RAPIDSMITH2

A Library for Low-level Manipulation of Vivado Designs at the Cell/BEL Level

Technical Report and Documentation

Brent Nelson, Thomas Townsend, and Travis Haroldsen

NSF Center for High Performance Reconfigurable Computing (CHREC) *
Department of Electrical and Computer Engineering
Brigham Young University
Provo, UT, 84602

Last Modified: August 15, 2017

*This work was supported in part by the I/UCRC program of the National Science Foundation, grant numbers 0801876 and 1265957.

Contents

1 Introduction

1.1 What is RapidSmith2?

The original BYU RapidSmith project began in 2010. Its goal was to develop a set of tools and APIs which would provide academics with an easy-to-use platform to implement experimental CAD ideas and algorithms on modern Xilinx FPGAs. It integrated with Xilinx's old design suite, ISE. RapidSmith2 represents a major addition to RapidSmith. Specifically, Vivado designs are now supported. Using RapidSmith2 you can write custom CAD tools which will:

- Export designs from Vivado
- Perform analyses on those designs
- Make modifications to those designs
- Import those designs back into Vivado for further processing or bitstream generation

Futhermore, you need not start with a Vivado design — you can create a new design from scratch in RapidSmith2 and then import it into Vivado if desired.

The other new major capability of RapidSmith2 is that it changes RapidSmith's design representation. Instead of using XDL's view of a design with Instances and Sites, RapidSmith2 uses Vivado's representation of design with Cells and BELs. This is a significant change as it exposes the actual design and device in a way that RapidSmith never did, opening up new CAD research opportunities which were difficult to perform using Rapidsmith.

1.2 Who Should Use RapidSmith2?

RapidSmith2 is aimed at anyone desiring to do FPGA CAD research on real Xilinx devices available in Vivado. As such, users of RapidSmith2 should have some understanding of Xilinx FPGA architecture, the Vivado design suite, and the Tcl programming language. However, one goal of this documentation is to provide sufficient background and detail to help bring developers up to speed on the needed topics. RapidSmith2 is by no means a Xilinx Vivado replacement. It cannot be used without a valid and current license to a Vivado installation (RapidSmith2 cannot generate bitstreams for example).

1.3 Why RapidSmith2?

The Xilinx-provided Tcl interface into Vivado is a great addition to the tool suite. It can be used to do a variety of useful things including scripting design flows, querying device and design data structures, and modifying placed and routed designs. In theory, the Tcl interface provides all of the functionality needed in order to create any type of CAD tool as a plugin to the normal Vivado tool flow. However, there are a few issues in TCL that motivate the use of external CAD tool frameworks such as RapidSmith2. These include:

- Tcl, being an interpreted language, is slow. It is far too slow to implement complex algorithms such as PathFinder. Compiled and managed runtime languages are a better option in terms of performance.
- Tcl is hard to program in. TCL is not an object oriented language, and so writing complex algorithms are difficult since Object-Oriented language constructs do not exist. That being said, TCL is great for writing automation scripts.
- There are some memory issues in Vivado's Tcl interface. In our experience, long-running scripts eventually cause the system to run out of memory even if they are not doing anything interesting.
- Vivado's TCL interface does not offer a complete device representation (determined by Brad White's MS work). Most notably, a user cannot gain access to sub-site wire objects through the Tcl interface. This limits the CAD tools that can be created in Tcl, but this additional information can be added to external tools with some manual work.

In short, the ability to export designs out of Vivado, manipulate them with more powerful languages such as Java, and then import the design back into Vivado is a very useful capability.

RapidSmith2 (in conjunction with Tincr which is described in **??**) abstracts this process into a few easy-to-use function calls. Generating FPGA part information, importing and exporting **all aspects** of a design, and dealing with other fairly arcane details is made mostly transparent to the user. RapidSmith2 and Tincr provide a nice API into equivalent Vivado device and design data structures. All of this enables researchers to have more time to focus on what matters most: the research of new ideas and algorithms.

1.4 How is RapidSmith2 Different than VPR and VTR?

VPR (Versatile Place and Route) has been an FPGA research tool for several years and has led to many publications on new FPGA CAD research. It has been a significant contribution to the FPGA research community and has grown to be a complete FPGA CAD flow for research-based FPGAs. The main difference between RapidSmith/RapidSmith2 and VPR is that the RapidSmith tools can target commercial Xilinx FPGAs, providing the ability to exit and re-enter the standard Xilinx flow at any point. All features of commercial FPGAs which are accessible via XDL and Vivado's Tcl interface are available in RapidSmith and RapidSmith2. VPR is currently limited to FPGA features which can be described using VPR's architectural description facilities.

1.5 Why Java?

RapidSmith2 is written in Java. We have found Java to be an excellent rapid prototyping platform for FPGA CAD tools. Java libraries are rich with useful data structures, and garbage collection eliminates the need to clean up objects in memory. This helps reduce the time spent debugging, leaving more time for researchers to focus on the real research at hand. Our experience over the past decade is that for student research projects, Java has greatly improved student productivity and led to far more stable CAD tools.

2 Background

RapidSmith2 is based on the original RapidSmith project written by Christopher Lavin as a part of his PhD work at BYU. It was based on the Xilinx Design Language (XDL) which provides a human-readable file format equivalent to the Xilinx proprietary Netlist Circuit Description (NCD) of **ISE**. With RapidSmith, researchers were able to import XDL, manipulate, place, route and export designs among a variety of design transformations. The RapidSmith project made an excellent test bed to try out new ideas and algorithms for FPGA CAD research because code could quickly be written to take advantage of the APIs available. RapidSmith also contained packages which could parse/export bitstreams (at the packet level) and represent the frames and configuration blocks in the provided data structures. RapidSmith continues to be functional and is still available at http://rapidsmith.sourceforge.net/. There, you will find documentation, installation instructions, the RapidSmith code base, and a collection of demo programs demonstrating example CAD tools.

RapidSmith is a great contribution to the FPGA community, but as previously stated is built upon the XDL interface offered in Xilinx's ISE tools suite. Unfortunately, Vivado (Xilinx's newest tool suite) does not support XDL. This makes CAD tools based on XDL incompatible with next-generation Xilinx devices such as UltraScale and UltraScale+. Instead, Vivado offers a Tcl interface that can potentially be used to extract Xilinx device and design information. The ultimate goal of RapidSmith2 is to update the original RapidSmith to support implementing CAD tools with Vivado designs. The remainder of this section gives an overview of the required components developed to support Vivado CAD tool creation in RapidSmith2.

2.1 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 can be used for this purpose.

2.2 Tincr

`Tincr` 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. `TincrIO` is especially important for RapidSmith2 because it demonstrated the plausibility of using Tcl commands to manipulate Vivado designs outside of Vivado.

2.3 Vivado Design Interface

`Tincr` design extraction was a good start for supporting Vivado designs in external CAD tools, but it was far from complete and didn't represent several important aspects of Vivado designs. The **Vivado Design Interface** (VDI), a complete Vivado device and design extraction tool, was developed by Thomas Townsend based on the intial work by Brad White. 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.

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. It provides an XDL-alternative for designs implemented in Vivado. Devices are represented with a XDLRC file and a set of XML files. Designs are represented with RSCP and TCP checkpoint files. RapidSmith2 uses VDI to interface desigs with the Vivado tools suite.

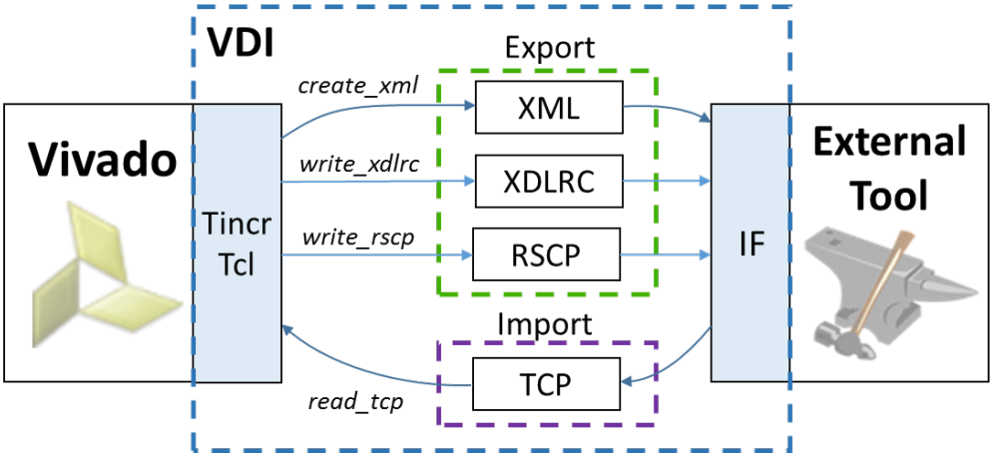


Figure 1: Components of the Vivado Design Interface (VDI)

2.4 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 with sub-site components (i.e. BELs, site PIPs, etc.) such as packers. Travis Haroldsen was interested in exploring packing algorithms for Virtex6 devices, and so created RapidSmith2. 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, the initial version of RapidSmith2 took as input a XDL netlist, and **unpacks** the netlist to its corresponding cells, nets, BELs, and wires. RapidSmith2 has since been updated to support Vivado designs exported through VDI (achieving the original goal of this work). The remainder of this document details important aspects of RapidSmith2 to help researchers quickly develop experimental CAD tools for their research.

2.5 RapidSmith2 Usage Model and Structure

The usage model for RapidSmith2 is shown in Figure 2. As can be seen, a design can be exported from Vivado at multiple different points in the Vivado design flow through VDI. In each case, Tincr is used to export a RapidSmith Checkpoint (RSCP)) which can then be imported into RapidSmith2. At those same points in the design flow, RapidSmith2 can export a Tincr Checkpoint (TCP) which can then be imported back into Vivado. Thus, a complete solution involves Vivado, Tincr, and RapidSmith2.

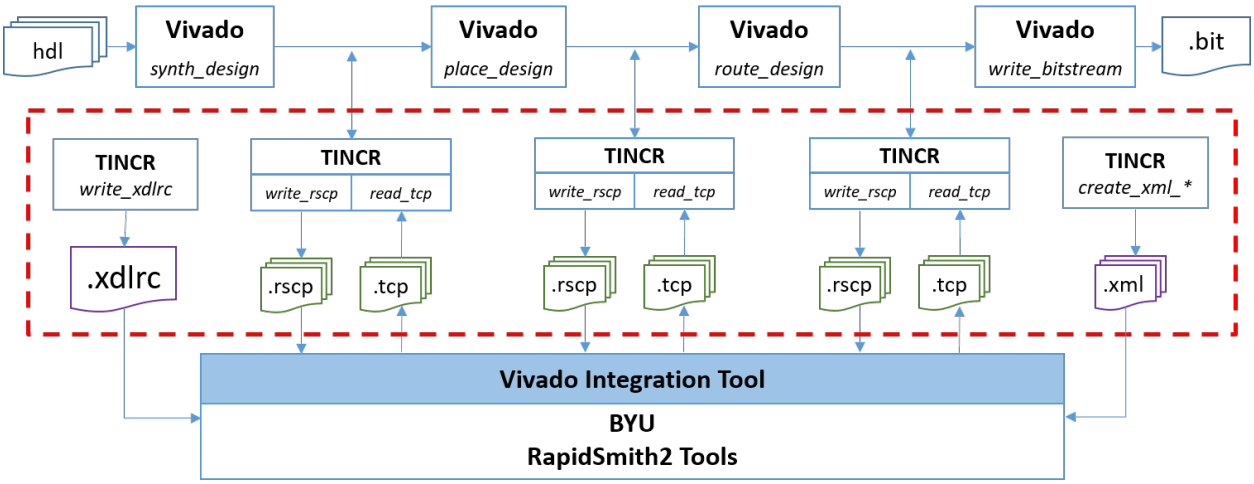


Figure 2: Vivado and RapidSmith2 Usage Model

2.6 Publications and Theses

Several publications and theses have been accepted as a result of this work. They are listed here for those who are interested in learning more about RapidSmith2, `Tincr`, and VDI.

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. 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

3. 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. 6669. 3, 17

4. 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. 16. 17

5. T. Townsend and B. Nelson, Vivado Design Interface: An Export/Import Capability for Vivado FPGA Designs in Field Programmable Logic and Applications (FPL), 2017 27th International Conference.

6. Townsend, Thomas James, "Vivado Design Interface: Enabling CAD-Tool Design for Next Generation Xilinx FPGA Devices" (2017). All Theses and Dissertations. 6492. http://scholarsarchive.byu.edu/etd/6492

3 Getting Started

3.1 Installation

RapidSmith2 is available on Github at: https://github.com/byuccl/RapidSmith2. You can either build RapidSmith2 into .class and .jar files for use in any Java environment, or configure RapidSmith2 to work in an IDE (recommended).

3.1.1 Requirements for Installation and Use

- Windows, Linux or Mac OS X all will work (see additional notes below for Mac OS X)
- Vivado 2016.2. Later versions of Vivado may work, have not been tested yet. Earlier versions will not work.
- JDK 1.8 or later
- Tincr

Tincr is a companion project (https://github.com/byuccl/tincr) which is used for importing/exporting designs between Vivado and RapidSmith2. Installation instructions for Tincr can be found at the repository linked above. For getting started (running the example programs on the provided sample designs) you will not need it installed. Later, as you actually start processing your own Vivado designs you will need to obtain and install it. There are additional dependencies beyond these required for installation, but they are either provided in the distribution itself or are automatically retrieved for you as a part of the installation process. Examples of these additional dependencies include QT Jambi and the BYU Edif Tools.

3.1.2 Steps for Installation

1. Clone the RapidSmith2 repository at https://github.com/byuccl/RapidSmith2. If you are not familiar with GitHub, you will need to install Git on your computer, and run the following command in an open terminal:

   ```
   git clone https://github.com/byuccl/RapidSmith2
   ```

   This will copy the RapidSmith2 repository into a local directory.

2. Create a new environment variable called RAPIDSMITH_PATH, and point it to your local repository of RapidSmith2 that you setup in step (1). This is needed so RapidSmith2 can find required device files and other items at runtime.

3. Build the RapidSmith2 project. RapidSmith2 is managed using a gradle build system. To build the project, navigate to your local repository of RapidSmith2 and execute one of the following scripts in a terminal:

   ```
   ./gradlew build (unix)
   gradlew.bat build (windows)
   ```

   The build process could take a few minutes.

4. At this point, you have two choices: set up RapidSmith2 for use in an IDE, or run RapidSmith2 from the command line. Both choices are detailed below.

   Running from an IDE The gradle scripts in RapidSmith2 currently support setup for both Eclipse and IDEA Java IDEs. This section will detail how to setup the Eclipse environment, but similar steps can be taken for IDEA. If using Eclipse, it is best to use version Eclipse Neon or later. To create a new eclipse project, execute one of the following in a terminal:

```
gradlew antlr eclipse (unix)
gradlew.bat antlr eclipse (windows)
```

Executing these will create an Eclipse *.project* file. After the project file has been created, you can import the project into Eclipse by opening Eclipse and selecting:

```
File->Open Projects From File System
```

and pointing it to your RapidSmith2 local repository. All Java source files will be found under *src/main/java*. **NOTE:** Your RapidSmith2 git repository should not be put inside your eclipse workspace. It is better to put it elsewhere, and then import it into your workspace.

Building on the Command Line After step (3) in the installation process, gradlew produces everything that you will need to run RapidSmith2 from the command line. The following directories are created:

- *build/classes/main*: This folder contains the RapidSmith2 class file directory tree.
- *build/libs*: This folder contains a Jar file of the RapidSmith2 class files.
- *build/distributions*: This folder has both .zip and .tar files with contains all Jars needed to run RapidSmith2 from the command line. This includes a full jar of the RapidSmith2 build alond with copies of dependency Jars (such as QT-Jambi).

After adding the appropriate .class files or Jars to your `CLASSPATH`, you should be able to run RapidSmith2 tools from the command line. If you make any changes to the RapidSmith2 code, you will have to rebuild before running the program again (Step 3). **CAUTION:** An obvious thing to try is to mix and match developing in Eclipse but then running the resulting apps from the command line. Just be aware that Eclipse puts its compiled .class files in very different places than where the gradle build process puts its .class and .jar files. Make sure you understand that before you try to combine these two build/execution methods. Our suggested approach is to choose one or the other, but not both.

3.1.3 Docker Support

RapidSmith2 is also available as a Docker container. To painlessly set up a working RapidSmith2 environment, type:

```
docker run -it byuccl/rapidsmith2
```

For more information about Docker, see the guide.

3.1.4 Additional Notes for Mac OS X Installation

The instructions above require you to set the `RAPIDSMITH_PATH` environment variable. If running from the command line, the environment variables can be added to your *.bash_profile* file as in any other UNIX-like system. However, if using an IDE such as Eclipse you either need to define the environment variable for every Run Configuration you create, or you need to add the `RAPIDSMITH_PATH` definition system-wide in OS X. This can be done, but how to do so differs based on what OS X version you are running (and seems to have changed a number of times over the years). Search the web for instructions for how to do so if you desire. **Hint**: you will likely have to edit some *.plist* files.

3.1.5 Running RapidSmith2 Programs

Some points to keep in mind while configuring and running RapidSmith2 programs:

- The RapidSmith2 code base contains a number of assertions which may be helpful as you are developing code. These are not enabled by default in Java. To enable them, add *-ea* as a VM argument. This is highly recommended.

- If you are running on a Mac, when running RapidSmith2 programs that use Qt (any of the built-in programs like **DeviceBrowser**) that are GUI-based, you will need to supply an extra JVM switch, *-XstartOnFirstThread*.
- A common error when running RapidSmith2 programs is failing to have your `RAPIDSMITH_PATH` defined. If this is the case when you try to execute a program, an `EnvironmentException` will be thrown telling you that you forgot to set the variable.
- If you are running on Windows, only a 32-bit QT Jar file is included in the RapidSmith2 repository. This means that you will need to set your JRE to a 32-bit version when running the GUI programs. We are working on updating QT to the latest version, so this will no longer be an issue.
- For Linux command line usage, the `CLASSPATH` environment variable must point to both the full (uncompressed) RapidSmith2 jar in the *build/distributions* folder as well as all the jar files in the */lib* subdirectory. An example `CLASSPATH` could look like this:

```
RAPIDSMITH2-SNAPSHOT/*:RAPIDSMITH2-SNAPSHOT/lib/*
```

3.1.6 Testing Your Installation

At this point you can test your installation by executing the java **DeviceBrowser** program:

```
java edu.byu.ece.rapidSmith.device.browser.DeviceBrowser
```

This can be done either from within Eclipse or from the command line, depending on how you are running RapidSmith2 (if running under OS X be sure to provide the *-XstartOnFirstThread JVM argument*. If all goes well you should see a graphical representation showing the details of a physical FPGA device as shown in Figure 3. You may initially be zoomed far in and might want to zoom out to see the entire chip layout.

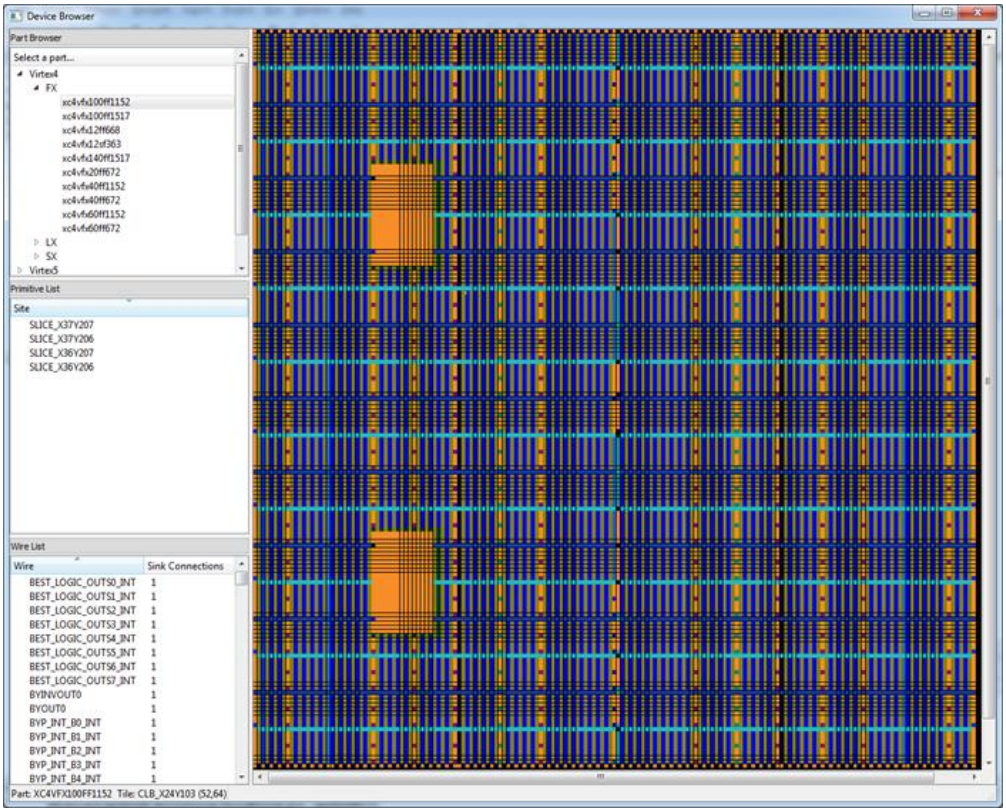


Figure 3: DeviceBrowser Sample Display

3.2 Running Real Designs - An Overview

This section will lead you through running an entire design from Vivado through RapidSmith2 and back. In this example you will fully synthesize and implement a complete design in Vivado and then move it into RapidSmith2. However, later as you gain experience you may choose to only synthesize designs before exporting them to RapidSmith2 or you may choose to export and work with Vivado Out-Of-Context (OOC) Checkpoints.

The steps to be covered include:

1. Synthesize, place and route a design in Vivado
2. Export the design from Vivado in the form of an RSCP (RapidSmith Checkpoint)
3. Import the RSCP into RapidSmith2
4. Run some analysis on the design
5. Export the design from RapidSmith2 as a TCP (Tincr Checkpoint)
6. Import the TCP back into Vivado

In a real CAD flow you would likely do something more interesting than the analysis in Step 4 above, but this overview is intended to help you through the Vivado-to-RapidSmith2 and back process so you can get started.

Note: the entire flow requires the installation of Tincr as well as RapidSmith2 so if you skipped the Tincr installation above, go and install it now.

3.2.1 Generating a RSCP

The first step in manipulating a Vivado design in RapidSmith2 is to generate a RapidSmith Checkpoint (RSCP) that fully represents the design. To do this, open Vivado in "Tcl mode" (this can be done by using the "Vivado Tcl Shell" or executing "vivado -mode tcl" if Vivado is correctly set on your `PATH`). Once the shell has started, execute the following Tcl commands for one of the directories in the *exampleVivadoDesigns* folder of the repository:[1]:

```
Vivado% cd <path to directory containing your HDL files>
Vivado% link_design -part xc7a100t-csg324-3
Vivado% read_verilog [glob *.v]
Vivado% synth_design -top myTopLevelEntityName -flatten_hierarchy full
Vivado% place_design
Vivado% route_design
Vivado% package require tincr
Vivado% tincr::write_rscp myTopLevelEntityName
Vivado% close_project
```

Most of the above commands should be self-explanatory and can be adapted to compile VHDL or SystemVerilog files. It is very important to note that the design **must be fully-flattened during sythesis**. Currently VDI and RapidSmith2 only support fully-flattened netlists. After all commands have finished executing, a RSCP should be created that can be loaded into RapidSmith2. As already mentioned, this shows how to create a fully placed and routed design in Vivado prior to export. There is no requirement, however, that the design is either placed or routed prior to export. You may choose to export it any time after it has been synthesized.

3.2.2 Importing the RSCP into RapidSmith2

The RapidSmith2 repository comes with an **importExportExample** program as described in subsection 9.4, that can be used to load a RSCP into RapidSmith2 data structures. The program can be found in the *edu.byu.ece.rapidSmith.examples* package. It takes as input a RSCP, and then performs the following steps:

1. Converts the RSCP to a RapidSmith2 `CellDesign` netlist (top-level netlist and is described in more detail in section 5.2.1).

[1] You can choose to implement the design any way you feel most comfortable with. In this walkthrough we use Vivado's command prompt because it is fastest.

2. Walks through the top-level `CellDesign` and prints the logical and physical information of the design (i.e. netlist structure, placement, and routing).

3. Exports the `CellDesign` to a TCP that can be loaded back into Vivado.

To learn a little more about the available RapidSmith2 data structures and method calls, you should examine the code to understand how the program works. Once you feel comfortable with the example code, run the program on the RSCP that you generated in the previous step.

3.2.3 Importing a TCP into Vivado

Assuming the original RSCP was named "add.rscp", the **importExportExample** program will produce a TCP called "add.rscp.tcp." After starting Vivado in Tcl mode ("vivado -mode tcl"), the generated TCP can be loaded back into Vivado by using the following set of Tcl/Tincr commands:

```
Vivado% cd <path to directory containing your add.rscp.tcp directory>
Vivado% package require tincr
Vivado% tincr::read_tcp add.rscp.tcp
Vivado% start_gui
```

The last command will open the Vivado GUI where you can see all of the cells and nets associated with this design along with where they have been placed/routed. This design is functionally equivalent to the original design converted to a RSCP. To save the imported design in Vivado's native checkpoint format (for later use) you can run the following command:

```
Vivado% write_checkpoint -force add.dcp
```

To reload the saved design back into Vivado, use the following Tcl commands:

```
Vivado% link_design -part xc7a100t-csg324-3
Vivado% open_checkpoint add.dcp
```

As you may have noticed, the flow presented in this section is based on running Vivado in "Tcl mode." The same set of commands could potentially be run in Vivado's GUI console, but they execute **significantly slower** than on the command line. When using this flow to implement CAD tools in RapidSmith2, it is *strongly encouraged* that you use the Vivado command line. Otherwise, importing and exporting designs will take much longer than needed. **NOTE:** For large designs, a TCP can take a long time to import back into Vivado.

4 Devices

4.1 Xilinx FPGA Architecture Overview

This section is intended to give a brief introduction to Xilinx FPGA architecture and terminology. The terminology introduced here is consistent with the terminolgy used in the Vivado Design Suite. If you are already familiar with Xilinx FPGA devices, then you can skip to subsection 4.2. As you read through this section, it may be helpful to open a sample device in Vivado's Device Browser. To do this, open a new command prompt and run Vivado in Tcl mode ("vivado -mode tcl"). Then, run the following commands in the Vivado prompt:

```
Vivado% link_design -part xc7a100tcsg324-3 -quiet
Vivado% start_gui
```

Replace "xc7a100tcsg324-3" with any Xilinx part you are interested in looking at. After these commands are run, a GUI view should pop up showing the components of an Artix7 FPGA part. Use this to explore the Xilinx device architecture if needed.

4.1.1 Device Hierarchy

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 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 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".

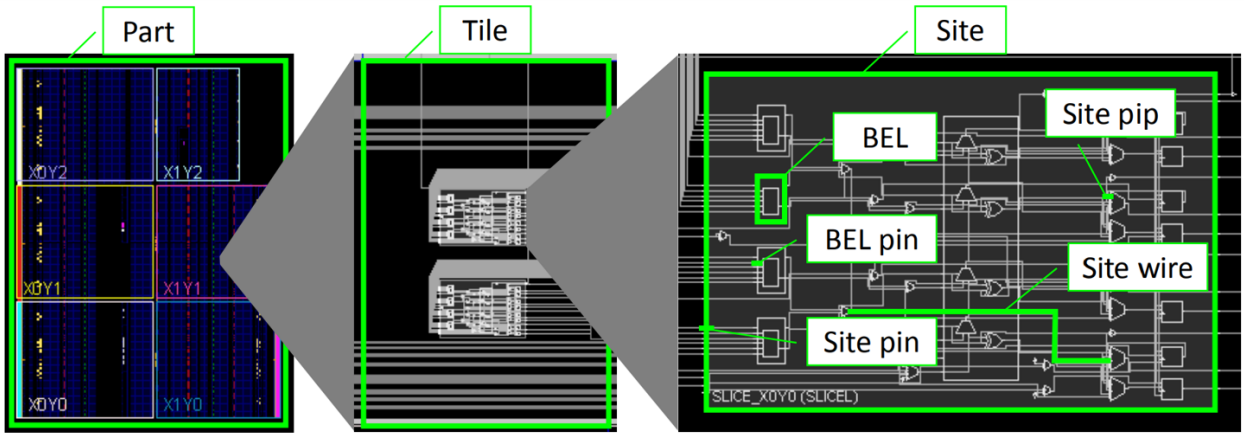


Figure 4: Xilinx Device Hierarchy

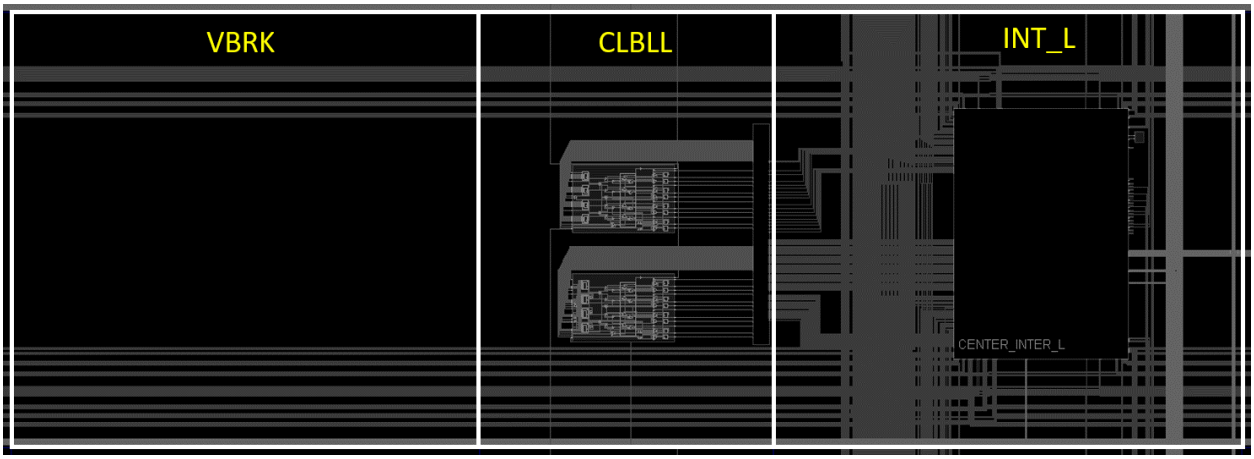


Figure 5: Artix7 Tiles

Figure 4 shows the device hierarchy of a Xilinx part . The following subsections describe each internal component of a Xilinx FPGA as shown in the figure.

4.1.2 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 5 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.

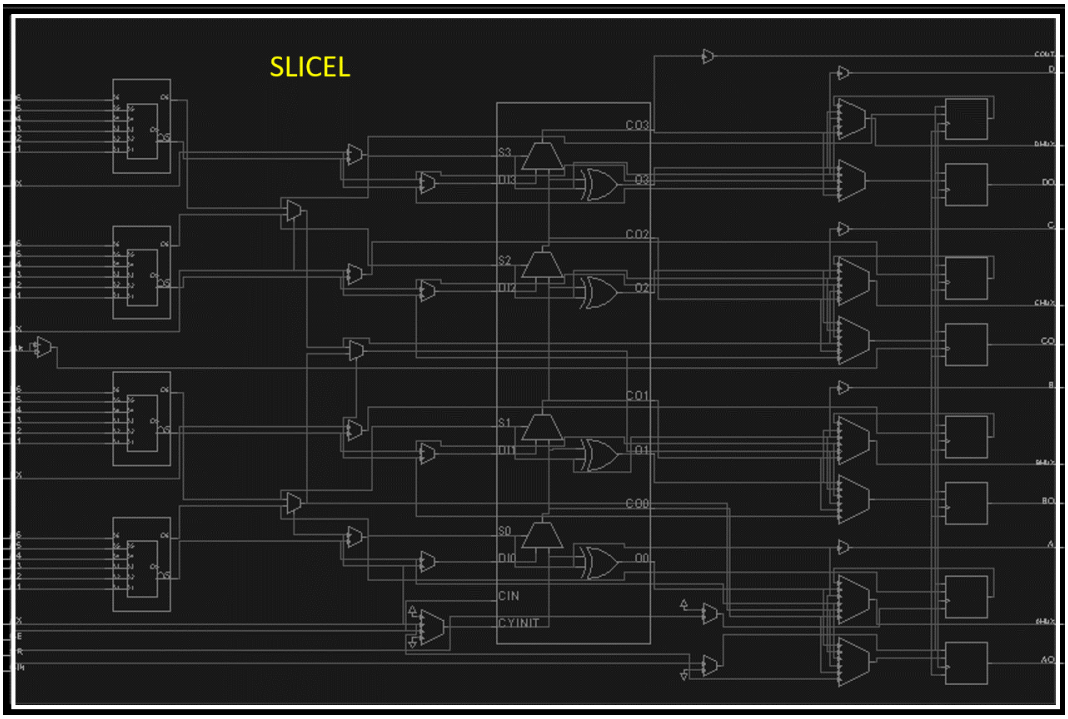
4.1.3 Sites

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 6 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:

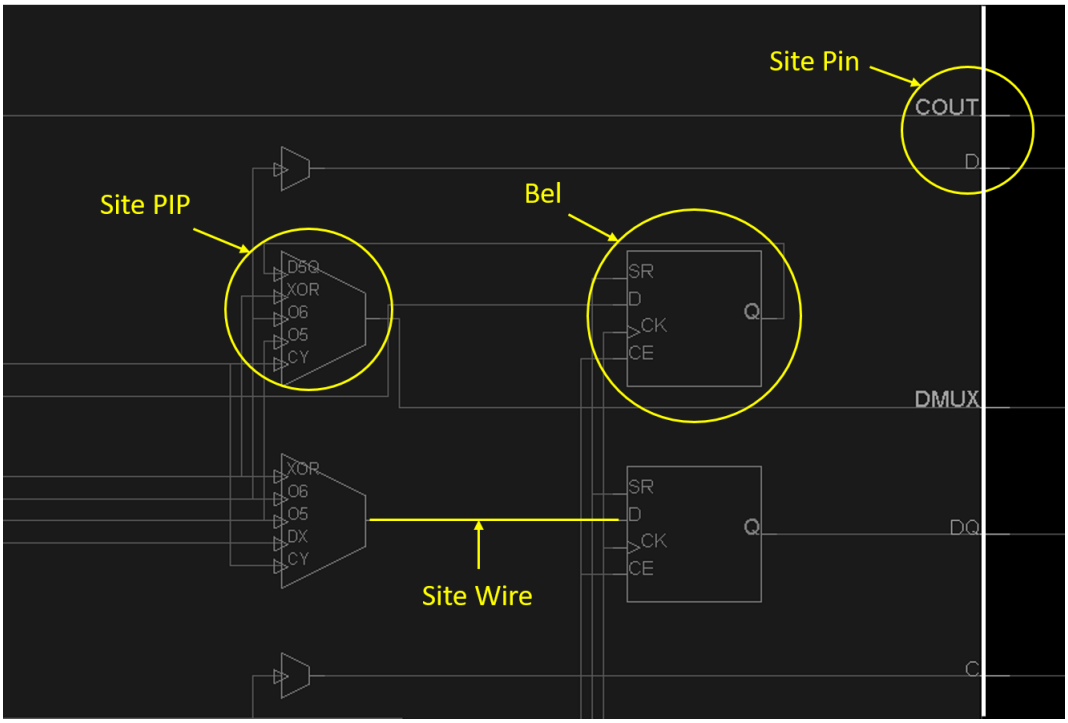
- **Site PIP**s: Also called routing muxes, these are reconfigurable routing PIPs used to specify the internal routing of a site. In Vivado, site PIPs are usually configured automatically as cells in a design are placed (based on cell properties and placement locations).
- **BEL**s: **B**asic **EL**ements 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.

4.1.4 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.



(a)



(b)

Figure 6: Series7 SLICEL Site (a), and Highlighted Site Components (b)

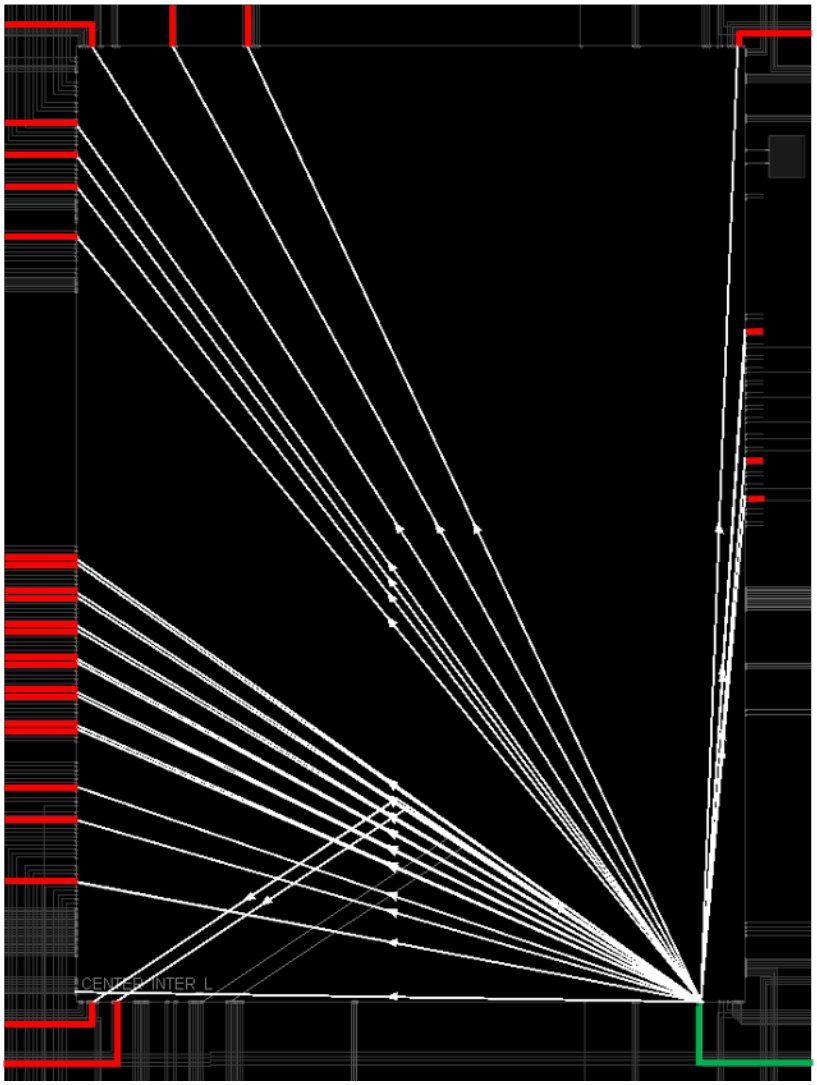


Figure 7: An example switchbox tile. The green wire represents a source wire, and the red wires represent all possible sink wires in the switchbox. The highlighed white sections of the figure are PIP connections.

PIPs are most commonly found in switchbox tiles, and enable a single wire to be routed to several locations on the chip. Figure 7 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.

4.2 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 8 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 `DeviceBrowser` and `DeviceAnalyzer` example programs illustrate how to load and browse a device with `Tile` and `Site` data structures, and Listing 1 shows basic device usage in RapidSmith2. The remainder of this section details important aspects of RapidSmith2 devices.

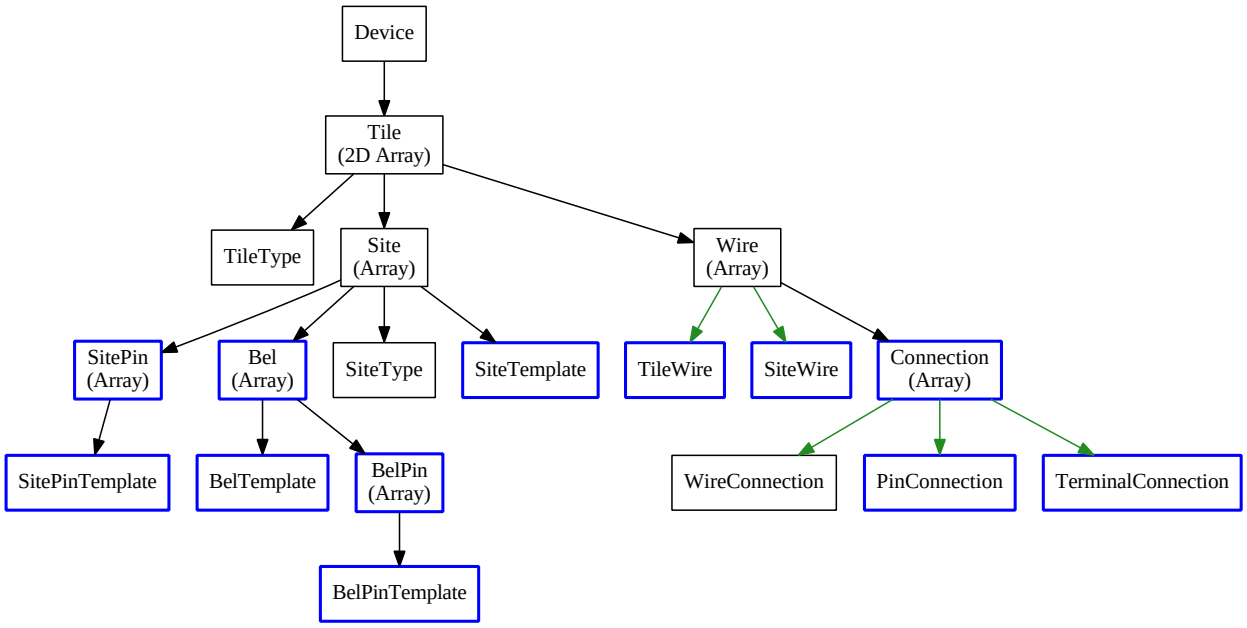


Figure 8: RapidSmith2 Device data structure tree. Green arrows represent inheritance, and black arrows represent association. Classes and Interfaces bolded in blue are new to RapidSmith 2.

Listing 1: Basic device function calls

```
// Get a handle to a device
Device device = Device.getInstance("xc7a100tcsg324-3");

// 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();
```

4.2.1 Templates

As Figure 8 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.

4.2.2 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 which contain a wire's integer enumeration and name. 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.

4.2.3 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 RapidSmith2 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 7.

4.2.4 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 2 shows some available package pin method calls.

Listing 2: RapidSmith2 package pin functions

```
// 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();
```

4.3 Loading a Device

Listing 3 below 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.

Listing 3: Loading a Device

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

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.

4.4 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. However, RapidSmith2 has general support for the following families:

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

Section 10 describes how to create new device files for these families and add them to RapidSmith2. ;

5 Designs

5.1 Xilinx Netlist Structure

During the synthesis stage of implementation, a digital circuit expressed using RTL (VHDL, Verilog, or SystemVerilog) 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 here for Series7 devices and here 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 9 shows an example netlist for a 3-bit counter created in Vivado.

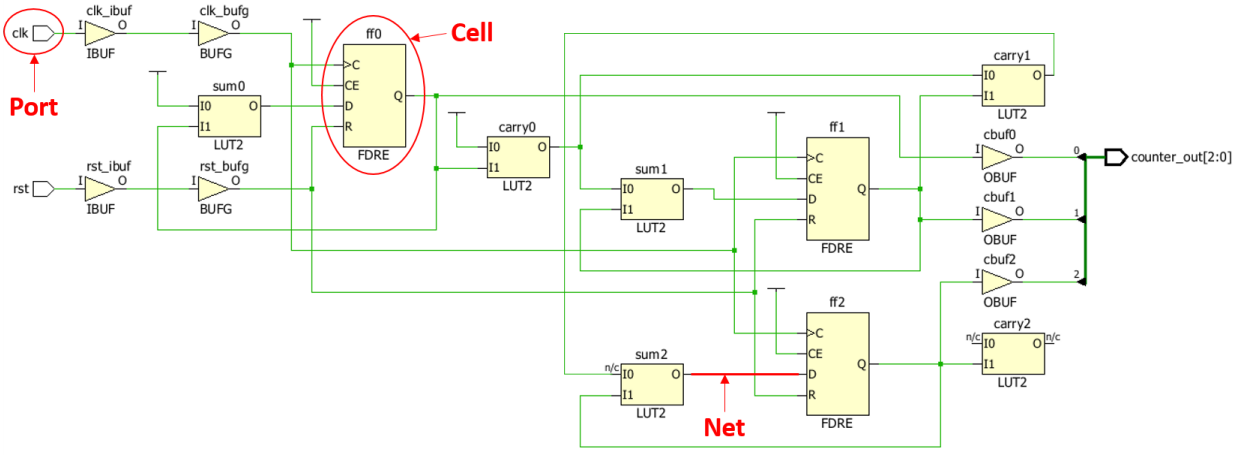


Figure 9: Schematic of a 3-bit counter in Vivado using LUT and FDRE cells. The yellow boxes are cells, the green lines are nets, and and the white figures on the edge of the diagram are ports.

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 $O6 = (A1 + A2) \oplus A3$.
- **Flip-Flops** (FDxx): Single-bit storage elements. Figure 9 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).

5.2 RapidSmith2 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 10.

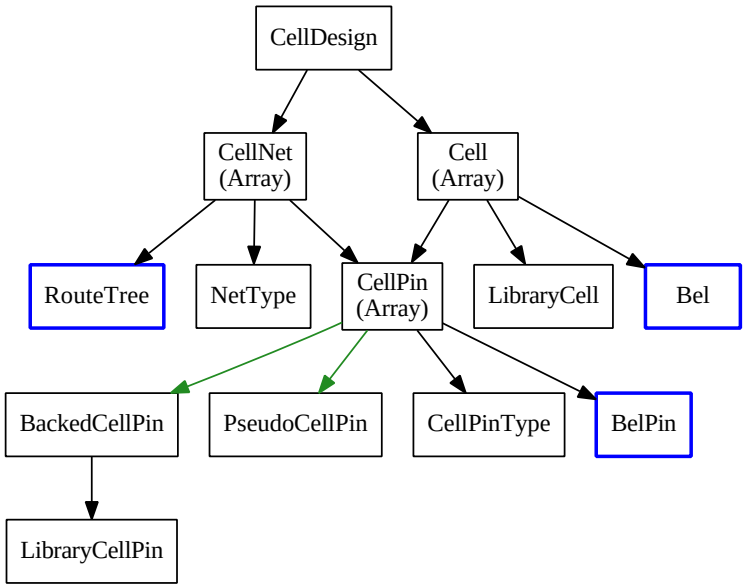


Figure 10: RapidSmith2 design data structure tree. Black arrows represent composition, green arrows represent inheritance, and blue boxes are physical implementation components of the netlist.

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 6 and 7 respectively. The best way to learn how to utilize the classes shown in Figure 10 is to generate and read through the JavaDocs, but important aspects of each class is included in the following subsections.

5.2.1 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 8 for more information about XDC constraints and how they are represented in RapidSmith2.
- The design mode. Possible options include OUT_OF_CONTEXT and REGULAR. OUT_OF_CONTEXT designs are those that have been implemented in Vivado "out-of-context", meaning that the top-level ports are not placed or routed. Most designs are implemented in REGULAR mode.

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.

5.2.2 Cell

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

- A `Cell` always contains a reference to a backing `LibraryCell` object. A `LibraryCell` is equivalent to a Xilinx primitive cell (described in subsection 5.1), and serves as a template for instantiated `Cell` objects. The template is used to save memory when creating several `Cell`s of the same type. Whenever a new `Cell` object is created, a corresponding `LibraryCell` must be specified in the constructor. Listing 4 demonstrates how to create new cells in RapidSmith2 and filter cells based on their type.

Listing 4: How to create new cells in RapidSmith2

```
// 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 subsubsection 5.3.1).
- `Cell`s can be placed onto `Bel` objects of the current `Device`. See section 6 for more information about cell placement.
- Top-level `Port`s in Vivado (design input/output) are represented as Port `Cell`s 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 subsection 5.1.

5.2.3 CellPin

`CellPin`s 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 2 displays the possible values for both of these properties. 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 subsubsection 5.3.1. 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.

5.2.4 CellNet

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

Table 2: Cell Pin Types and Directions

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

Listing 5: Basic CellNet functions

```
// get a handle to a design
CellDesign design = getCellDesign();

// creating a new net
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"));
```

- `CellNet`s are routed using `RouteTree` data structures. RapidSmith2 routing is described in more detail in section 7.
- All `CellNet`s 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 11 shows an example for a GND net. RapidSmith2 handles this oddity by allowing `CellNet`s to have more than one `RouteTree` object associated with it. In the case of Figure 11 the net would have two `RouteTree`s, one for each source TIEOFF.

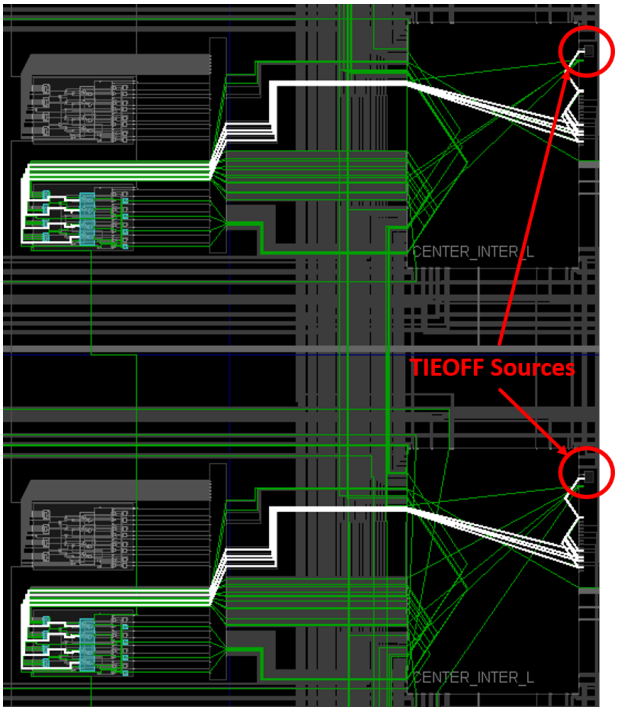


Figure 11: Example of a Vivado net with multiple sources. The highlighted wires in white are all part of the same net.

- Figure 12 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()`.

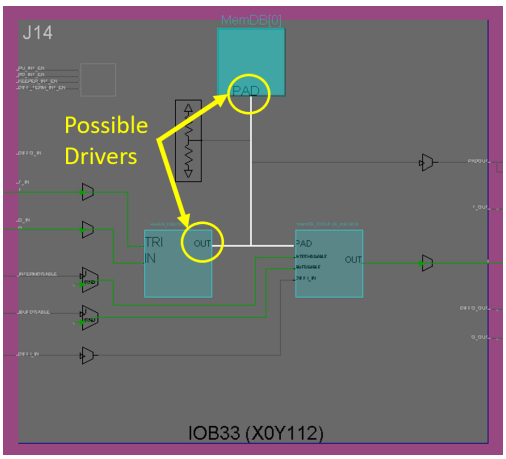


Figure 12: Bidirectional Net

- After a design has been placed, `CellNets` fall into one of two categories: **intrasite** vs. **intersite**. Figure 13 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.

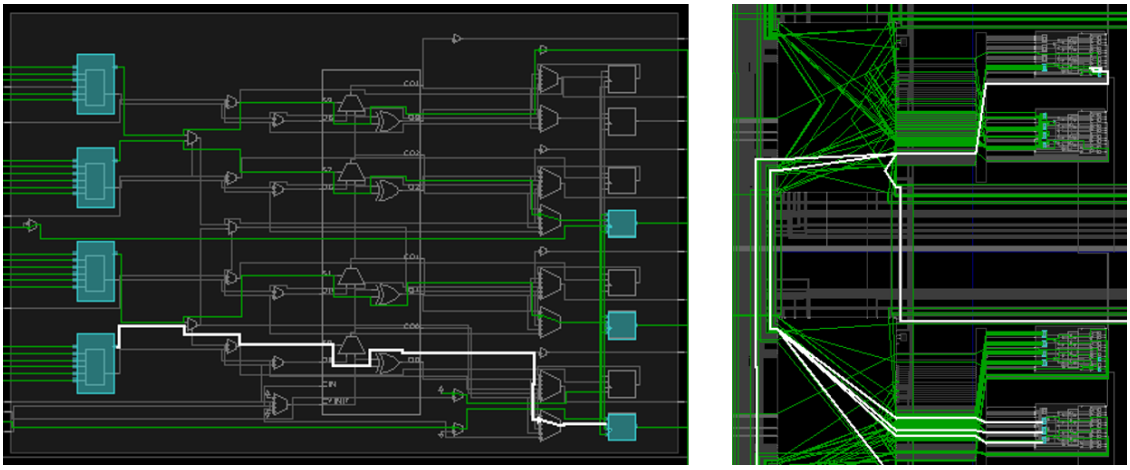


Figure 13: Example INTRASITE Net (left) and INTERSITE Net (right)

5.2.5 Macro Cells

Most cells in RapidSmith2 or Vivado designs are leaf cells (LUTs, Flip Flops, etc.), but 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 14 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. External cell pins of the macro connect to one or more internal cell pins.

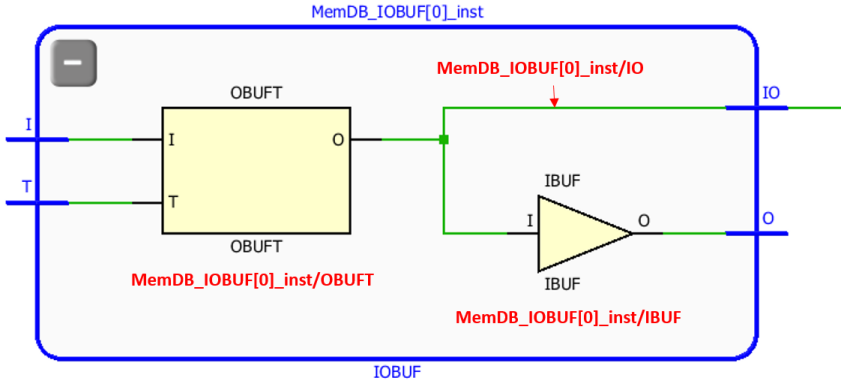


Figure 14: Vivado Macro Cell

RapidSmith2 now supports importing macro cells from Vivado and adding them to a `CellDesign`. Listing 6 gives a brief introduction to using macros in RapidSmith2.

Listing 6: How to use macros in RapidSmith

```
1 // Get a handle to a design and cell library
2 CellDesign design = getCellDesign();
3 CellLibrary libCells = getCellLibrary();
4
5 // Create a new macro cell and add it to a design
6 Cell macro = new Cell("myMacro", libCells.get("IOBUF"));
7 design.addCell(macro);
8
9 // Connect the macro cell to a net
```

```
CellPin pin = macro.getPin("IO");
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, but the design will still be functionally equivalent.

5.2.6 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 15 shows a list of

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 15: Properties of a Vivado FDCE Cell

properties for a FDCE flip flop cell in Vivado. 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 7 shows some basic property usage.

Listing 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 [2].
- 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 if you want to export the property to Vivado. All other properties will be ignored during design export.

[2] 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

5.2.7 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 tool. 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.

5.3 Vivado Design Considerations / Advanced Topics

The previous section detailed the most important aspects of a Vivado netlist, and how they are represented in RapidSmith2. However, there are some subtle considerations for Vivado implemented designs that you need to understand in order to fully utilize RapidSmith2 as a CAD tool. These design aspects, and how RapidSmith2 chooses to handle them is described in the following subsections.

5.3.1 Pseudo Cell 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 16. 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?

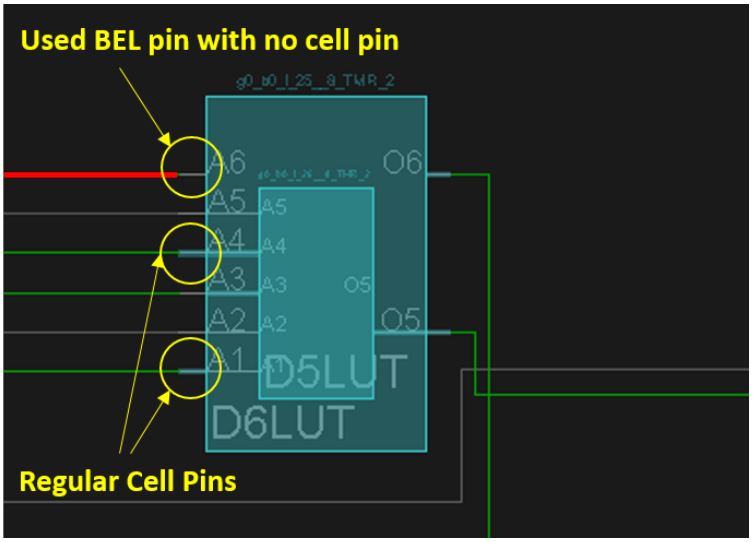


Figure 16: An example of VCC routing to an unused BEL pin (A6)

To address this issue, RapidSmith2 allows users to create and attach `PseudoCellPins` to an existing cell. A `PseudoCellPin` is a "fake" cell pin that can be attached to a cell (after the cell has been created), and then attached to a net to create a more complete view of the netlist. For example, a `PseudoCellPin` can be attached to the cell shown in Figure 16, attached to the VCC net of the design, and then mapped to the `A6` pin. Assuming the cell in the figure has the name of "foo", Listing 8 demonstrates how to create and attach a new `PseudoCellPin`.

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

```
1   // get a handle to a device and design
```

```
CellDesign design = loadDesign();
Device device = loadDevice();

// get a handle to the appropriate cells, nets, and bel pins
Cell cell = design.getCell("foo");
CellNet vcc = design.getVccNet();
BelPin bp = device.getSite("SLICE_X0Y179").getBel("D6LUT").getBelPin("A6");

// create and attach the psuedo cell pin, and map it to the BelPin
CellPin pseudo = cell.attachPseudoPin("VccTmpPin");
vcc.connectToPin(pseudo);
pseudo.mapToBelPin(bp);
```

5.3.2 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. Figure 17 shows two example routethrough LUTs in Vivado. As described in section 7, routethroughs are represented in RapidSmith2 with specific `Connection` objects, and can be used when routing a net.

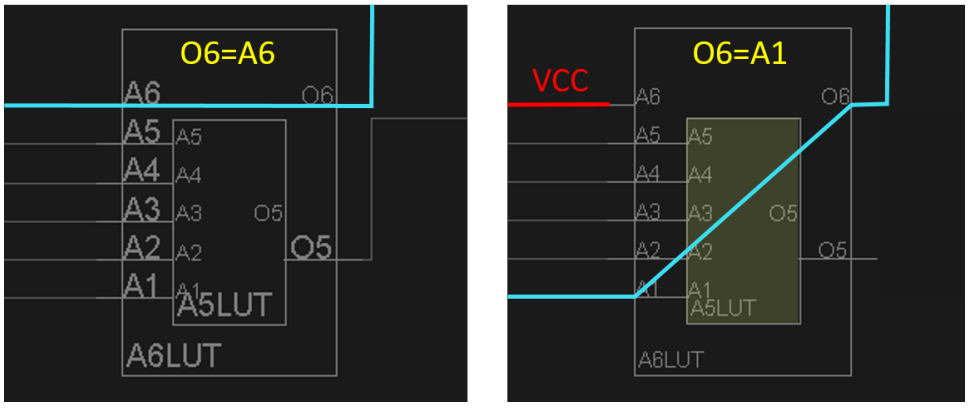


Figure 17: Two examples of LUTs configured as routethroughs in Vivado. The net highlighted in red represents VCC.

5.3.3 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

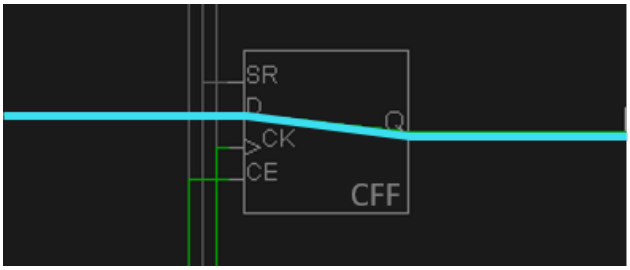


Figure 18: Flip-Flop BEL Configured as a Permanent Latch in Vivado

is retained. An example is shown in Figure 18. As the figure shows, permanent latches look very similar to LUT routethroughs described in the previous section. Because of this similarity, RapidSmith2 treates permanent latches the same as LUT routethroughs.

5.3.4 Static Source LUTs

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 19. 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. RapidSmith2 does explicitly represent static source LUTs. Like other VCC and GND sources, they are implied based on where a VCC/GND route begins.

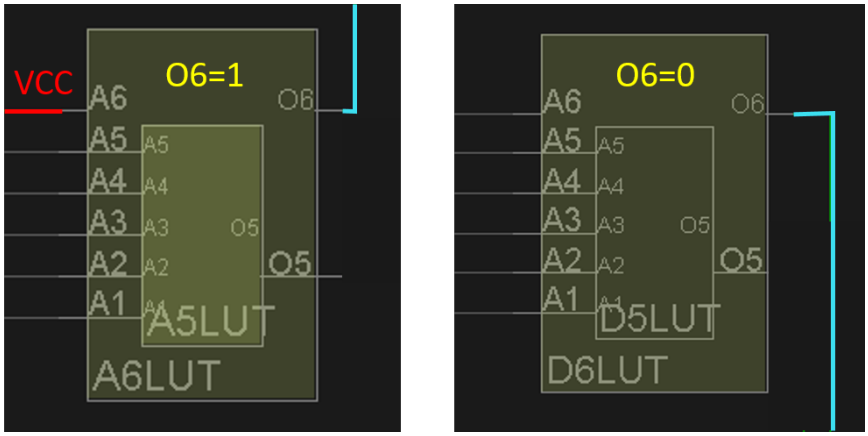


Figure 19: Two LUTs Configured as Static Sources in Vivado

5.3.5 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 20 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. In RapidSmith2, a list of used site PIPs is stored for each site on design import. The function call `CellDesign::getUsedSitePipsAtSite(Site)`

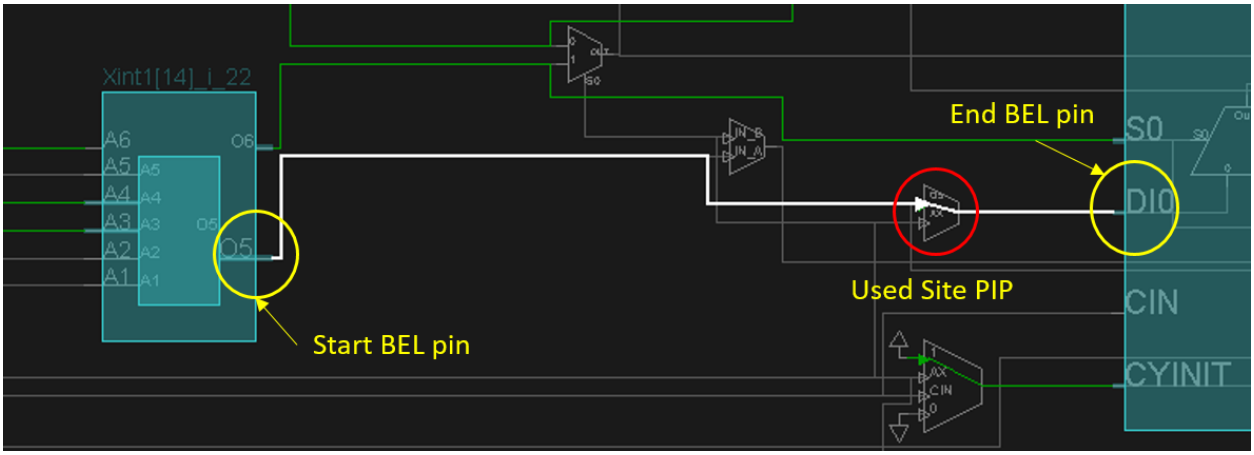


Figure 20: Site PIP Usage

can be used to obtain the used site PIPs at a given site location. After design import, it is the user's responsibility to update this data structure accordingly.

5.3.6 Site Properties

Some properties not only affect how a Cell is configured, but they can also affect how a Site is configured. For example, on SLICEL sites there exists a clock routing mux that chooses between a regular clock signal and an inverted signal (Figure 21). The output of this inverter is connected to all flip flops of the SLICEL, and decides whether *all* flip flops are rising or falling-edge triggered. The clock that is selected is determined by the property IS C INVERTED **of the flip flops cells that have been placed onto the SLICEL**. The clock inverter is programmed automatically based on the cell properties. There are other such properties, but they will not be listed here. When creating designs in RapidSmith2, it is important to not place cells together that may have conflicting properties. RapidSmith2 does not perform any error checking, and so an error in Vivado will be thrown if you violate property restrictions.

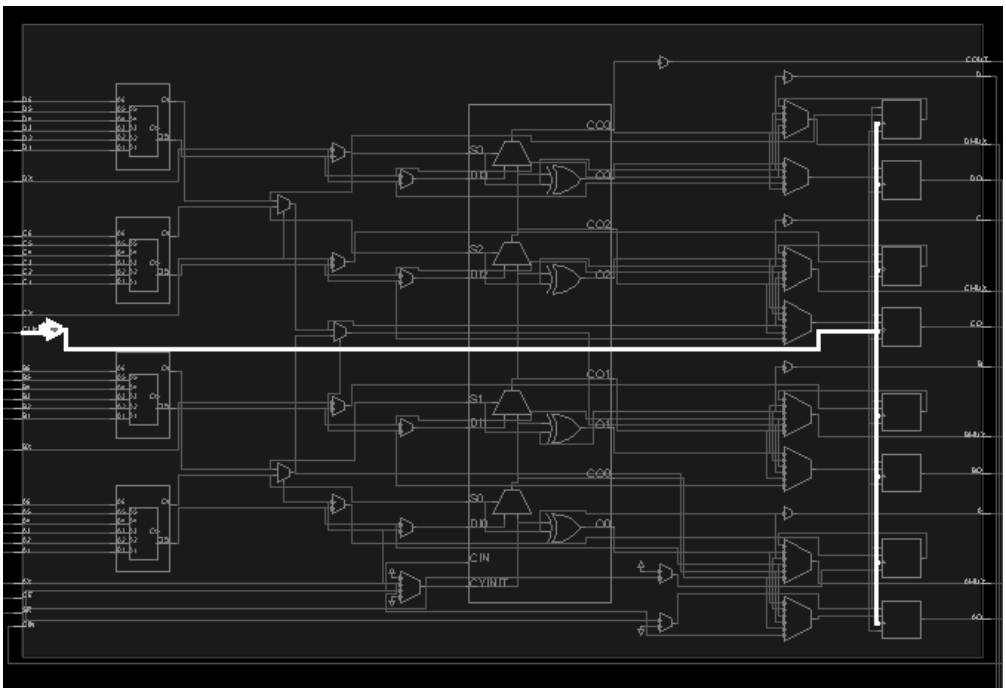


Figure 21: SLICEL clock inverter mux.

5.4 The Cell Library

As described in subsection 5.1, 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 Flip-Flop (FDRE, FDCE, etc.) cells. 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, `Tincr` defines 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

RapidSmith2 parses the cell library XML file described above into a `CellLibrary` data structure. This data structure is very useful when performing any type of netlist manipulation or addition. 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 [3]. Listing 9 shows two ways to load a `CellLibrary` in RapidSmith2. The following subsections detail important aspects of a cell library.

Listing 9: Loading a CellLibrary in RapidSmith2

```
// First way, load a Tincr Checkpoint
VivadoCheckpoint vcp = VivadoInterface.loadRSCP("checkpoint.rscp");
CellLibrary libCells1 = vcp.getLibCells();

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

5.4.1 Generating A New Cell Library

The `Tincr` command `[tincr::create_xml_cell_library]` can be used to generate a new cell library for a device. Specifically, follow the steps listed below to create a new cell library (the items marked with **SERIES7** only need to be done for series7 families).

1. Open Vivado in Tcl mode, and run the command shown in the listing below. Replace "xc7a100tcsg324-3" with the part you want to generate and "mycellLibrary.xml" with the location where you want to store the generated cell library XML. This will generate most of what you need in the cell library XML automatically.

```
Vivado% ::tincr::create_xml_cell_library xc7a100tcsg324-3 mycellLibrary.xml
```

2. **SERIES7**: Open the generated XML file in a text editor and search for the "CARRY4" cell. Scroll down to the "bels" XML element within the CARRY4 cell, and add the following lines to each pin that is named "CI":

```
1122        </pin>
1123        <pin>
1124          <name>CI</name>
1125          <possible>CIN</possible>
1126          <possible>CYINIT</possible>
1127        </pin>
1128        <pin>
```

   You should have to insert this line in two places only.

3. **SERIES7**: Save your changes and exit the text editor
4. Copy the XML file to *RapidSmithPath/device/family* directory where "family" is replaced by the family of your part (such as artix7), and "RapidSmithPath" is the location of your RapidSmith repository. Once this is complete the new `CellLibrary` should be ready to use.

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

5.4.2 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 an XML specification of your macro that follows the format laid out below:

Listing 10: Sample Macro XML for a CellLibrary

```
<?xml version="1.0" encoding="UTF-8"?> <root>
  <macros>
     <macro>
       <type>RAM128X1D</type>
       <!-- List of internal cells with name and leaf type of each -->
       <cells>
          <internal>
             <name>DP.HIGH</name>
             <type>RAMD64E</type>
          </internal>
          <internal>
             <name>DP.LOW</name>
             <type>RAMD64E</type>
          </internal>
          <internal>
             <name>F7.DP</name>
             <type>MUXF7</type>
          </internal>
        ...
       </cells>
       <!-- List of macro pins with name, direction, pin type, and internal
           connections -->
       <pins>
          <pin>
             <name>DPO</name>
             <direction>output</direction>
             <type>MACRO</type>
             <internalConnections>
                <pinname>F7.DP/O</pinname>
             </internalConnections>
          </pin>
          <pin>
             <name>SPO</name>
             <direction>output</direction>
             <type>MACRO</type>
             <internalConnections>
                <pinname>F7.SP/O</pinname>
             </internalConnections>
          </pin>
          <pin>
             <name>A[6]</name>
             <direction>input</direction>
             <type>MACRO</type>
             <internalConnections>
                <pinname>F7.SP/S</pinname>
                <pinname>DP.HIGH/WADR6</pinname>
                <pinname>DP.LOW/WADR6</pinname>
                <pinname>SP.HIGH/WADR6</pinname>
                <pinname>SP.LOW/WADR6</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>
  ...
 </macros>
</root>
```

2. Import the macro into the `CellLibrary` using the API call shown in Listing 11.

Listing 11: Adding new macros to the Cell Library

```
1 // Get a handle to a CellLibrary
2 CellLibrary libCells = getCellLibrary();
3
4 // Add the macros in an XML file.
5 libCells.loadMacroXML(Paths.get("myMacro.xml");
```

Once this is complete, you can use your custom macro in a `CellDesign` like a normal cell.

6 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 12 demonstrates the basic steps to placing cells in RapidSmith2.

Listing 12: Steps for placing a Cell in RapidSmith2

```
1  // Load the device and design
2  VivadoCheckpoint vcp = VivadoInterface.loadRSCP("myCheckpoint.rscp");
3  Device device = vcp.getDevice();
4  CellDesign design = vcp.getDesign();
5
6  // Get a handle to a Cell and Bel. The cell is of type LUT2
7  Cell cell = design.getCell("MyCell");
8  Bel bel = device.getSite("SLICE_X40Y137").getBel("D6LUT");
9
10 // Place the cell onto the bel
11 design.placeCell(cell,bel);
12
13 // Two ways to map bel pins
14 CellPin pin1 = cell.getPin("I0");
15 CellPin pin2 = cell.getPin("I1");
16 CellPin pin3 = cell.getPin("O")
17
18 // First way
19 pin1.mapToBelPin(bel.getPin("A1"));
20 pin2.mapToBelPin(bel.getPin("A2"));
21
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 22 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)` 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 13. The correct pin mappings are always used when a RSCP is imported into RapidSmith2.

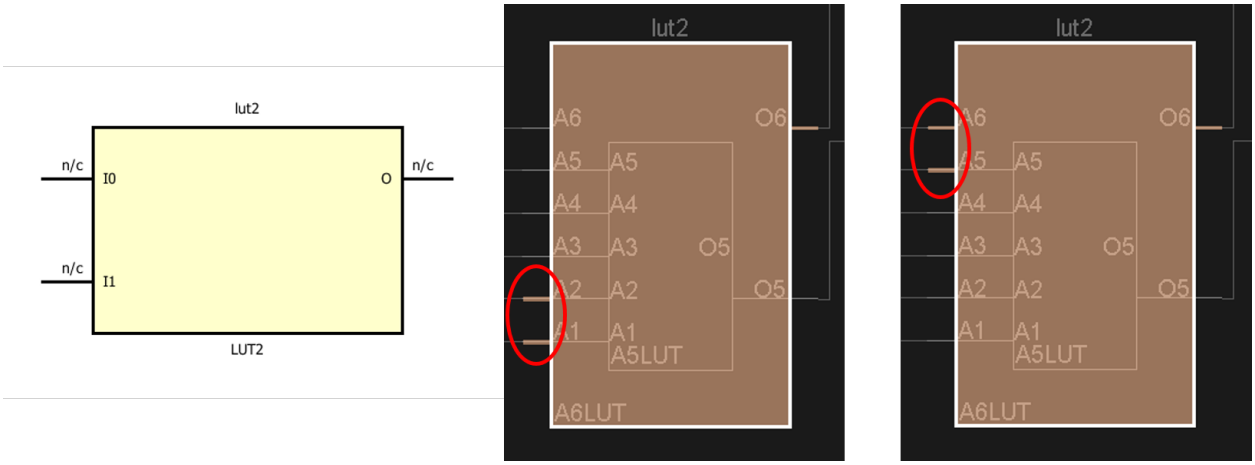


Figure 22: LUT Pin Permutation Example

Listing 13: 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, `RouteTree`s 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 locations 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.

7 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.

7.1 Wires and Wire Connections

Routing in RapidSmith2 is done using `Wire` objects, which are described in subsection 4.1.4. `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 7). 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 23. 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 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.

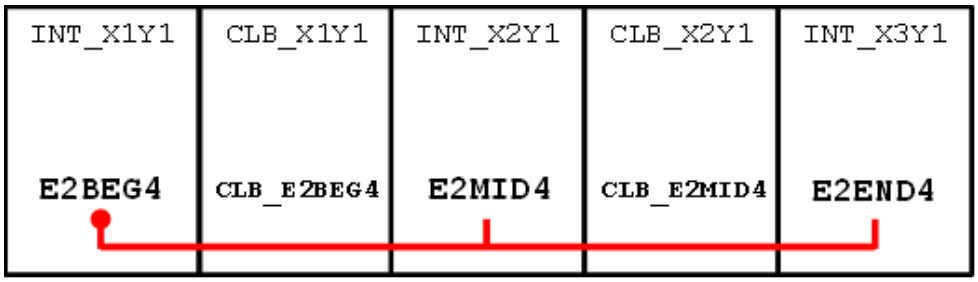


Figure 23: Multi-Tile Xilinx Wire

7.2 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 14 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.

7.3 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 14 demonstrates how to iterate through different `Connection` types in RapidSmith2.

Listing 14: How to iterate over Connections in RapidSmith2

```
1  // Get a handle to a wire
2  Wire wire = sitePin.getExternalWire();
3
4  // Iterate over all WireConnections
5  for (Connection conn : wire.getWireConnections()) {
6    Wire sinkWire = conn.getSinkWire();
7
8    if (conn.isPip()) {
9      // Do something with a PIP
10   }
11   else if (conn.isRoutethrough()) {
12     // Do something with a routethrough
13   }
14   else {
15     // Do something with a regular wire connection
16   }
17 }
18
19 // Get the site pin connected to a wire
20 SitePin pin = wire.getConnectedPin();
21 if (pin != null)
22   // Do something with the SitePin
23 }
24
25 // Get the bel pin connected to a wire
26 BelPin pin = wire.getTerminal();
27 if (pin != null)
28   // Do something with the BelPin
29 }
30
31 // Iterate over all connections at the same time
```

```
32   Iterator<Connection> connIt = wire.getAllConnections().iterator();
33
34   while (connIt.hasNext()) {
35     Connection conn = connIt.next();
36
37     if (conn.isPinConnection()) {
38       //do something with the site pin
39     }
40     else if (conn.isTerminal()) {
41       //do something with the bel pin
42     }
43     else if (conn.isPip()) {
44       //do something with the pip wire connection
45     }
46     else {
47       //do something with regular wire connection.
48     }
49   }
```

7.4 RouteTrees

`Wire`s and `WireConnection`s 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 24.

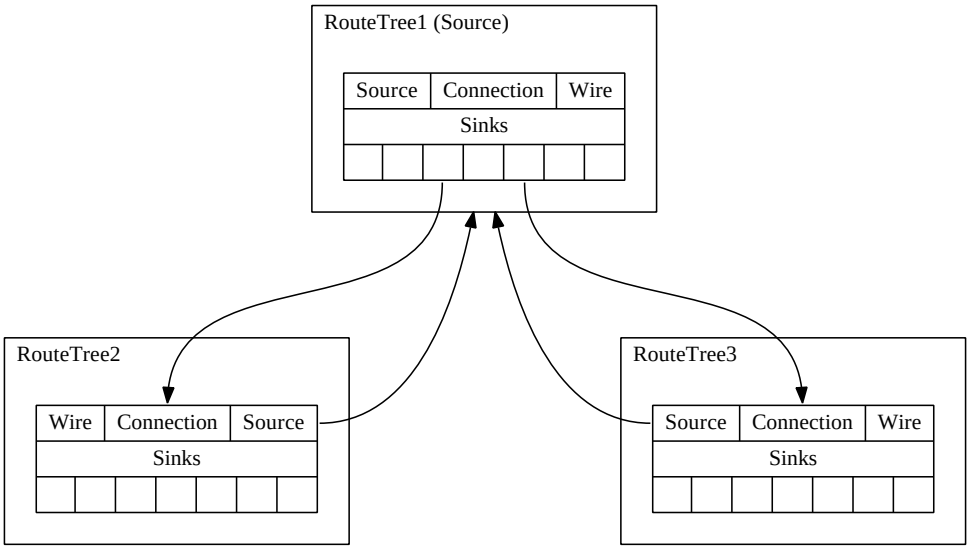


Figure 24: Visual Representation of a RapidSmith2 RouteTree

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.
- **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 25 shows an example of a complete `RouteTree` in RapidSmith2. As can be seen, the `CellNet` that is being routed has once source site pin, and two sink site 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

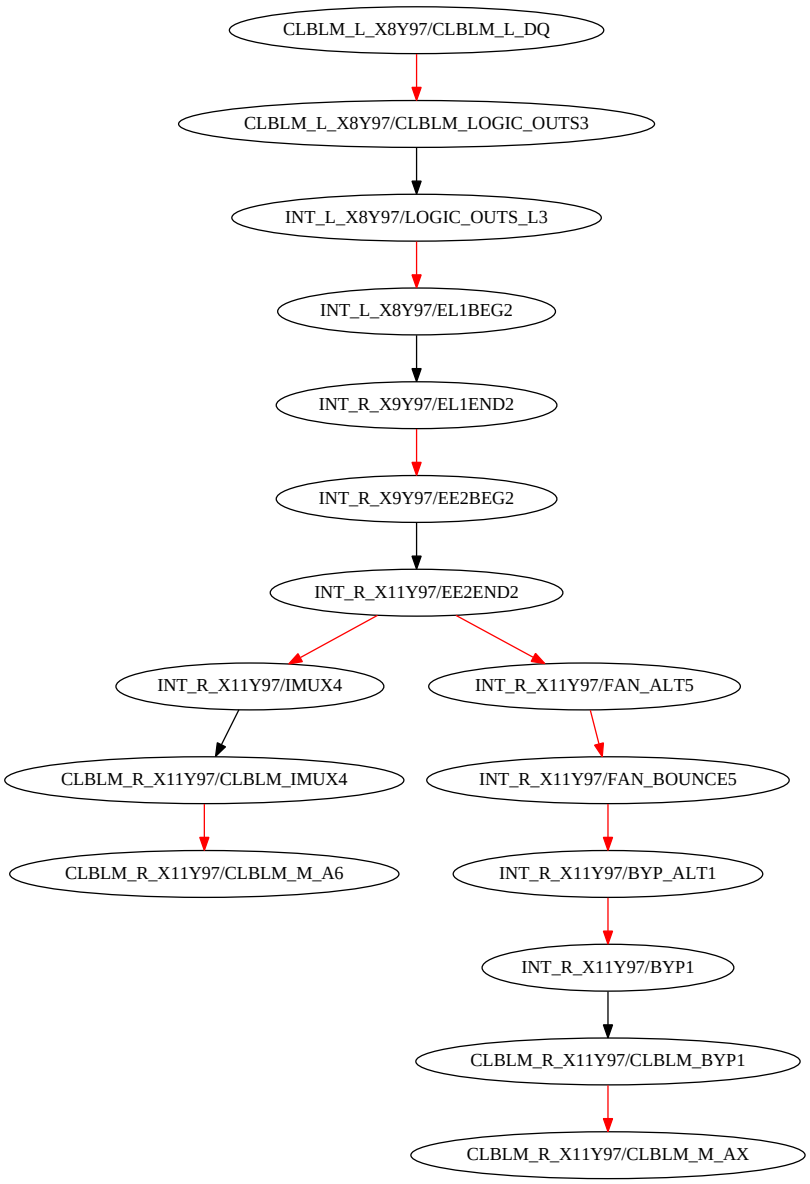


Figure 25: Sample RapidSmith2 RouteTree (red edges represent PIP connections)

are reached. Listing 15 demonstrates the basic usage of RouteTrees in RapidSmith2. The **DesignAnalyzer**, **AStarRouter**, and **HandRouter** examples in the RapidSmith2 repository also demonstrate how to traverse and build a RouteTree.

Listing 15: Building a RouteTree

```
// 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.

7.5 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[4].
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 26 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 because they are completely contained within a site. Listing 16 demonstrates how to utilize three-part routing in RapidSmith2. On design import, the routing sections of each CellNet are created automatically.

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

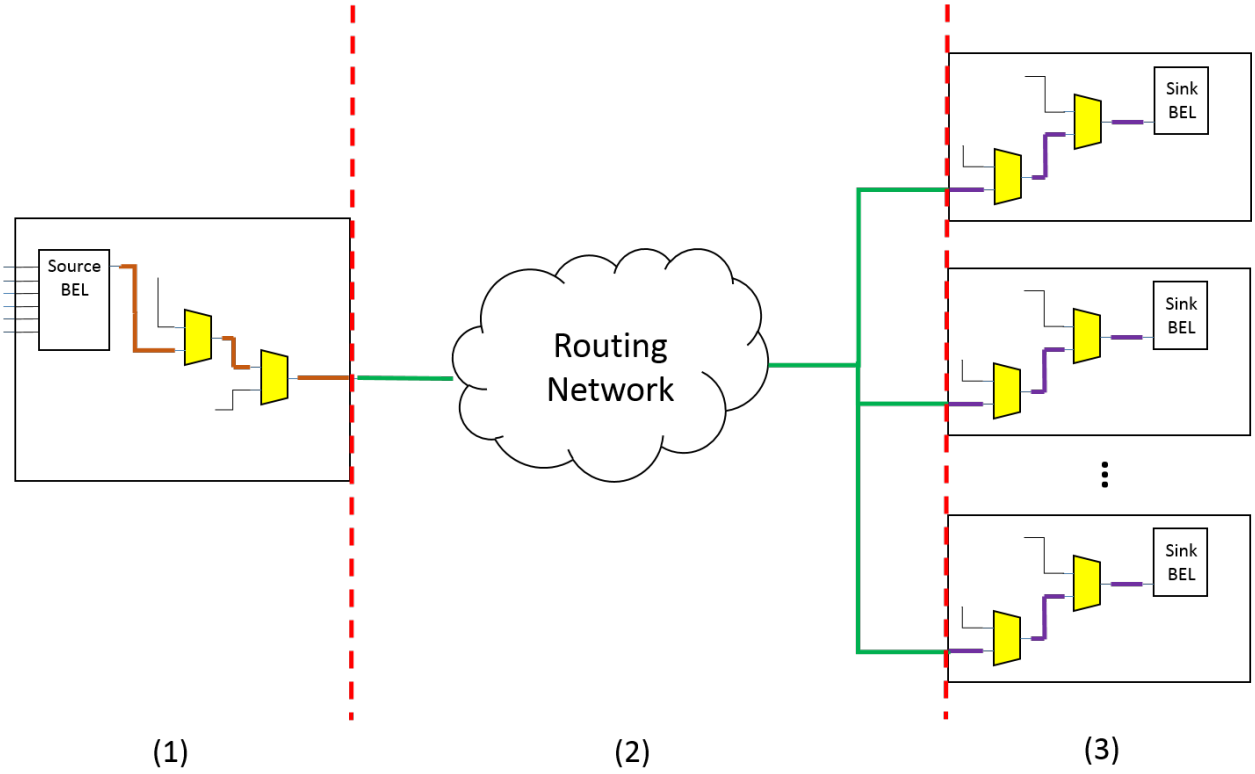


Figure 26: Three-Part Routing

Listing 16: 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));
```

7.6 Routing in Vivado

The reason a three-part routing distinction is necessary in RapidSmith2, is due to how routing is represented in Vivado. Inside site boundaries, a route is represented using site PIPs. A string of enabled PIPs determines what pins are connected together within the site. Between site boundaries, a route is instead represented using wires. The wires are formatted into a Vivado ROUTE string, which uniquely specifies the intersite route for a net. The three-part routing representation makes this distinction explicit to the users of RapidSmith (it also makes import/export easier). When a design is exported from Vivado, the intrasite portions of a net are exported as site PIPs, and the intersite portion of the net is exported as wires.

```
// An example of a string of used site pips in Vivado
{IUSED:0 IBUFDISABLE_SEL:GND INTERMDISABLE_SEL:GND}

// An example of a Vivado ROUTE string
 { CLBLL_LL_AQ CLBLL_LOGIC_OUTS4 { NW6BEG0 NE2BEG0 WR1BEG1 IMUX_L34
IOI_OLOGIC0_D1 LIOI_OLOGIC0_OQ LIOI_O0 } IMUX_L1 CLBLL_LL_A3 }
```

7.7 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 17 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 17: Code to transform a set of SiteWires into Site PIPs

```
1   // Get a handle to a Design and a Site
2   CellDesign design = vcp.getDesign();
3   Device device = vcp.getDevice();
4   Site site = device.getSite("SLICE_X5Y84");
5
6   // Get a list of used site wires somehow (this is up to you)
7   Set<Wire> usedSiteWires = getUsedWires(site);
8
9   // Convert the list of wires to their integer enumeration
10  Set<Integer> usedPipWires = usedSiteWires.stream()
11                          .map(w -> w.getWireEnum())
12                          .collect(Collectors.toSet());
13
14  // Set the used site pips with the design class
15  design.setUsedSitePipsAtSite(site, usedPipWires);
```

8 Design Import/Export

RapidSmith2 supports modifiying Vivado designs post-synthesis, post-place, and post-route as shown in Figure 27. As the figure shows, RapidSmith Checkpoints (RSCP) are generated from `Tincr` which are parsed and loaded into RapidSmith2 `CellDesign` data structures. After a CAD tool has been run, a RapidSmith2 `CellDesign` can be converted to a Tincr Checkpoint (TCP), which can then be loaded back into Vivado to complete the remainder of the implementation flow. This section details how to load a RSCP into RapidSmith2, and generate TCP from a `CellDesign`.

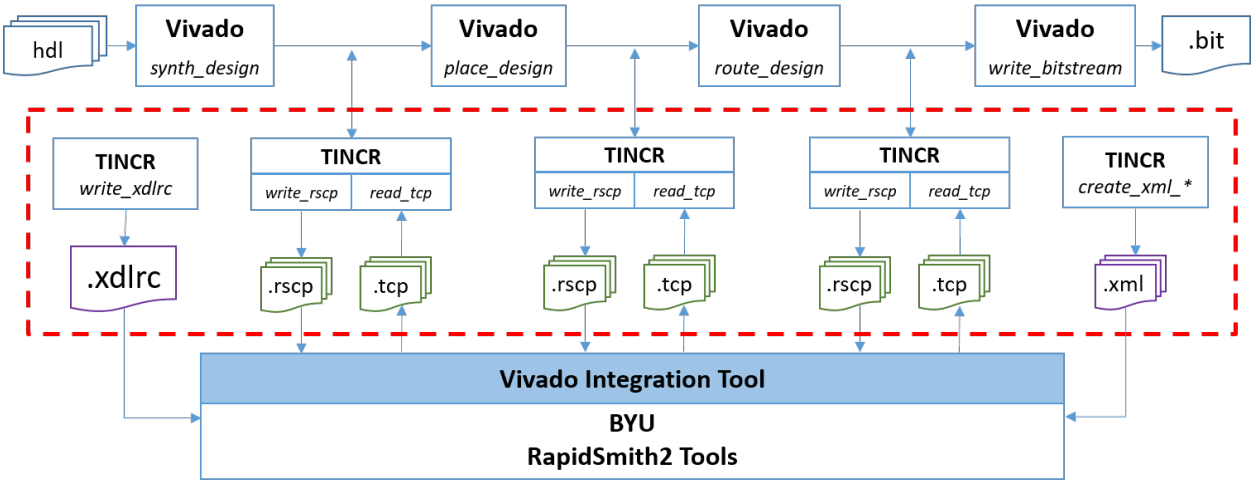


Figure 27: RapidSmith2 Design Flows

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 18.

Listing 18: How to import RSCP and export TCP files to and from RapidSmith2

```
1  // Loading a RapidSmith Checkpoint
2  VivadoCheckpoint vcp = VivadoInterface.loadRSCP("pathToCheckpoint.rscp");
3  CellDesign design = vcp.getDesign();
4  Device device = vcp.getDevice();
5  CellLibrary libCells = vcp.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::loadRSCP()`, as shown in Listing 19.

Listing 19: Importing a RSCP with additional information

```
1  // Loading a Tincr Checkpoint with additional info
2  VivadoCheckpoint vcp = VivadoInterface.loadRSCP("PathToCheckpoint.rscp", true);
3  Collection<BelRoutethrough> belRts = vcp.getRoutethroughObjects();
4  Collection<Bel> staticSources = vcp.getStaticSourceBels();
5  Map<BelPin, CellPin> belPinToCellPinMap = vcp.getBelPinToCellPinMap()
```

Line 10 of Listing 18 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]`

8.1 Import Notes

There are a few things to be aware of when a design is converted from a RSCP to a RapidSmith2 `CellDesign`.

- All VCC nets of the RSCP 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 static net in the design.
- 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 subsubsection 5.3.1. 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 `PseudoCellPin`s, are added to the global VCC/GND net, attached to the cell placed at the corresponding BEL, and then mapped to the BEL pin.
- 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). After design import, it is the user's responsibility to update the route status of the net based on which cell pins have been routed to.
- RSCPs include how the design is implemented in Vivado (i.e. if the design has been implemented "out-of-context"). After a RSCP has been imported, the design mode can be retrieved with the function call `CellDesign::getImplementationMode()`.

8.2 Export Notes

On design export, the structure of the original netlist is changed to support importing the TCP back into Vivado. It is important to understand that the TCP netlist generated from RapidSmith2 will be **structurally different**, but **functionally equivalent** to the netlist produced from Vivado.

9 Example Programs

A variety of example programs can be found in the `edu.byu.edu.rapidSmith.examples` package of the RapidSmith2 installation. They have been heavily commented to provide a means to learn the RapidSmith2 API by example. We believe this approach is better than reading through a block of text while trying to understand the data structures and what they do. There is a *README.txt* file in that directory to provide an overview of each example. The order that they appear in the *README.txt* is also the suggested learning order for beginners. In addition, the subsections below describe one or more built-in RapidSmith2 programs which you might find useful.

9.1 Sample Vivado Designs

To enable new users of RapidSmith2 to quickly start running the example programs, a small set of pre-compiled Vivado designs have been included in the distribution. They are located in the *exampleVivadoDesigns* directory of the repository, and consist of 3 designs:

- **add.rscp**: synthesized only
- **cordic.rscp**: synthesized and placed
- **count16.rscp**: synthesized, placed and routed

There is a sample design to demonstrate each possible tool flow between Vivado and RapidSmith2. Equivalent Vivado checkpoints files (.dcp) are also included in the same directory as *add.dcp*, *cordic.dcp*, and *count16.dcp* files. To open these checkpoints in Vivado, you can either (a) double click on the .dcp file in a file explorer, or (b) use the command `open_checkpoint` when a Vivado terminal is open. It is suggested that you have the equivalent Vivado designs open when going through the example programs listed below.

If you want to recompile the designs from scratch, the source code for each design has also been included in the same directory. The Tcl script called **compile.tcl** can be used for this purpose. Simply open Vivado in Tcl mode, and type the Tcl commands shown in Listing 20 to re-compile and implement one of the example designs. This will synthesize, place, and route a design and, from that compiled design, generate the *.rscp* directory and the *.dcp* file. For example programs that only explore the architecture, opening the device browser in Vivado can also be helpful.

Listing 20: Sample Tcl commands to run the Vivado compilation script

```
Vivado% cd <path to exampleVivadoDesigns directory>
Vivado% compile_hdl_to_checkpoint_files add
Vivado% close_project
```

9.2 DeviceBrowser

The DeviceBrowser is a GUI program located in the `edu.byu.ece.rapidSmith.device.browser` package. It lets you browse parts at the tile level, and is useful for becoming more familiar with FPGA architecture. As long as a valid device file exists, then the DeviceBrowser can operate (no design required). A screenshot from the DeviceBrowser can be seen in Figure 3. On the left, the user may choose the desired part by navigating the tree menu and double-clicking on the desired part name. This will load the part in the viewer pane on the right (the first available part is loaded at startup). The status bar in the bottom left displays which part is currently loaded. Also displayed is the name of the current tile which the mouse is over, highlighted by a yellow outline in the viewer pane. The user may navigate inside the viewer pane by using the mouse. By right-clicking and dragging the cursor, the user may pan. By using the scroll-wheel on the mouse, the user may zoom. If a scroll-wheel is unavailable, the user may zoom by clicking inside the viewer pane and pressing the minus(-) key to zoom out or the equals(=) key to zoom in.

The device browser also allows the user to follow the various connections found in the FPGA. By double clicking a wire in the wire list, the application will draw the connection on the tile array (as shown in Figure 28). By hovering the mouse pointer over the connection, the wire becomes red and a tooltip will appear describing the connection made by declaring the source tile and wire followed by an arrow and the destination tile and wire. By clicking on the wire, the application will redraw all the connections that can be made from the currently selected wire. By repeating this action, the user can follow connections and discover how the FPGA interconnect is laid out. Thanks to Chris Lavin for originally creating this application.

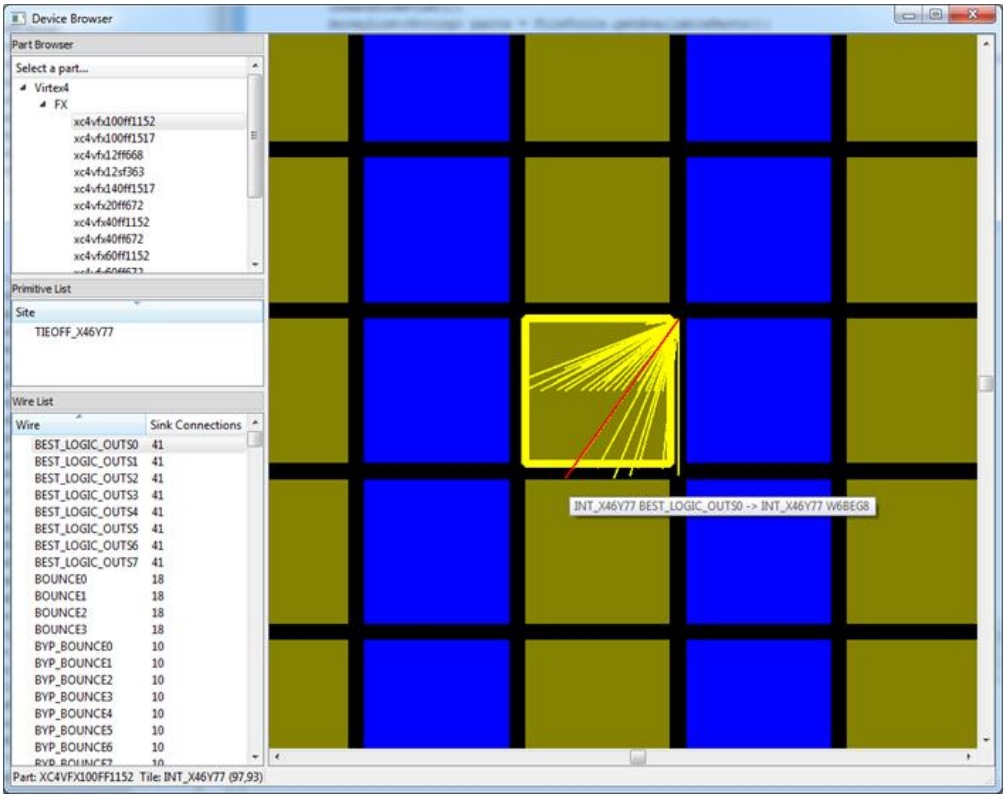


Figure 28: DeviceBrowser Screen Shot Showing Wire Connections

9.3 DeviceAnalyzer

The DeviceAnalyzer is designed as a simple getting started program and demonstrates how to use some of the Device data structures in RapidSmith2. This includes how to query for and print tiles in a device, how to use wires and wire connections, and other useful device functions.

9.4 ImportExportExample

The ImportExportExample demonstrates how to load a RapidSmith Checkpoint (RSCP) into RapidSmith2, and how to export a RapidSmith2 design back into a Tincr Checkpoint (TCP). This is a very important step in passing digital designs back and forth between Vivado and RapidSmith2.

9.5 DesignAnalyzer

The DesignAnalyzer loads a RapidSmith Checkpoint into RapidSmith2, walks the design data structures, and prints what it finds as it goes in a readable format. As such, it provides a nice example of a number of things which would be useful for getting started with RapidSmith2 including:

- How to enumerate the Cells in a design, determine and print their placement information, and determine and print their properties.
- How to enumerate the logical nets in a design and print out their source and sink pins.
- How to traverse and print out the physical route for a logical net (if it is routed)

9.6 CreateDesignExample

The CreateDesignExample program builds a RapidSmith2 netlist from scratch (using `Cell`s and `CellNet`s) and then places the design. While this is certainly not recommended for substantial designs, it does demonstrate how to do the following useful tasks in RapidSmith2:

- Create new `Cell`s and add them to an existing netlist
- Create new `CellNet`s, and connect them to `CellPin`s
- Modify the properties on `Cell`s
- Place `Cell`s onto a `Bel`s
- Find compatible `Bel` placement for a given `Cell`

9.7 Other Test Programs

The programs introduced in this section are designed for beginners of RapidSmith2. Once you start becoming more comfortable with RapidSmith2 and its data structures, there are several other more advanced examples. These examples include the **HandRouter**, **AStarRouter**, and **SimulatedAnnealingPlacer** programs. See the README.txt file for more information about each of these examples.

10 Installing New Device Files

The device files included with the RapidSmith2 installation (listed in subsection 4.4) have been well-tested, and are great starting points for new users. If you are new to RapidSmith2, it is *strongly encouraged* to start with these existing device files. However, RapidSmith2 also supports installing new devices for parts not listed in subsection 4.4. To create a new RapidSmith2 device file, four `Tincr` intermediate files are required:

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

The contents and format of these files are described in great detail in Appendix A, Appendix B, Appendix C, and section 5.4 respectively, and will not be described here. For those that are curious about what each file represents, refer to the listed appedices. 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.

10.1 Creating New Device Files for Supported Families

Section 4.4 gives a list of currently supported families in RapidSmith2. If the device to install is **not** within a supported family, see subsection 10.2 for how to add support for a new family in RapidSmith2. Otherwise, a new device file can be added in five easy steps:

1. Open Vivado in Tcl mode, and execute the `Tincr` command `[::tincr::write_xdlrc]`. An example usage is shown in Listing 21 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), and so 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 21: 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 generated *deviceInfo.xml* file to the directory *rapidSmithPath/devices/family*, where "rapidSmithPath" is the path to your RapidSmith2 installation and "family" is the corresponding Vivado family name (i.e. artix7, kintex7, virtex7, zynq, kintexu, virtexu, kintexuplus, virtexuplus, etc.)..
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 22. 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 22: 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 23 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.

5. The final step is to create a *cellLibrary.xml* file, which details all Xilinx primitives that can target the device. subsubsection 5.4.1 demonstrates how to generate a new cell library. Copy the generated cell library to the corresponding family folder of the device and rename it to "cellLibrary.xml."

Listing 23: 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 24 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 24: Generated RapidSmith2 device files

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

10.2 Supporting New Device Families

Vivado 2016.2 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. **NOTE: In general, you should not be adding support for a new family as a general user of RapidSmith2. Instead, create an issue at the RapidSmith2 GitHub repository and the maintainers of RapidSmith2 in the BYU CCL lab will add support. The following instructions are intended for BYU students only**.

1. Create the primitive definitions of the family using VSRT. The VSRT user guide is given in located at https://github.com/byuccl/RapidSmith2/tree/master/doc.

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 (shown in parentheses above).

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 25 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 25: 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 10.3 and subsection 10.4 respectively.
5. Copy the generated *familyInfo.xml* file to the directory *rapidSmithPath/devices/family*, where "rapidSmithPath" is the path to your RapidSmith2 installation and "family" is the corresponding Vivado family name. Make sure the family info is named "familyInfo.xml". For example, if I generated a family info for the artix7 part "xc7a100tcsg324", I would copy the family info into the *devices/artix7* directory.
6. Follow the steps laid out in subsection 10.1 to generate RapidSmith2 device files.
7. Run the Family Builder in RapidSmith2 (an example usage is shown in Listing 23). 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 23 will generate an `Artix7.java` file which can be used to find site and tile types as shown in Listing 26. 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 27 for reference.

Listing 26: How to access SiteTypes and TileTypes in RapidSmith2

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

Listing 27: Device classifications example

```
/* ------ CLASSIFICATIONS GO HERE ------ */
     // Tile Types
      _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);

     // Site Types
      _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.

10.3 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 35). 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 28 can be used to select a specific site type in Vivado to view its alternate types.

Listing 28: 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 29 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 29 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 29). Table 3 gives a list of all alternate types that rename pins for Artix7 devices.

Listing 29: 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>
```

Table 3: Artix7 Alternate Pin Mappings

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

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.

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 XML.

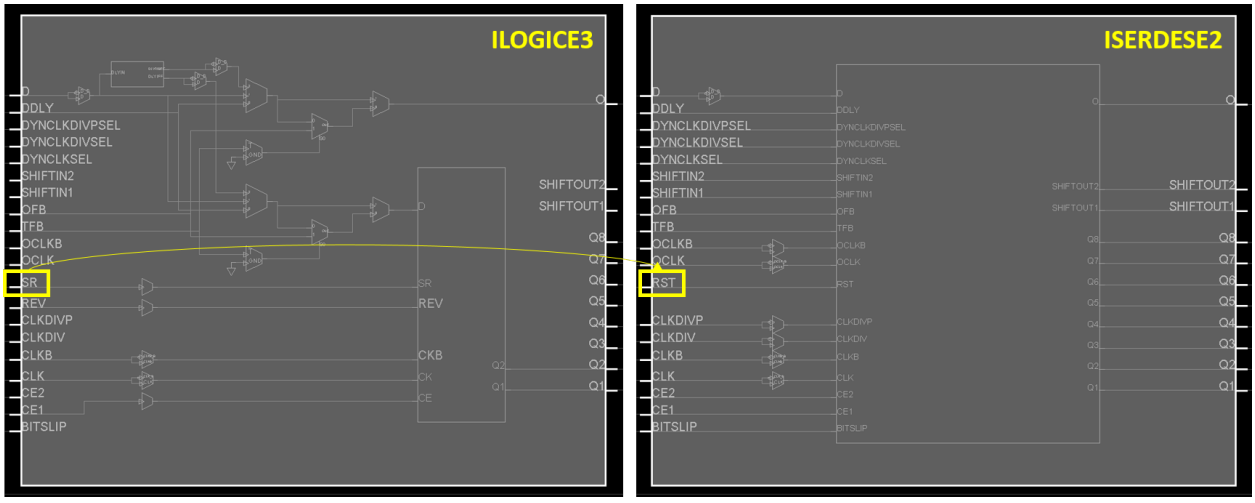


Figure 29: Example Alternate Site Pin Renaming

10.4 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 30, 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 30: 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>
```

11 Bitstreams in RapidSmith2

In the original RapidSmith, bitstreams can be parsed, manipulated, and exported for Virtex 4, Virtex 5 and Virtex 6 Xilinx FPGA families. Because of the proprietary nature of Xilinx bitstreams, RapidSmith provided only documented functionality when working with bitstreams (and was limited mainly to manipulation at the frame level including helping to assemble sequences of configuration commands which are interpreted by the FPGA configuration controller circuitry). While this has proven valuable to many researchers, it does not provide the ability to create your own bitstream from scratch because it does not provide the specific meaning of each bit in a bitstream.

If you desire to use RapidSmith's bitstream manipulation features, you should download and work with RapidSmith instead of RapidSmith2 (the RapidSmith bitstream packages have been removed from RapidSmith2). If you do so, note that RapidSmith's bitstream packages have not been tested beyond Virtex 6. The authors would be interested in upgrading RapidSmith's bitstream functionality to device families beyond Virtex 6 if users create it and are willing to contribute it to us for inclusion.

12 License

RapidSmith2 is released under GPL version 3 with the following license:

```
BYU RapidSmith Tools

Copyright (c) 2010-2016 Brigham Young University

BYU RapidSmith Tools is free software: you may redistribute it
and/or modify it under the terms of the GNU General Public License
as published by the Free Software Foundation, either version 2 of
the License, or (at your option) any later version.

BYU RapidSmith Tools is distributed in the hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the implied warranty
of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
General Public License for more details.

A copy of the GNU General Public License is included with the BYU
RapidSmith Tools. It can be found at doc/gpl2.txt. You may also get
a copy of the license at http://www.gnu.org/licenses/.
```

13 Included Dependency Projects

RapidSmith2 includes the Caucho Technology Hessian implementation which is distributed under the Apache License. A copy of this license is included in the doc directory in the file APACHE2-LICENSE.txt. This license is also available for download at:
http://www.apache.org/licenses/LICENSE-2.0

The source for the Caucho Technology Hessian implementation is available at:
http://hessian.caucho.com

RapidSmith2 also includes the Qt Jambi project jars for Windows, Linux and Mac OS X. Qt Jambi is distributed under the LGPL GPL3 license and copies of this license and exception are also available in the /doc directory in files LICENSE.GPL3.TXT and LICENSE.LGPL.TXT respectively. These licenses can also be downloaded at:
http://www.gnu.org/licenses/licenses.html

Source for the Qt Jambi project is available at:
http://qtjambi.org/downloads, and
https://sourceforge.net/projects/qtjambi/files/

RapidSmith2 also includes the JOpt Simple option parser which is released under the open source MIT License which can be found in this directory in the file MIT_LICENSE.TXT. A copy of this license can also be found at:
http://www.opensource.org/licenses/mit-license.php

A copy of the source for JOpt Simple can also be downloaded at:
http://jopt-simple.sourceforge.net/download.html

RapidSmith2 also includes the JDOM jars. JDOM is available under an Apache-style open source license, with the acknowledgment clause removed. This license is among the least restrictive license available, enabling developers to use JDOM in creating new products without requiring them to release their own products as open source. This is the license model used by the Apache Project, which created the Apache server. The license is available at the top of every source file and in LICENSE.txt in the root of the JDOM distribution.

The user is responsible for providing copies of these licenses and making available the source code of these projects when redistributing these jars.

A XDLRC Files and Syntax

In general, users of RapidSmith2 do not need to understand the syntax of XDLRC files to create CAD tools in RapidSmith2. The syntax is introduced here for those who are interested, and for those who want to modify the XDLRC parser in some way. If these don't apply to you, then go ahead and skip this section. XDLRC files are textual descriptions of Xilinx FPGA devices and can be very verbose (which is why they get so large). This section highlights the main parts of an XDLRC file with accompanying images. As you will see, much of the terminology is the same as subsection 4.1.

Tiles

```
# Example of an XDLRC tile declaration
     (tile 1 14 CLB_X6Y63 CLB 4
…
          (tile_summary CLB_X6Y63 CLB 122 403 148)
     )
```

Figure 30: Tile syntax in XDLRC files

A tile in an XDLRC file corresponds to the same thing as the `Tile` described in subsection 4.1. Each tile is declared with a "(tile" directive as shown above followed by the unique row and column index of where the tile fits into the grid of tiles found on the FPGA. The tile declaration also contains a name followed by a type with the final number being the number of primitive sites found within the tile. The tile ends with a "tile_summary" statement repeating the name and type with some other numbered statistics. Tiles can contain three different sub components, primitive sites, wires, and PIPs.

Primitive Sites

```
# Example of an XDLRC primitive site declaration
          (primitive_site SLICE_X9Y127 SLICEL internal 27
               (pinwire BX input BX_PINWIRE3)
               (pinwire BY input BY_PINWIRE3)
               (pinwire CE input CE_PINWIRE3)
               …
               (pinwire XMUX output XMUX_PINWIRE3)
          )
```

Figure 31: Primitive site syntax in XDLRC files

Primitive site declarations in XDLRC files contain a list of pinwires which describe the name and direction of pins on the primitive site. The first pinwire declared in the example above is the BX input pin which is the internal name to the SLICEL primitive site. Pinwires have an external name as well to differentiate the multiple primitive sites that may be present in the same tile. In this case, BX of SLICE_X9Y127 has the external name BX_PINWIRE3. In RapidSmith2, only the first pin name (i.e. BX above) is used.

Wire

A wire as declared in XDLRC is a routing resource that exists in the tile that may have zero or more connections leaving the tile. In the example above, the wire "E2BEG0" connects to 5 neighboring tiles. These connections (denoted by "conn") are described using the unique tile name and wire name of that tile to denote connectivity. The connections are not programmable, but hard wired into the FPGA. Wire portions of the XDLRC file are included in the definition of every tile (even if the same tile type has already been printed), which has a big impact on the final size of XDLRC files. How RapidSmith2 handles wire duplication is described in subsubsection 4.2.2. The `WireConnection` objects that are created from this part of the XDLRC are described in subsection 7.1.

```
# Example of an XDLRC wire declaration
          (wire E2BEG0 5
               (conn CLB_X7Y63 CLB_E2BEG0)
               (conn INT_X8Y63 E2MID0)
               (conn CLB_X8Y63 CLB_E2MID0)
               (conn INT_X9Y63 E2END0)
               (conn INT_X9Y62 E2END_S0)
          )
```

Figure 32: Wire syntax in XDLRC files

PIP

```
# Example of an XDLRC PIP declaration
          (pip INT_X7Y63 BEST_LOGIC_OUTS0 -> BYP_INT_B5)
```

Figure 33: PIP syntax in XDLRC files

A PIP (programmable interconnect point) is a possible connection that can be made between two wires. In the example above, the PIP is declared in the tile and repeats the tile name for reference. It specifies two wires by name that both exist in that same tile ("BEST_LOGIC_OUTS0" and "BYP_INT_B5") and declares that the wire "BEST_LOGIC_OUTS0" can drive the wire "BYP_INT_B5". A collection of these PIPs in a net define how a net is routed and is consistent with saying that those PIPs are turned on. subsection 7.1 describes in detail how PIPs are represented in RapidSmith2.

Primitive Definitions

The Primitive Definition portion of an XDLRC file textually describes the components found within a `Primitive Site` (a SLICEL for example) and how they are connected. This includes:

- BELs
- Site Pins

```
(primitive_def BUFHCE 3 7
       (pin CE CE input)
       (pin I I input)
       (pin O O output)
       (element CE 1
              (pin CE output)
              (conn CE CE ==> CEINV CE)
              (conn CE CE ==> CEINV CE_B)
       )
       (element I 1
              (pin I output)
              (conn I I ==> BUFHCE I)
       )
       (element O 1
              (pin O input)
              (conn O O <== BUFHCE O)
       )
       (element CEINV 3
              (pin OUT output)
              (pin CE input)
              (pin CE_B 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)
       )
       (element BUFHCE 3 # BEL
              (pin CE input)
              (pin I input)
              (pin O output)
              (conn BUFHCE CE <== CEINV OUT)
              (conn BUFHCE I <== I I)
              (conn BUFHCE O ==> O O)
       )
       (element CE_TYPE 0
              (cfg SYNC ASYNC)
       )
       (element INIT_OUT 0
              (cfg 0 1)
       )
)
```

Figure 34: Primitive Def sections of XDLRC files

- Site Pips (Routing Muxes)
- Configuration options (both site and BEL)
- Site Routethroughs (configurable connections from a site input pin to a site output pin)
- Site Wire Connections

An example of a complete primitive definition file of type BUFHCE can be seen in Figure 34. The sub-site data structures in RapidSmith2 (`Bels`, `SiteWires`, etc.) are built by parsing this section of the XDLRC file. For a more detailed description of primitive definitions, view the VSRT User Guide in the RapidSmith2 repository.

B 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 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 each component of a family info XML and why they may be useful for external CAD tools.

B.1 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 35. 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. Listing 31 shows how alternate types are represented in a family info XML. Pinmap tags are included for alternate pin names changing as decribed in subsection 10.3.

Listing 31: Example Alternate Type XML

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

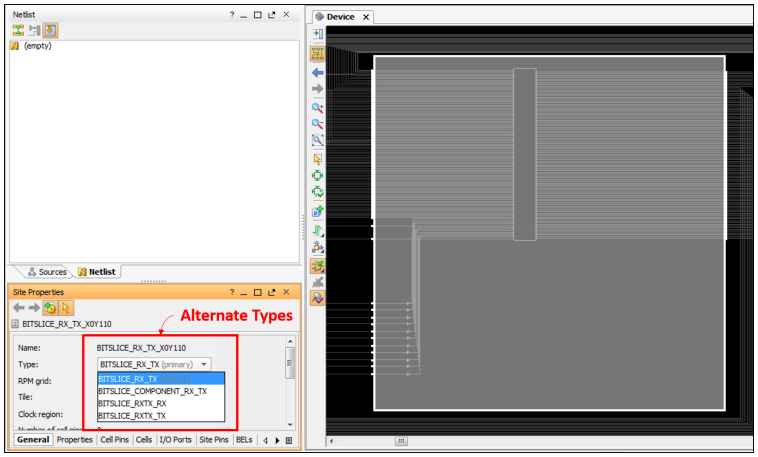


Figure 35: BITSLICE_RX_TX Alternate Types

B.2 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 36, 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. 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.

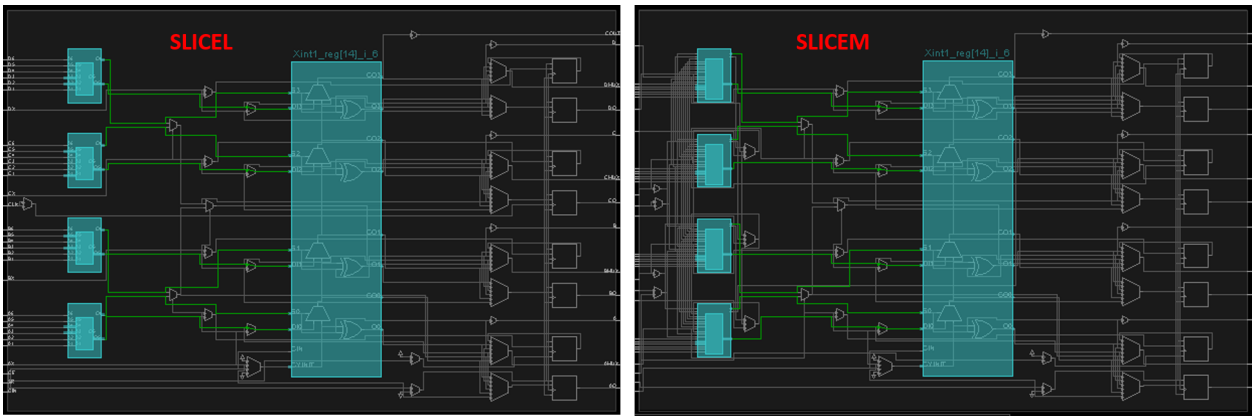


Figure 36: A group of cells placed on a SLICEL site (left) and a SLICEM site (right).

A visualization of compatibility is also given in Figure 37. In this case, "Site 1" and "Site 2" are both compatible to each other since the same set of cells can be placed onto both sites. "Site 4" is compatible with "Site 3" because each cell that can be placed on "Site 4" can also be placed on "Site 3." "Site 3", however, is not compatible with "Site 4" because cell `Z` cannot be placed on "Site 4."

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

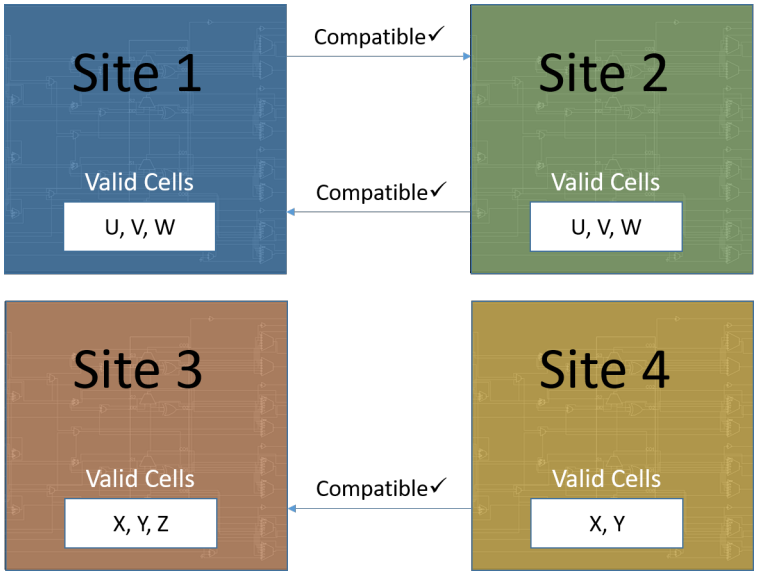


Figure 37: Compatibility Testing for Single-BEL Sites

site type for each group of cells, lowering the quality of results. Listing 32 shows example XML for compatible types in a family info file.

Listing 32: Compatible Type XML for SLICEL

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

B.3 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 33 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 D.1.8.

Listing 33: 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>
```

B.4 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 34 shows how site PIP corrections are represented in a family info file.

Listing 34: Example Site PIP Corrections

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

RapidSmith2 decomposes a site PIP into its individual PIPs as shown in Figure 38. 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".

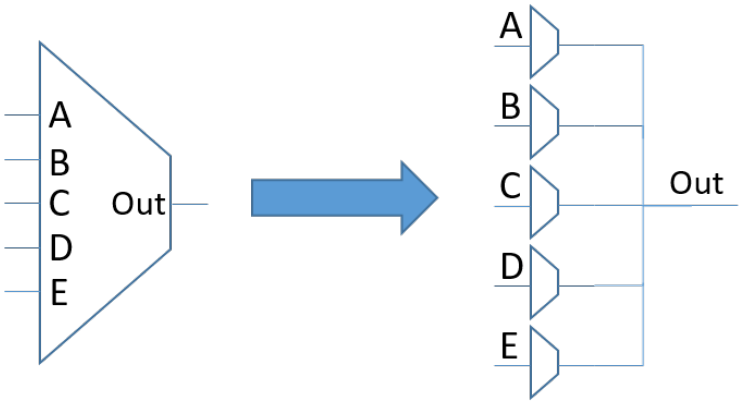


Figure 38: Site PIP Decomposition

B.5 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 RapidSmith2. The direction of a BEL pin in Vivado can be determined with the Tcl command `[get_property DIRECTION $belpin]`. Listing 35 shows how pin direction corrections are represented in XML.

Listing 35: Example Pin Direction Correction

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

C DeviceInfo 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 36. 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 `Tincr` command `[tincr::create_xml_device_info]`. This command is fully automated, and requires no hand edits.

Listing 36: 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>
```

D VDI Checkpoint Formats

Before reading this appendix, it is encouraged for the reader to first read Thomas Townsend's master thesis located for download at: http://scholarsarchive.byu.edu/etd/6492/. This document gives a very detailed introduction to the **Vivado Design Interface**, the interface used to extract device and design information out of Vivado.

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 39. The remainder of this chapter describes each external design representation, and the required Tcl commands to generate each. Section D.1 introduces RSCPs and subsection D.2 introduces TCPs. Section D.3 lists the variety of Tcl challenges associated with using the Vivado Tcl interface to represent designs externally. Section D.4 includes a discussion on why two different external design formats are necessary. The purpose of this appendix is to document the format of RSCPs and TCPs so that maintainers of RapidSmith2 will understand how to parse/generate these files. These checkpoint formats may change with future versions of Vivado. **General users of RapidSmith2 can ignore this appendix**.

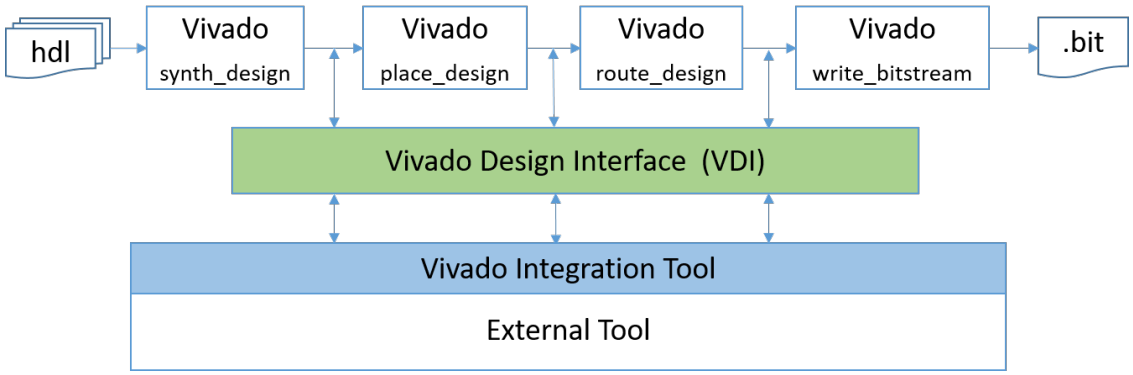


Figure 39: Vivado Design Interface (VDI) Design Flows

D.1 Design Export (RSCP Format)

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.

D.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 37 shows an example *design.info* file.

Listing 37: Sample design.info

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

D.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 38. External tools can use any open-source EDIF parser (such as BYU EDIF tools) to translate the EDIF into their own design representation to perform netlist modifications.

Listing 38: 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_lUT (viewref netlist (cellref LUT3 (libref hdi_primitives)))
          (property INIT (string "8hE8"))
        )
        ...
```

D.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 39, and is essentially a list of Tcl commands.

Listing 39: 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]}]
```

D.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, 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 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.

D.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 40. 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 40 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 correct. Cell pin mappings contained in the cell library 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 40 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 40 demonstrates the order in which the above tokens appear in the file. Figure 40 shows a visual representation of how cells are placed onto BELs in Vivado.

Listing 40: 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
```

```
    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
    ...
```

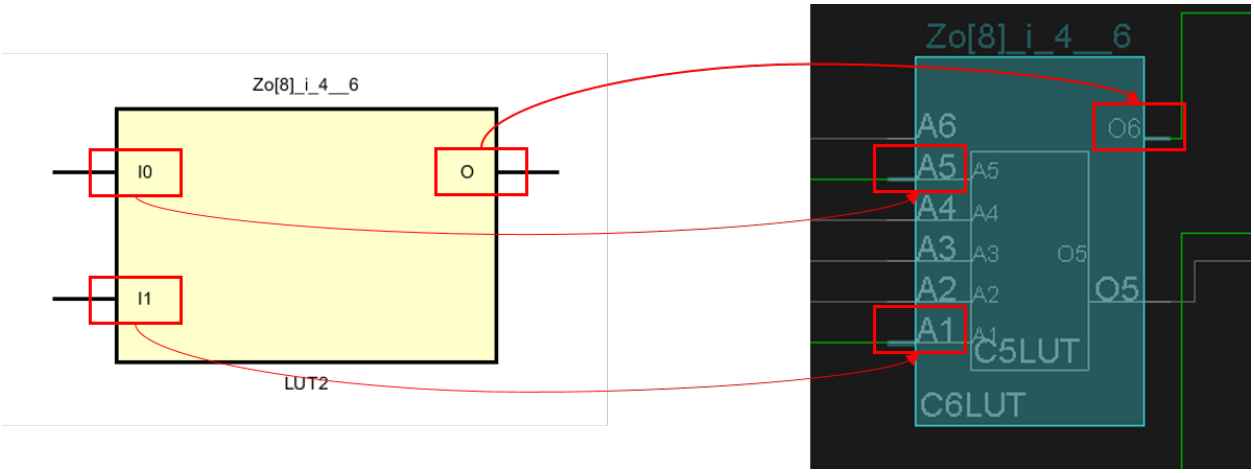


Figure 40: Example Cell Placement

D.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 41. 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 41: 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]
```

```
9     INTERSITE AngStep1[0] SLICE_X2Y82/AX SLICE_X2Y87/AQ SLICE_X2Y84/A1 ...
10    ROUTE AngStep1[0] INT_X28Y26/INT.LOGIC_OUTS_W1->>SDNDSW_W_0_FTS ...
11    ...
12    VCC INT_L_X2Y107/INT_L.VCC_WIRE->>IMUX_L42 ...
13    START_WIRES INT_L_X2Y107/VCC_WIRE INT_L_X2Y106/VCC_WIRE ...
14    GND INT_R_X3Y106/INT_R.GND_WIRE->>GFAN1 ...
15    START_WIRES INT_R_X3Y106/GND_WIRE INT_L_X4Y106/GND_WIRE ...
```

D.1.7 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 41 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 41. External tools can use the `SITE_PIPS` token within a *routing.rsc* file to reconstruct the internal routing of each site.

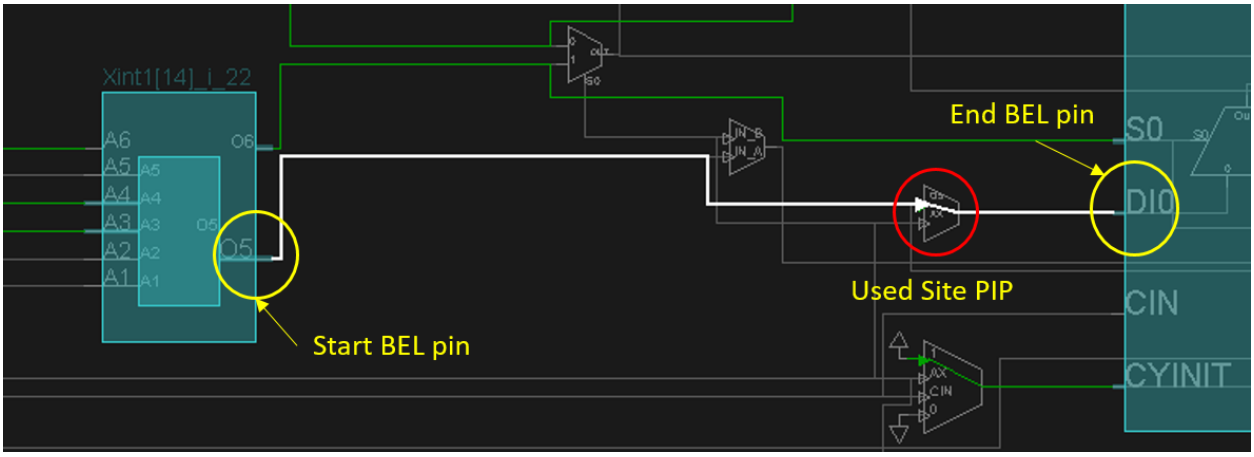


Figure 41: Site PIP Usage

D.1.8 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 42 shows two example routethrough LUTs in Vivado.

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

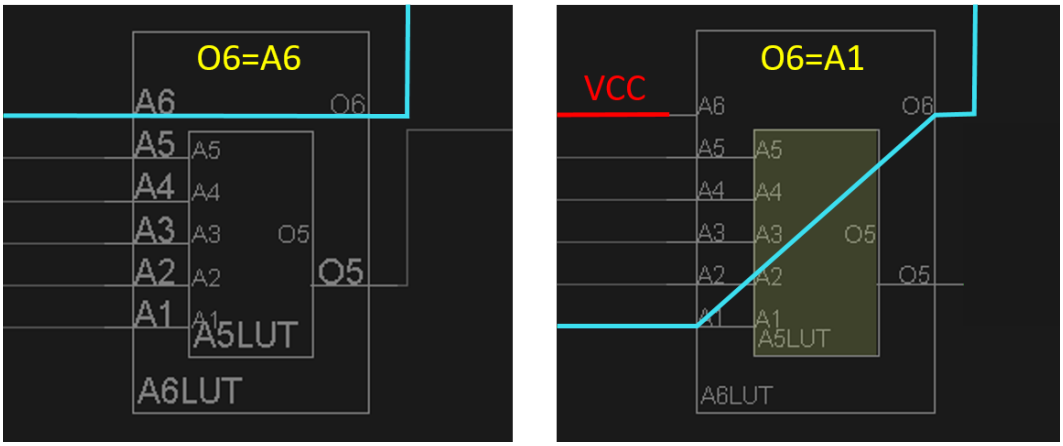


Figure 42: Two examples of LUTs configured as routethroughs in Vivado. The net highlighted in red represents VCC.

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 41. 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.

D.1.9 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 43. As the figure shows, permanent latches look very similar to LUT routethroughs described in the previous section.

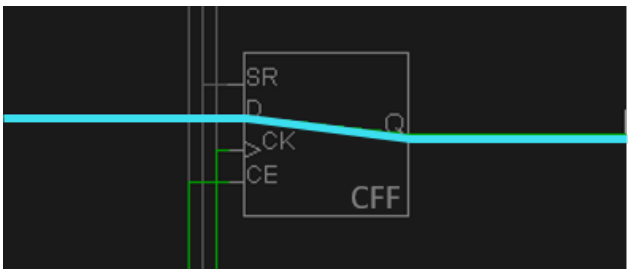


Figure 43: Flip-Flop BEL Configured as a Permanent Latch in Vivado

Because of this similarity, *routing.rsc* files treat permanent latches the same as LUT route-throughs. That is, each permanent latch is included alongside the `LUT_RTS` token as shown on line 6 of Listing 41. The string "SLICE_X40Y-96/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 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`.

D.1.10 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 44. 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`

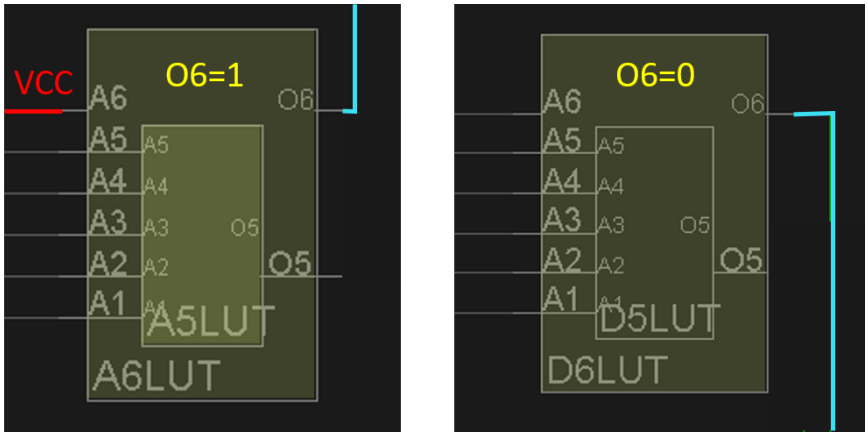


Figure 44: Two LUTs Configured as Static Sources in Vivado

respectively. This is shown on line 4 and 5 of Listing 41. 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.

D.1.11 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 45 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 41. 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}]`.

D.1.12 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 46. After a design has been placed in Vivado, each intersite net

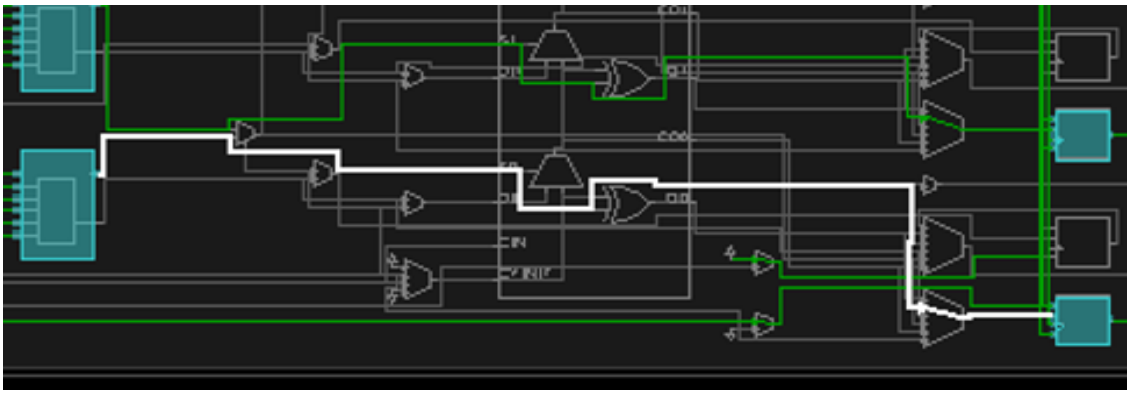


Figure 45: Vivado Intrasite Net (highlighted in white)

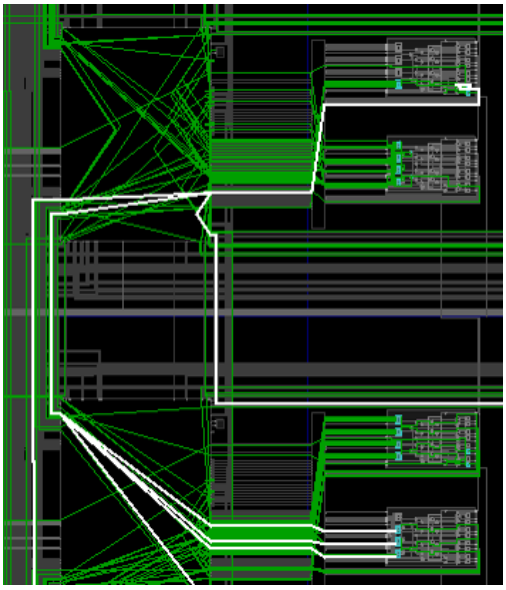


Figure 46: Vivado Intersite Net (highlighted in white)

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 41 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]`.

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 subsection D.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 41 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. The RapidSmith2 import code contains an algorithm to convert the pip list to a tree structure.

D.1.13 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 41 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.

D.2 Design Import (TCP Format)

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 subsection D.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 subsection D.3

D.2.1 Placement.xdc

Listing 42 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 42 shows how a top-level port is mapped to a FPGA package pin.

Listing 42: 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  ...
```

D.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 43. A directed routing string represents the tree structure of a physical route by using nested brackets ("{") to represent branching. As an example, take the hypothetical route shown in Figure 47. 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 43), or use only relative wire names (line 3 of Listing 43). 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.

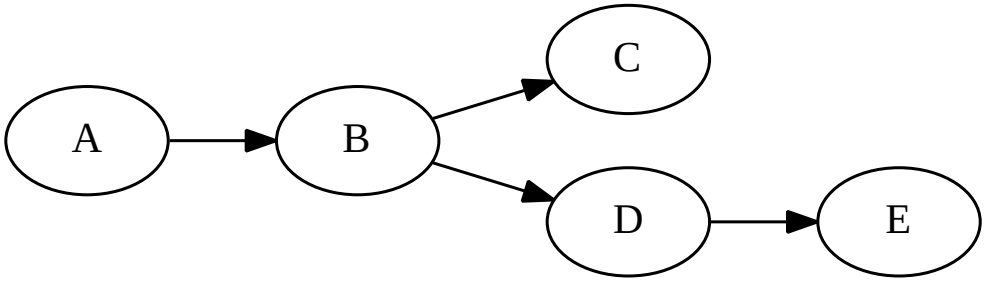


Figure 47: Sample Route (A, B, C, D, and E represent device wires)

Listing 43: Sample routing.xdc

```
...
set_property ROUTE { CLEL_R_X36Y8/CLE_L_SITE_0_COUT ... } [get_nets {Zo}]
set_property ROUTE { CLE_L_SITE_0_COUT ... } [get_nets {Zo}]
...
```

D.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.

D.3.1 Ambiguous ROUTE Strings

As described in subsubsection D.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 44 shows how Vivado distinguishes these two connections within `ROUTE` strings.

Listing 44: Ambiguous ROUTE string example

```
// Tile BRAM_L_X6Y165
{ ... BRAM_CASCOUT_ADDRBWRADDRU6 BRAM_ADDRBWRADDRU6 ... }
```

```
3    // Tile BRAM_L_X6Y175
4    { ... 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.

D.3.2 Alternate Site Pins

As discussed in subsection B.1, the Vivado Tcl command `[get_site_pins -of $site]` does not return the correct result for alternate sites on Series7 devices[5]. An example of this behavior is shown in Figure 48. 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.

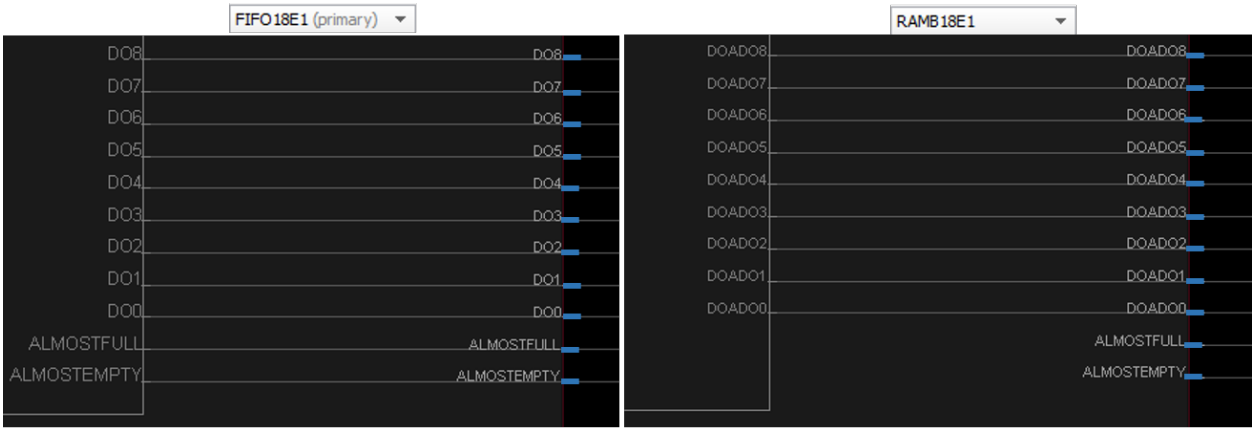


Figure 48: An example of [get_site_pins] returning incorrect results in Vivado

VDI fixes this issue with an assumption about single-BEL sites in Series7 devices. As can be seen in Figure 48, 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 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.

D.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 49. 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

[5] UltraScale devices always return the correct result

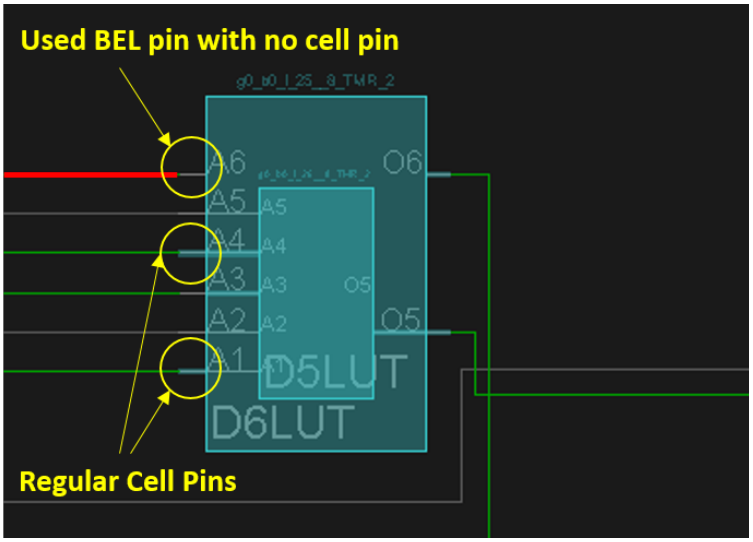


Figure 49: An example of VCC routing to an unused BEL pin (A6)

RSCPs. It is up to external tools to understand the placement configurations that require VCC and GND to be routed to individual BEL pins.

D.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 50 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` → LUT1, (2) `C` → FF1, and (3) `B` → 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` → FF2, (2) `A` → LUT1, and (3) `C` → 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

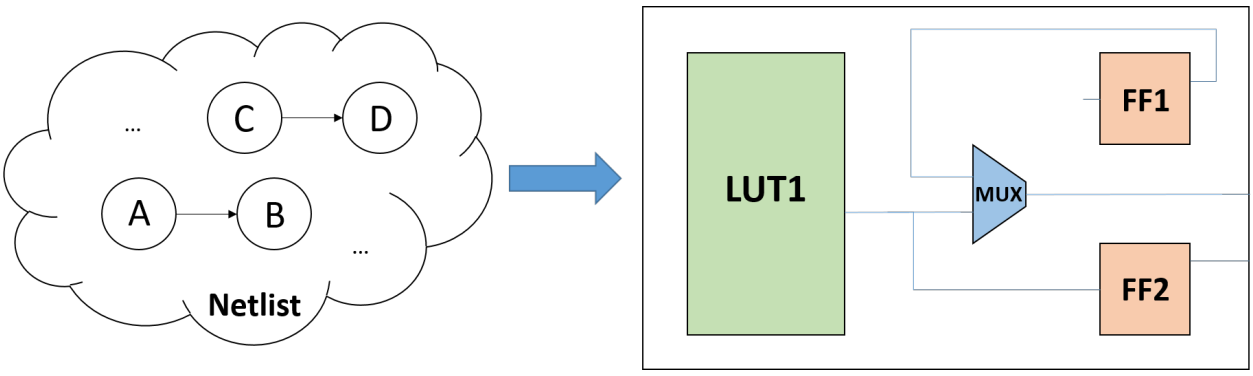


Figure 50: Routing Contention Example

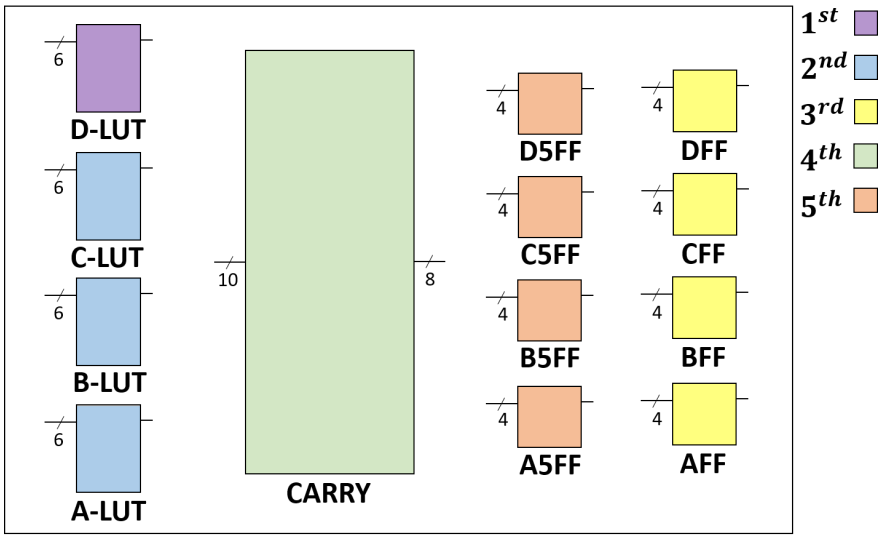


Figure 51: Required placement order for Series7 SLICE sites. The figure shows a simplified representation of a SLICEL site.

SLICE sites is as shown in Figure 51. 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 external tools to sort the cells of a design in the proper order shown in Figure 51 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.

D.3.5 Macro Placement

As described in subsubsection 5.2.5 fully-flattened Vivado netlists may include macro primitives. Section D.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.

D.3.6 LUT Routethroughs

As described in subsubsection D.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 52.

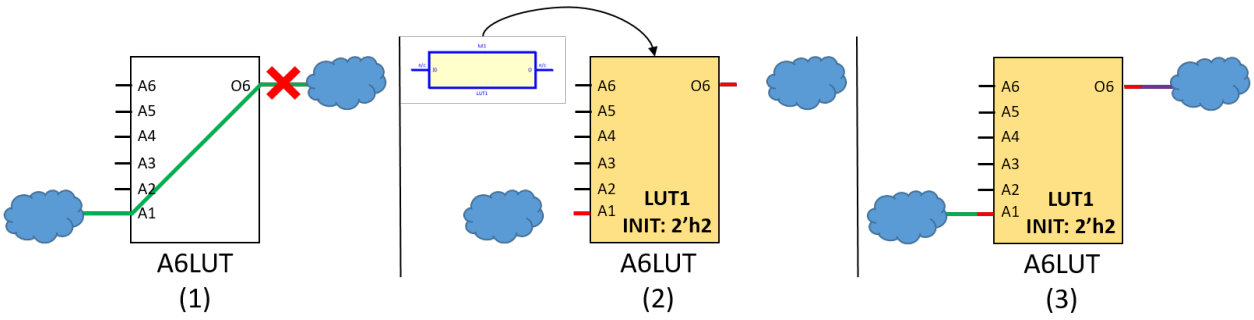


Figure 52: Visualization for how to replace a LUT routethrough with a LUT1 Xilinx cell.

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.

D.3.7 Routing Differential Pairs

Xilinx devices support differential input signals for design input. An example differential pairing is shown in Figure 53. The net highlighted in white connects the two differential signals 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 45 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 45: 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 }
```

D.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*

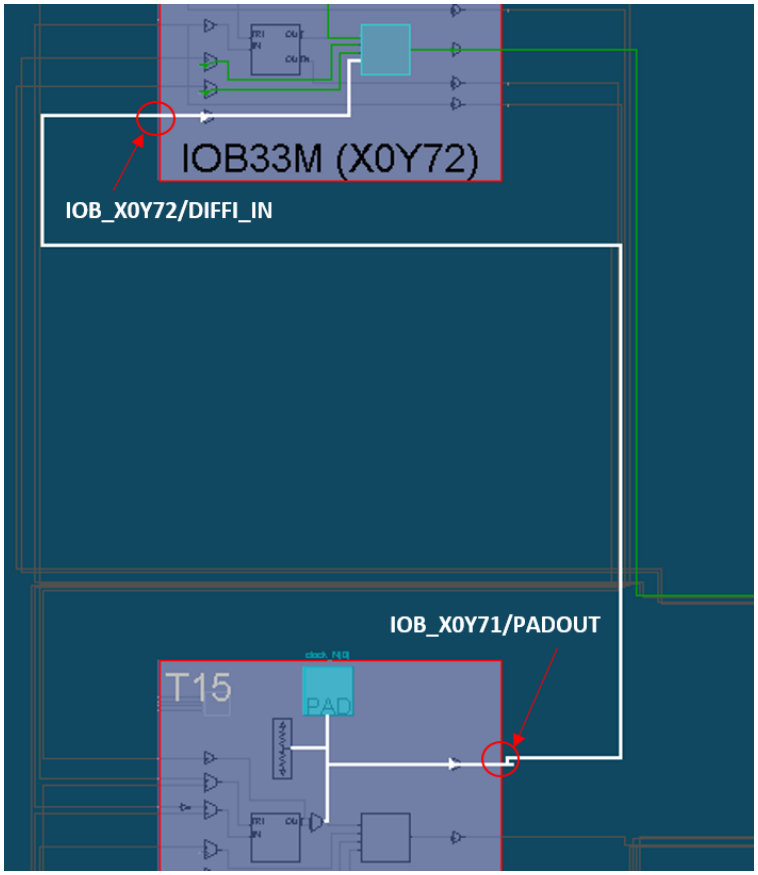


Figure 53: Differential Pair Net

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 subsubsection D.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 (subsubsection D.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 parse-able**, 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.