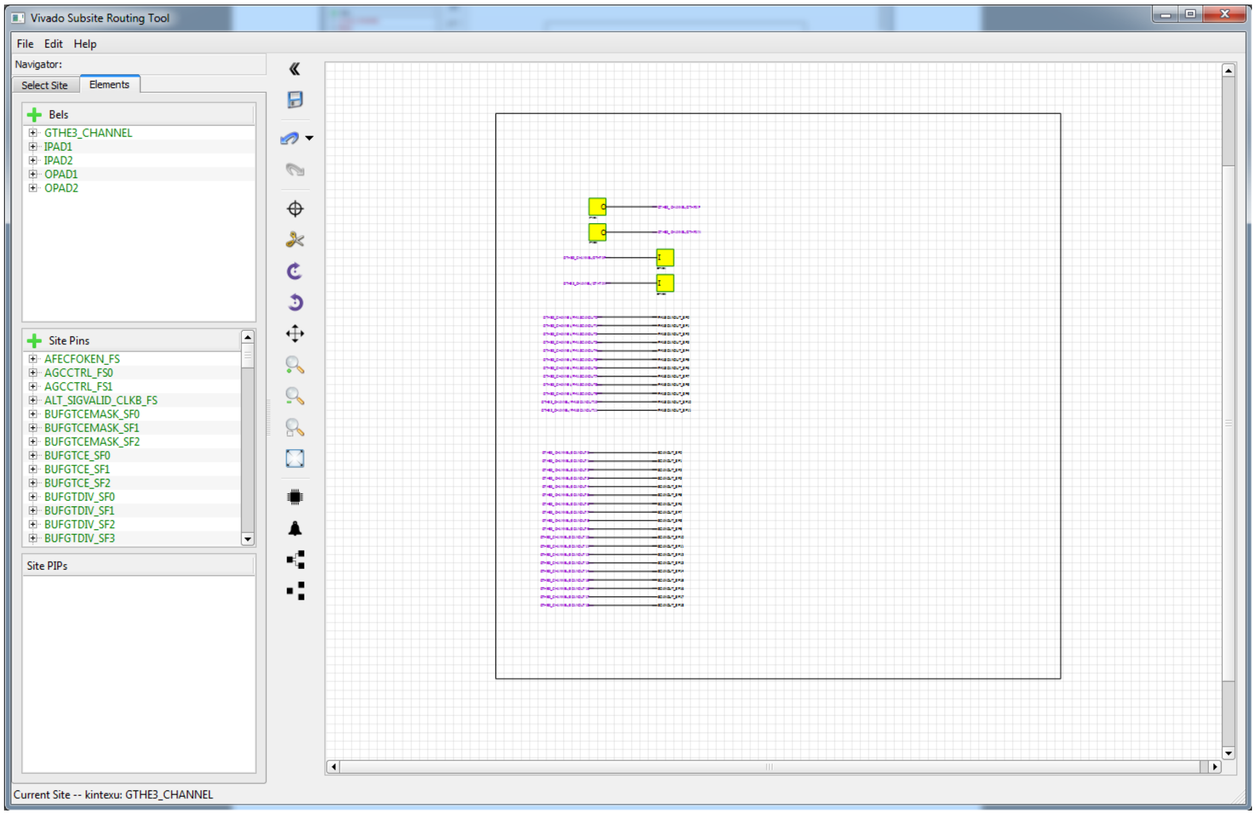


Figure B.13: Complete GTHE3_CHANNEL in VSRT

We are not quite done with the GTHE3_CHANNEL site however. As you scroll through the site pins in the “Elements” tab, you will see several pins that are still unconnected. This may seem like a problem, but it is not. If you view the offending pins in Vivado’s device browser, you will see that they do not connect to any elements inside of the site. Therefore, the final step to complete the GTHE3_CHANNEL site in VSRT is to mark the remaining site pins as unconnected.

To do this, right click the unconnected site pins and select “Mark As Unconnected” (Figure B.14). The pin colors will change to gray indicating that they no longer need to be connected. After all unconnected pins are marked as unconnected, the complete primitive definition can be generated using the steps introduced in subsection B.4.1.

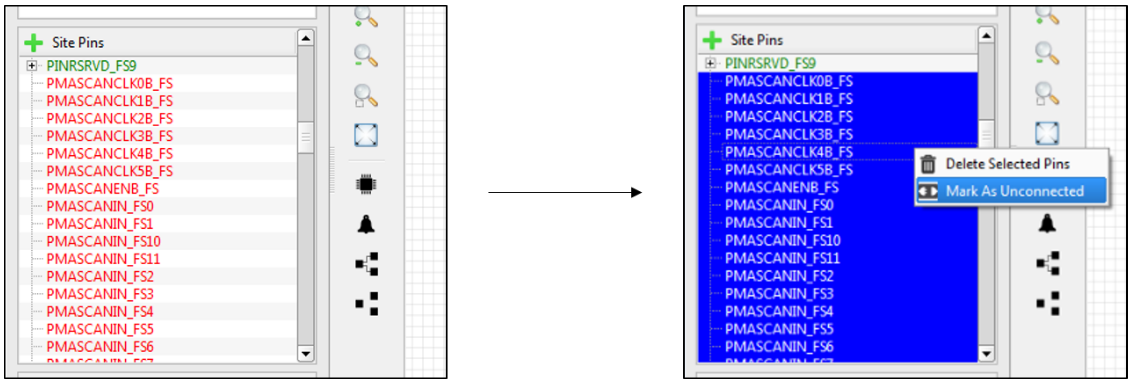


Figure B.14: How to mark site pins as unconnected in VSRT

B.5 Features

B.5.1 Wire Hiding

The current version of VSRT does not support drawing wires along the grid system. Wires always appear as straight lines between two connected pins. For complex primitive sites (such as SLICEL or DSP48E2), this can quickly clutter up the drawing pane and make it difficult to see what pins are connected to where. To alleviate this, VSRT gives the users the option to hide wires. The toolbar has two buttons for wire hiding:

- Hide *all* wires in the current drawing pane
- Show *all* wires in the current drawing pane

You can also hide and show wires for individual BELs and Site PIPs. To do this, right-click on the element in the drawing pane and select "Hide/Show Wires".

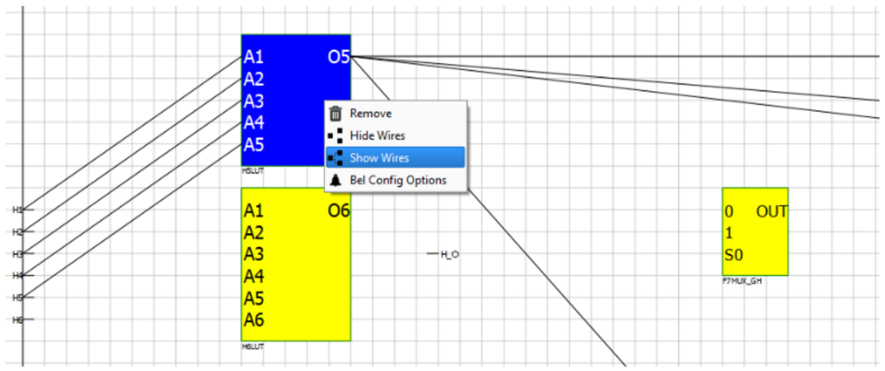


Figure B.15: Wiring Hiding Options

The suggested flow for VSRT users is to draw the connections for one BEL at time, while hiding all other wires. Even if an element's wires are hidden, the border color around the shape in the drawing pane indicates how many pins of the element have been connected (see next section).

B.5.2 Shape Border Colors

Each element in the drawing pane has a border color based on how many of its pins are connected. Figure B.16 shows the possible colors. Green means all pins have been connected, orange means some (but not all) pins have been connected, and red means no pins have been connected. This color coding makes it easier to identify which elements in the drawing pane have unconnected pins when wire hiding is enabled.

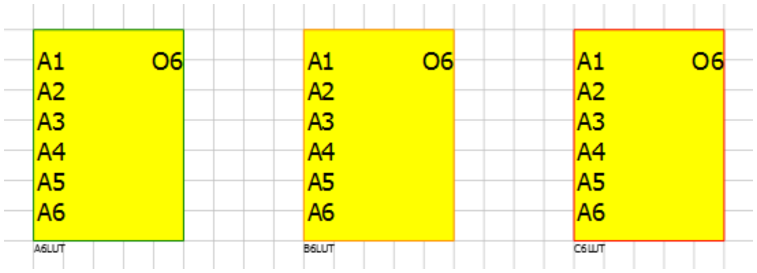
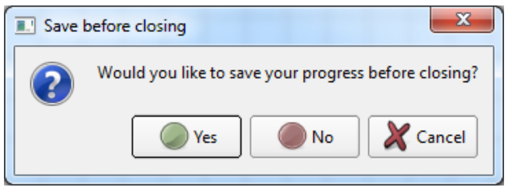


Figure B.16: Shape Border Colors

B.5.3 Saving

VSRT supports saving and restoring the progress of a primitive site. To save your progress, click File→Close Site. The prompt shown in Figure B.17 will appear. Click “Yes” and your site progress will be saved to a XML file. To restore the site progress and continue working, simply open the site again. A site is automatically saved when a complete primitive definition is created. **NOTE:** If you save a site open in single-BEL mode, the connections will be saved but the drawing pane will not.



Save before closing

Would you like to save your progress before closing?

Figure B.17: Save Progress Prompt

B.5.4 INOUT Pins

BEL pins in Xilinx FPGAs have three possible directions: INPUT, OUTPUT, and INOUT. INPUT pins are signal sinks, OUTPUT pins are signal sources, and INOUT pins are bidirectional, meaning the pin can act as either a source or sink. Because of this behavior INOUT pins can connect to pins of any direction. Table B.1 shows the connections that should be created when an INOUT pin is connected to another pin.

Table B.1: INOUT Pin Connections

Pin 1 Type	**Pin 2 Type**	**Connection(s) Generated**
INOUT	INPUT	Pin1 $\rightarrow$ Pin2
INOUT	OUTPUT	Pin1 $\leftarrow$ Pin2
INOUT	INOUT	Pin1 $\leftarrow$ Pin2 Pin1 $\rightarrow$ Pin2

VSRT automatically generates the correct connections for INOUT pins according to the table. Also, VSRT displays the color of INOUT pins as gray instead of black so they can be easily identified in the drawing pane.

B.5.5 Undo/Redo Functionality

Most actions in VSRT can be undone. So, if you make a mistake don't worry! Click "Undo" and continue working. Two buttons in the tool bar allow actions to be undone or redone (shortcut keys are shown in parenthesis):

Undo the last action (Ctrl+Z)

Redo the last action (Ctrl+Y)

The "Undo" button on the tool bar also has a dropdown menu to undo multiple actions at once.

B.5.6 Deleting Elements

You can remove items from the drawing pane by entering "delete mode." To toggle delete mode, click the toolbar button shown below or click the hotkey shortcut "d."

Enter delete mode (d)

The cursor will now have a small red “x” in the bottom right-hand corner to signify that you can remove elements. To remove a set of elements, click on the drawing pane and draw a bounding box around the elements you want to delete. All elements within the bounding box will be deleted (including wires).

B.5.7 Adding VCC/GND BELs

In some device architectures, the Tcl API into Vivado *cannot* access VCC or GND BELs (i.e. the Tcl function *get_bels* will omit VCC and GND). An example GND BEL that exhibits this behavior is shown in Figure B.18. Because of this, some partial primitive defs generated from Vivado are missing required BEL elements to generate all connections. VSRT allows users to add these missing VCC and GND BELs to a site by clicking the add icon next to the “Bels” header. Figure B.19 demonstrates the complete process.

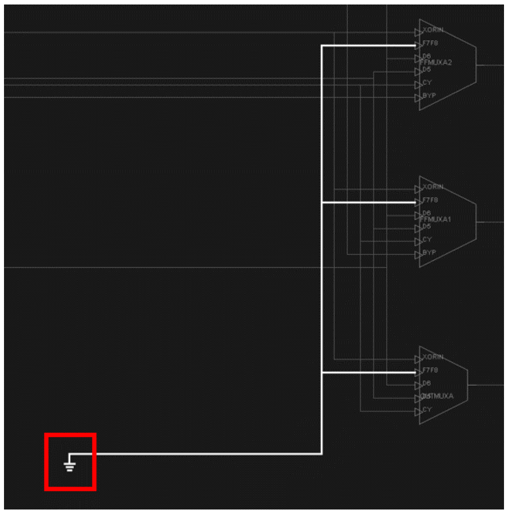
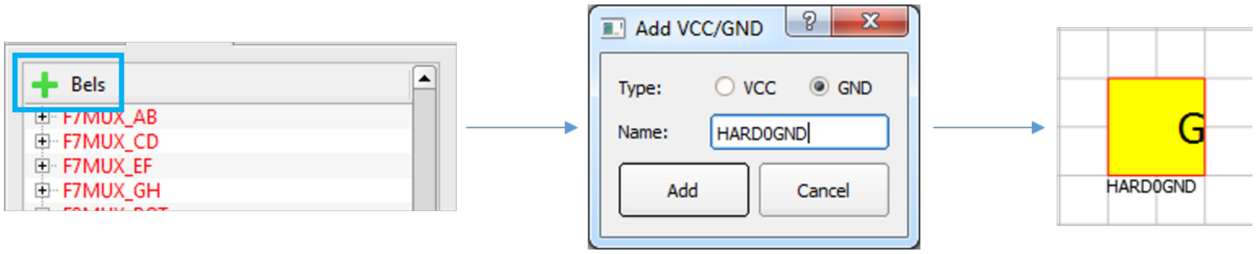


Figure B.18: Vivado GND BEL (shown in the red box)



Type: ( ) VCC [✓ GND]
Name: HARD0GND

Figure B.19: How to add new VCC/GND BELs to a site

B.5.8 Manipulating Site Pins

As you are using VSRT to create complete primitive definitions, you may notice three issues with site pins:

1. Site Pins that appear in Vivado, but not VSRT
2. Site Pins that appear in VSRT, but not Vivado.
3. Unconnected Site Pins

VSRT provides solutions to handle each of these issues. For site pins that appear in Vivado but not VSRT (1), they can be added by clicking the add icon next to the "Site Pins" header. Figure B.20 demonstrates the complete flow. As the figure shows, after you click the add icon a popup window appears. Enter the pin name, number of pins to create, and the pin direction in the window and click "Add." The pins will then be created and added to the scene in the top-left corner of the drawing pane. The "Count" field is useful for creating many site pins with the same name. For example, if the pin name entered was "Test" and the count value entered was "5", the pins Test0, Test1, Test2, Test3, and Test4 will be created. **NOTE:** To create a single site pin with no number on the end of the name, simply set the count field to 1.

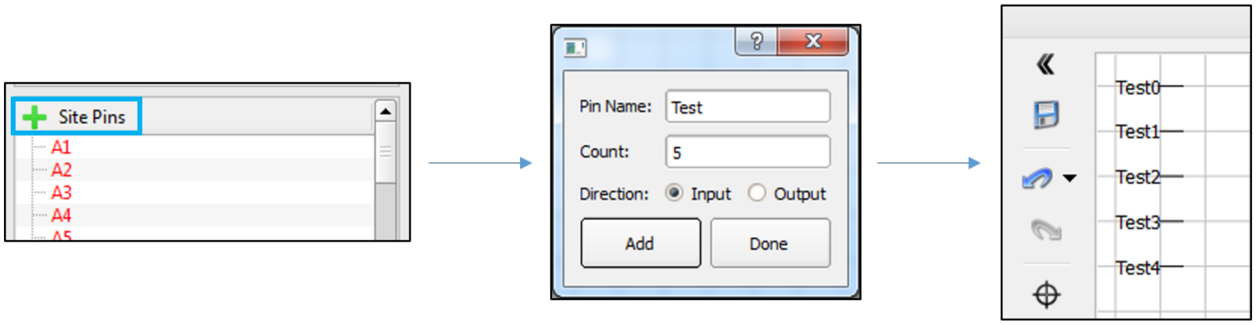


Figure B.20: How to add new site pins to a site

For site pins that appear in VSRT but not Vivado (2), you can simply remove them. To do this, right-click the site pins you want to remove in the "Site Pins" section of the "Elements" tab and select "Delete Selected Pins." For site pins that are unconnected (3), you can mark them as unconnected by using the same steps but selecting "Mark As Unconnected" instead. Figure B.21 shows what these options look like in the GUI. **Do NOT delete unconnected site pins. They still have external connections in the tile, so need to be included in the final primitive def file**.

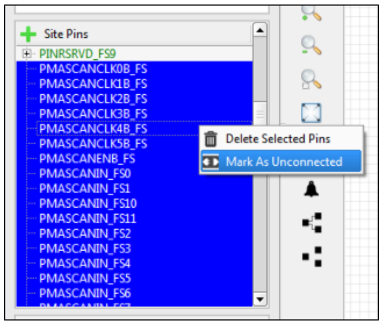
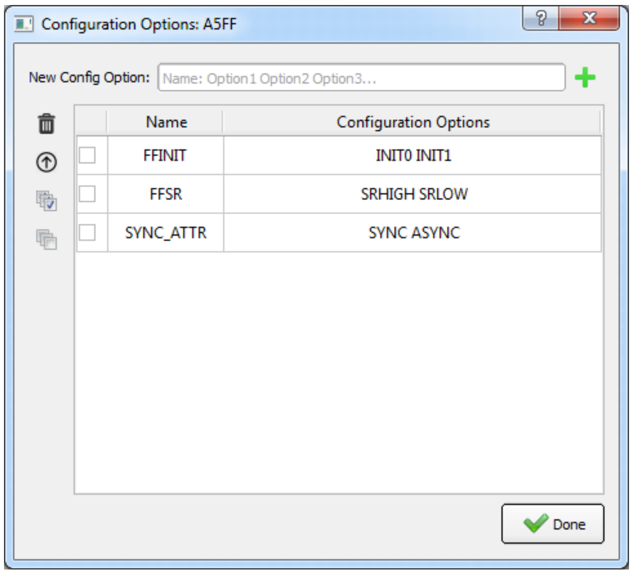


Figure B.21: How to mark site pins as unconnected or remove them in VSRT

B.5.9 Configuration Options

If you generated the partial primitive defs from Vivado with the option "includeConfigs" enabled, then the primitive defs will include configurable properties for each BEL. To view a BELs configurable properties in VSRT, right-click the BEL in the drawing pane and select "Bel Config Options." Alternatively, you can select the BEL in the drawing pane and click [🔔] on the tool bar. Figure B.22 shows the popup window that is displayed when you view a BEL's configuration options for a SLICEL A5FF. As the figure shows, the window allows you to delete configurations



Name	Configuration Options
FFINIT	INIT0 INIT1
FFSR	SRHIGH SRLOW
SYNC_ATTR	SYNC ASYNC

Figure B.22: BEL Configuration Popup Window

[ ], add new configurations [ ], and promote configurations to the site-level [ ]. Adding and deleting configurations is fairly straightforward, but what does it mean to promote configurations to the site-level? The "FFSR" property shown in Figure B.22 is a good example of why promoting properties is required. FFSR determines if the reset on the corresponding Flip-Flop should be high or low-asserted. However, every Flip-Flop in a SLICEL **must have the same FFSR property**. This means that FFSR is really a property on the site, and not on the individual BEL. VSRT allows you to "promote" these types of properties to the site-level. Specifically, the promoted property will be removed from all BELs and added to the site's configuration properties instead. A site's configuration properties can be viewed by pressing [ ] on the tool bar.

B.5.10 Moving Shapes

All shapes in the drawing pane can be moved to another location (even site pins) by dragging and dropping them. Multiple shapes can be moved at a time by selecting multiple items at once. Also, the site boundary rectangle can be resized if the initial site estimate was too large or small.

B.5.11 Tool Bar Buttons and Shortcuts

This section contains a description of the options on the VSRT tool bar, and keyboard shortcuts if they exist.

Hide the navigator pane

Show the navigator pane

Write the completed primitive definition (Ctrl + S)

Undo the last action (Ctrl + Z)

Redo the last action (Ctrl + Y)

Toggle wire drawing mode (w)

Toggle delete mode (d)

Rotate selected elements clockwise (Ctrl + R)

Rotate selected elements counterclockwise (Ctrl + Shift + R)

Drag View (Ctrl + D)

Zoom in (Ctrl + mouse forward scroll)

Zoom out (Ctrl + mouse backward scroll)

Zoom to selection

Zoom Best Fit: Set the zoom level on the drawing pane to show the entire primitive site.

Show site configuration properties

Show BEL configuration properties for selected BEL.

Show all wires on the drawing pane

Hide all wires on the drawing pane

B.6 Known Issues

This section contains a list of known issues with both Vivado 2016.2 and VSRT when trying to generate complete primitive definitions. If you find any other bugs with VSRT, please create a new issue at `https://github.com/byuccl/RapidSmith2` to report it.

- When moving multiple site pins at once, occasionally not all site pins will snap to the grid (see Figure B.23). To fix this, simply select all site pins that are not on the grid, and drag-and-drop them in place. This should snap the pins to the grid and update their locations.

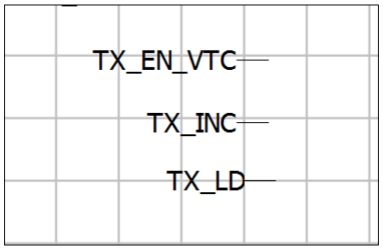


Figure B.23: Site Pin Moving Bug

- Sometimes the VSRT drawing pane gets stuck in zooming mode so that the mouse scroll wheel will zoom in and out when “Ctrl” is not pressed. To fix this press “Ctrl”, zoom in and out slightly (by scrolling the mouse wheel), and release “Ctrl.” The scroll wheel should work normally after.

- **Do not select a group of items while scrolling the drawing pane**. The items may end up in the top-left corner of the drawing pane (or some other random location) where they can be hard to find. Instead of scrolling while selecting items, zoom out enough so you can see all the items you want to move, and then select all items at once.

- When you resize the left side of a primitive site on the drawing pane, all input site pins will move as well. Similarly, when you resize the right side of a primitive site, all output site pins will move. This can be useful if you haven’t moved any site pins yet, but otherwise it can disturb the current state of the drawing pane. **Best practice is to resize the site ONCE before you start working on a primitive site, and then do not resize it again**.

- There are some site pins in Vivado whose names do not show up in the device browser (see Figure B.24). To determine the names of these pins, click on the **Tile Wire** connected to the pin (highlighted as white in the figure) and run the Tcl command `[get_site_pins -of [get_selected]]`:

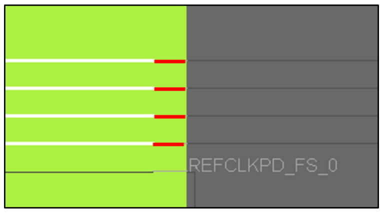


Figure B.24: Missing Site Pin Names in Vivado (shown in red)

- In the Vivado device browser, there appears to be some site wires that can have multiple drivers but are not connected to any bidirectional BEL pins. Figure B.25 shows an example for an UltraScale BITSLICE_COMPONENT_RX_TX site. In VSRT, the two output pins highlighted with red boxes should not be connected. They should only be connected to the sink BEL pins highlighted with green boxes. As a general rule, **do not connect two output**

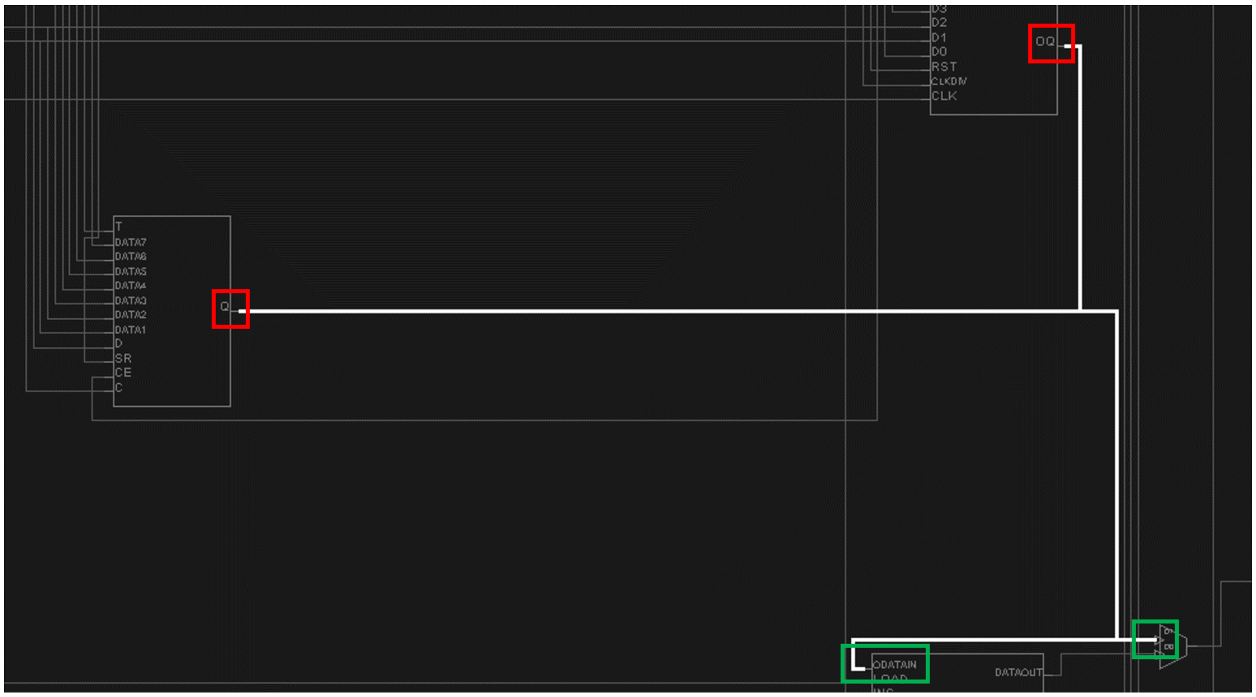


Figure B.25: Example of multiple drivers on a site wire. Output BEL pins are highlighted with red boxes and input BEL pins are highlighted with green boxes.

pins together in VSRT even if that is how they appear in the Vivado device browser. Only connect output pins to input pins (or inout pins) and vice versa.

APPENDIX C. SIMULATED ANNEALING PLACEMENT ALGORITHM

To test the capabilities of VDI, a site-level placer targeting Artix7 devices was implemented using RapidSmith2 data structures. The basic simulated annealing algorithm used to implement the placer is described in chapter 14 of [22]. Specifically, the algorithm operates on groups of cells that have been clustered together into sites. Site clusters are swapped until a minimum cost placement is found on the device. This appendix describes the placement algorithm in detail including (a) the annealing schedule of the algorithm, (b) the cost function used to determine an optimal placement, and (c) the custom data structures used for placement.

C.1 Annealing Schedule

There are four important aspects to consider when designing the annealing schedule of a custom simulated annealing algorithm. Each of these, and the corresponding solution in the RapidSmith2 placer, is discussed below.

1. **What should the initial temperature be?** The initial temperature in the algorithm is set to ten times the average cost of a move. 10,000 temporary moves are used to calculate this average cost, and are then rejected to preserve the initial state of the placer. [22] suggests that a good initial temperature is twenty times the average cost, but that seemed to create initial temperatures that were far too high to be usable.

2. **How should the temperature be updated over time?** The algorithm uses a very simple temperature updating scheme, and decreases the current temperature by a factor of .99 after each iteration of the annealing loop. An optimization that could be made to the placer is updating the temperature scheme to be a function of the acceptance rate instead.

3. **How many moves per temperature should be evaluated?** The number of moves per temperature is a function of the move acceptance rate at the *previous temperature*. While the

acceptance rate is above 65%, only 10,000 moves are evaluated. This is because at these temperatures, the algorithm is likely just randomizing the initial placement and not much useful work is being done. When the acceptance rate is between 65% and 5%, anywhere between 30,000 and 50,000 moves are evaluated. This is the critical region of the annealing schedule and where most decreases in cost are achieved. When the acceptance rate drops below 5%, only 20,000 moves per second are evaluated. At these temperatures incremental improvements are being made, but not many.

4. **When should the algorithm stop?** The algorithm stops when less than .1% of moves are being accepted. This number was arbitrarily chosen based on experimentation.

C.2 Cost Function

The cost of a given placement configuration is determined by summing the cost of each individual net in the design. For each net, the cost is set to the half-perimeter of the bounding box of the net multiplied by the fanout factor (λ) of the net. The corresponding equation is

$$C(n) = \lambda * [(top - bottom) + (right - left)].$$

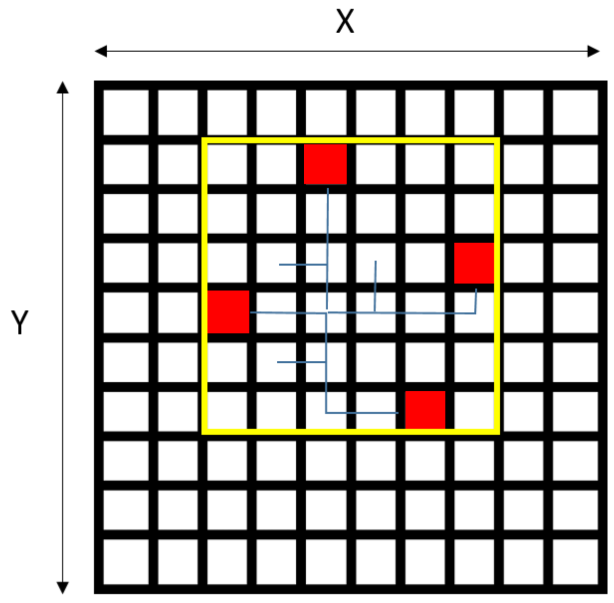


Figure C.1: Example Bounding Box

Top is the largest Y coordinate of any site connected to the net and *bottom* is the smallest. `Right` and `left` are the same except for using the X coordinate. An example of a net's bounding box is shown in Figure C.1. Once the half-perimeter of the bounding box is determined it is multiplied by the *number of sink sites* that the net is connected to. This gives more weight to large nets in the design to create a better final placement. When a given site cluster is moved to another location, only the nets connected to the site update their cost.

C.3 Data Structures

Two data structures are created for placing designs: `SiteClusters` and `VirtualNets`. `SiteClusters` group cells that must be moved together during the annealing process. Generally, this corresponds to cells within the same site, but there are a few noticeable exceptions. Carry cells connected through the carry chain, for example, must be placed vertically to one another to use the dedicated carry routing. The same applies to DSP carry chains and BRAM cascade paths. To account for these situations, the class `SiteCluster` is a top-level class with the inheritance tree shown in Figure C.2.

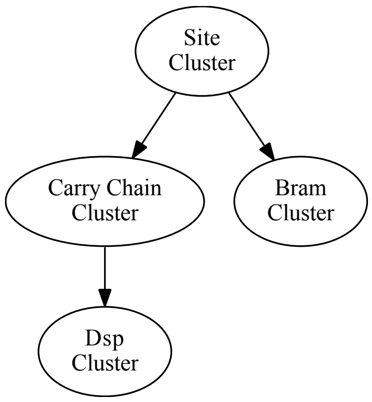


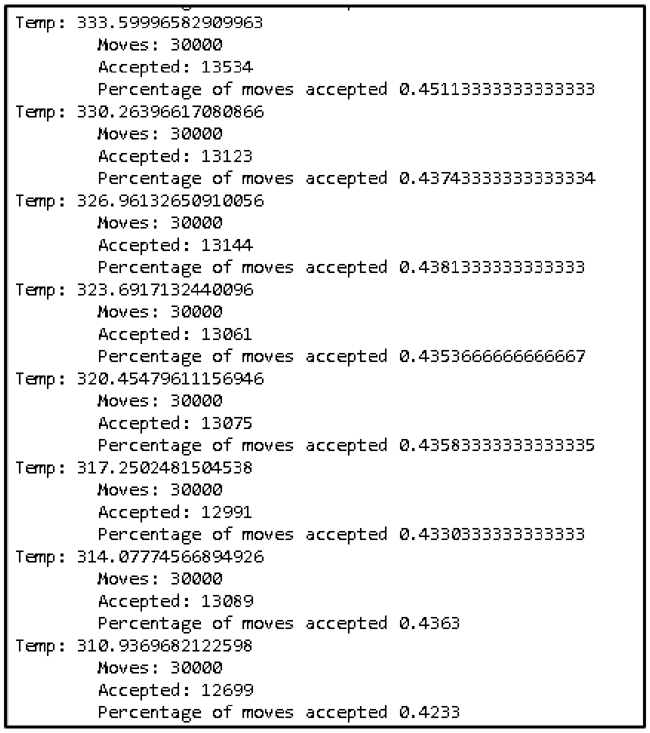
Figure C.2: Site Cluster Inheritance Tree.

The sub-classes implement custom move and swap functions in order to move all cells within the cluster accordingly. The placement algorithm chooses to not swap with carry-chain connected sites. The cost of swapping with a carry-chain is too expensive, and could cause a chain-reaction of site swaps which could degrade the performance of the placer.

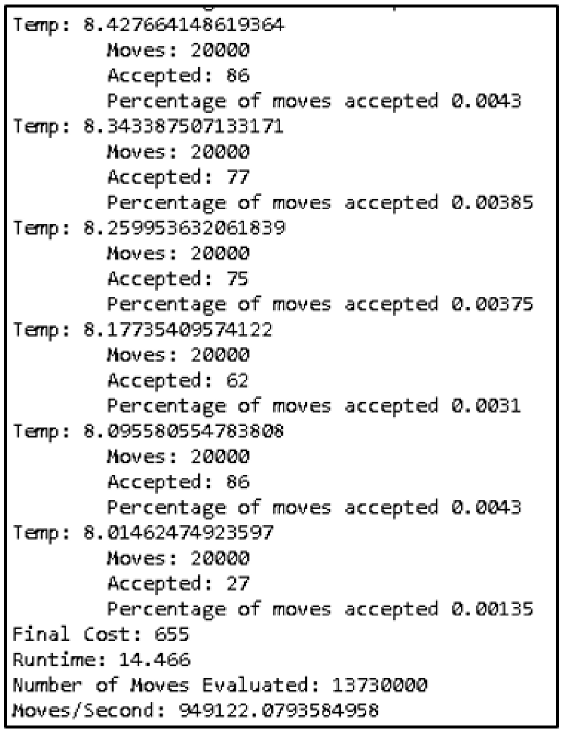
A `VirtualNet` is a net object that connects site clusters together. Instead of connecting to cell pins like they would in a regular netlist, these nets connect to sites on a device. Each net in a design to be placed is assigned a virtual net, that can be used to compute its current cost based on the cost function given in the previous section. `SiteClusters` and `VirtualNets` make it easy to modify and update the placer as new special cases are found.

C.4 Sample Output

To help debugging, the placer prints several useful statistics while running the annealing algorithm. Figure C.3 shows the sample output that is printed after each temperature change, and once the algorithm has completed.



```
Temp: 333.59996582909963
        Moves: 30000
        Accepted: 13534
        Percentage of moves accepted 0.45113333333333333
Temp: 330.26396617080866
        Moves: 30000
        Accepted: 13123
        Percentage of moves accepted 0.43743333333333334
Temp: 326.96132650910056
        Moves: 30000
        Accepted: 13144
        Percentage of moves accepted 0.4381333333333333
Temp: 323.6917132440096
        Moves: 30000
        Accepted: 13061
        Percentage of moves accepted 0.4353666666666667
Temp: 320.45479611156946
        Moves: 30000
        Accepted: 13075
        Percentage of moves accepted 0.43583333333333335
Temp: 317.2502481504538
        Moves: 30000
        Accepted: 12991
        Percentage of moves accepted 0.4330333333333333
Temp: 314.07774566894926
        Moves: 30000
        Accepted: 13089
        Percentage of moves accepted 0.4363
Temp: 310.9369682122598
        Moves: 30000
        Accepted: 12699
        Percentage of moves accepted 0.4233
```



```
Temp: 8.427664148619364
        Moves: 20000
        Accepted: 86
        Percentage of moves accepted 0.0043
Temp: 8.343387507133171
        Moves: 20000
        Accepted: 77
        Percentage of moves accepted 0.00385
Temp: 8.259953632061839
        Moves: 20000
        Accepted: 75
        Percentage of moves accepted 0.00375
Temp: 8.17735409574122
        Moves: 20000
        Accepted: 62
        Percentage of moves accepted 0.0031
Temp: 8.095580554783808
        Moves: 20000
        Accepted: 86
        Percentage of moves accepted 0.0043
Temp: 8.01462474923597
        Moves: 20000
        Accepted: 27
        Percentage of moves accepted 0.00135
Final Cost: 655
Runtime: 14.466
Number of Moves Evaluated: 13730000
Moves/Second: 949122.0793584958
```

Figure C.3: Sample Placement Output