

RAPIDSMITH 2

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

Technical Report and Documentation

Brent Nelson, Travis Haroldsen, Thomas Townsend

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

Last Modified: August 14, 2017



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

Contents

1	Introduction	5
1.1	What is RapidSmith2?	5
1.2	Who Should Use RapidSmith2?	5
1.3	Why RapidSmith2?	5
1.4	Which Xilinx Parts does RapidSmith2 Support?	6
1.5	How is RapidSmith2 Different than VPR and VTR?	6
1.6	Why Java?	6
2	Vivado, RapidSmith2, and Tincr	7
2.1	RapidSmith vs. RapidSmith2	7
2.1.1	What Was The Original RapidSmith?	7
2.1.2	What is RapidSmith2?	7
2.1.3	Tincr: Integrating Custom CAD Tool Frameworks with the Xilinx Vivado Design Suite	7
2.1.4	RapidSmith2: A Framework for BEL-Level CAD Exploration on Xilinx FPGAs	7
2.1.5	Vivado and RapidSmith2 Integration	8
2.1.6	What is All This About XDL and XDLRC and How Does RapidSmith2 Fit Into That?	8
2.2	RapidSmith2 Usage Model and Structure	8
3	Getting Started	9
3.1	Installation	9
3.1.1	Requirements for Installation and Use	9
3.1.2	Steps for Installation	9
3.1.3	Alternative Installation and Use	10
3.1.4	Additional Notes for Mac OS X Installation	10
3.1.5	Running RapidSmith2 Programs	10
3.1.6	Testing Your Installation	11
3.2	Running Real Designs - An Overview	12
3.2.1	Synthesize, Place, and Route a Design in Vivado and Then Export It As a RSCP	12
3.2.2	Import the RSCP into RapidSmith2	13
3.2.3	Importing a Tincr Checkpoint Back into Vivado	13
4	Devices	14
4.1	Xilinx FPGA Architecture Overview	14
4.1.1	Device Hierarchy	14
4.1.2	Tiles	15
4.1.3	Sites	15
4.1.4	Wires and PIPs	15
4.2	Device Data Structures	17
4.2.1	Templates	18
4.2.2	WireEnumerator	18
4.2.3	TileWire and SiteWire	19
4.2.4	Package Pins	19
4.3	Loading a Device	19
4.4	Supported Device Files	20
5	Designs	21
5.1	Xilinx Netlist Structure	21
5.2	RapidSmith2 Netlist Data Structures	21
5.2.1	CellDesign	22
5.2.2	Cell	23
5.2.3	CellPin	23
5.2.4	CellNet	23
5.2.5	Macro Cells	26

5.2.6	PropertyList	27
5.2.7	Xdc Constraints	29
5.3	Vivado Design Considerations / Advanced Topics	29
5.3.1	Pseudo Cell Pins	29
5.3.2	LUT Routethroughs	30
5.3.3	Permanent Latches	30
5.3.4	Static Source LUTs	31
5.3.5	Site PIPs	31
5.4	The Cell Library	32
5.4.1	Generating A New Cell Library	32
5.4.2	Adding Custom Macros to a Cell Library	33
6	Placement	35
7	Routing	37
7.1	Wires and Wire Connections	37
7.2	Traversing Wire Objects	37
7.3	Other Types of Connections	38
7.4	RouteTrees	39
7.5	Three Part Routing	41
7.6	Routing in Vivado	43
7.7	Intrasite Routing	43
8	Design Import/Export	44
8.1	Import Notes	45
8.2	Export Notes	45
9	Example Programs	46
9.1	Sample Vivado Designs	46
9.2	DeviceBrowser	46
9.3	DeviceAnalyzer	47
9.4	ImportExportExample	47
9.5	DesignAnalyzer	47
9.6	CreateDesignExample	48
9.7	Other Test Programs	48
10	Installing New Device Files	49
10.1	Creating New Device Files for Supported Families	49
10.2	Supporting New Device Families	50
10.3	Series7 Family Info Hand Edits	52
10.4	UltraScale Family Info Hand Edits	54
11	Bitstreams in RapidSmith2	55
12	License	56
13	Included Dependency Projects	57
A	XDLRC Files and Syntax	58
B	Family Info XML	61
B.1	Alternate Types	61
B.2	Compatible Types	62
B.3	BEL Routethroughs	63
B.4	Site PIP Corrections	63
B.5	Pin Direction Corrections	64

C DeviceInfo Info XML	65
D RSCP and TCP Format	66

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. RapidSmith 2 (abbreviated RapidSmith2 hereafter) 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 subsection 2.1.3) 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 Which Xilinx Parts does RapidSmith2 Support?

As of the writing of this document, Artix 7 has been tested the most and is currently supported in all forms and applications. In addition, an Ultrascale device file was created and demonstrated as a part of Brad White's MS work to show that it is possible. At some point, Ultrascale should be fully supported.¹

As will be seen later, to generate additional device files for additional parts within a supported family is relatively straightforward and can be done by any user. New families can also be supported but this requires a bit more work. As time goes on the process will become simpler — that is one of the goals for RapidSmith2 moving forward.

1.5 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.

1.6 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.

¹ An XDL-based import/export capability has also been created and used with Virtex 6 devices as a part of Travis Haroldsen's PhD work but that path is not being released, documented, or supported.

2 Vivado, RapidSmith2, and Tincr

2.1 RapidSmith vs. RapidSmith2

2.1.1 What Was The Original RapidSmith?

The original RapidSmith was 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/NCD, 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. In this regard, RapidSmith did not include any proprietary information about Xilinx FPGAs that is not publicly available.

RapidSmith continues to be functional and is still available at the SourceForge.net website. There, you will find documentation, installation instructions, the RapidSmith code base, and a collection of demo programs based on it.

2.1.2 What is RapidSmith2?

With the announced end of ISE (with the Virtex7 family of parts being the last family to be supported by ISE), there was no path forward to newer parts using RapidSmith. This is because XDL is not available with Vivado. With Vivado, however, Xilinx has provided an extensive Tcl scripting capability which initially looked as if it could provide a similar capability to that provided by XDL in terms of accessing both Vivado's design and device data and in terms of creating and modifying Vivado designs. However, as described above, Vivado's Tcl is limited by speed and memory challenges. The development of RapidSmith2 consisted of three parts.

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

In the first part, the Vivado Tcl capability was investigated to ensure that, indeed, it did provide the needed ability to access design and device data and export that to external tools such as RapidSmith. This resulted in the Tincr project, led by Brad White as a part of his MS work at BYU, with Thomas Townsend making additions as a part of his research.

Tincr is a Tcl-based library of routines which (a) provide a variety of functions to simply make working with Vivado via Tcl easier, (b) provide a way to export all the data associated with a Vivado design into what is called a Tincr Checkpoint (TCP), (c) provide a way to reimport Tincr Checkpoints back into Vivado, and (d) access device data from Vivado and output that data in the form of XDLRC files (these are the files which XDL used to describe devices and are necessary for RapidSmith and RapidSmith2 to understand the structure of and the resources available for use in a given Xilinx part). Tincr is available at <https://github.com/byucl/tincr>. Tincr is described in two publications:

B. White and B. Nelson, "Tincr A custom CAD tool framework for Vivado," 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14), Cancun, 2014, pp. 1-6, DOI: 10.1109/ReConFig.2014.7032560

White, Brad S., "Tincr: Integrating Custom CAD Tool Frameworks with the Xilinx Vivado Design Suite" (2014), BYU Scholars Archive, Paper 4338.
URL:<http://scholarsarchive.byu.edu/etd/4338>

2.1.4 RapidSmith2: A Framework for BEL-Level CAD Exploration on Xilinx FPGAs

The second part of the development of RapidSmith2 was to add a new layer of design representation to RapidSmith which more closely matches that of Vivado. This was done as a part of his PhD work by Travis Haroldsen at BYU. As of this writing, one paper on RapidSmith2 has appeared:

Travis Haroldsen, Brent Nelson, and Brad Hutchings, RapidSmith 2: A Framework for BEL-Level CAD Exploration on Xilinx FPGAs, Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 2015, Monterey CA, pp. 66-69, DOI: 10.1145/2684746.2689085.

2.1.5 Vivado and RapidSmith2 Integration

The third part of the development of RapidSmith2 was to create the ability to export designs from Vivado and into RapidSmith2 and, correspondingly, to import RapidSmith2 data back into Vivado. This was completed during 2016, largely by Thomas Townsend as an MS student at Brigham Young University. The initial public release of RapidSmith2 was made in January 2017 once that piece was in place.

2.1.6 What is All This About XDL and XDLRC and How Does RapidSmith2 Fit Into That?

The Xilinx ISE tools had the capability to export XDL and XDLRC files which RapidSmith used:

- An XDLRC file was a complete description of a given Xilinx FPGA, describing every tile, every switchbox, every wire segment, and every PIP in the part. RapidSmith was able to process this information and create a device representation for use in support of CAD tools such as placers and routers.
- An XDL file was a textual representation of an NCD file (a user design). It described the user design as a collection of `Instances` and `Nets`. `Instances` correspond to things like `SLICES`, `BRAMs`, `DSP48s`, and `IOBs`. `Instances` could be placed onto `Sites`. Additionally, `Nets` in XDL consisted of a list of `Pins` (their logical connections) and an optional list of `PIPs` (their physical routing connections).

In Vivado, however, designs are described as a collection of `Cells` where a `Cell` corresponds to things like `LUTs`, flip flops, etc. A `Cell` may be placed onto a `BEL` object such as an `ALUT` or a `BFF`. RapidSmith2 contains a new layer of hierarchy in its design and device descriptions where `Cells` and `BELs` are first-class objects and design manipulation is all done at the `Cell/BEL` level.

Also, Vivado `Nets` are described using directed routing strings rather than lists of `PIPs`. RapidSmith2 also contains a set of new classes to enable the representation and manipulation of `Nets` in a format compatible with these routing strings.

Thus, using RapidSmith2, design manipulation is now done at the level of `Cells` and `BELs` and importing/exporting designs to/from Vivado is now fully supported.

2.2 RapidSmith2 Usage Model and Structure

The usage model for RapidSmith2 is shown in Figure 1. As can be seen, a design can be exported from Vivado at multiple different points in the Vivado design flow. 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.

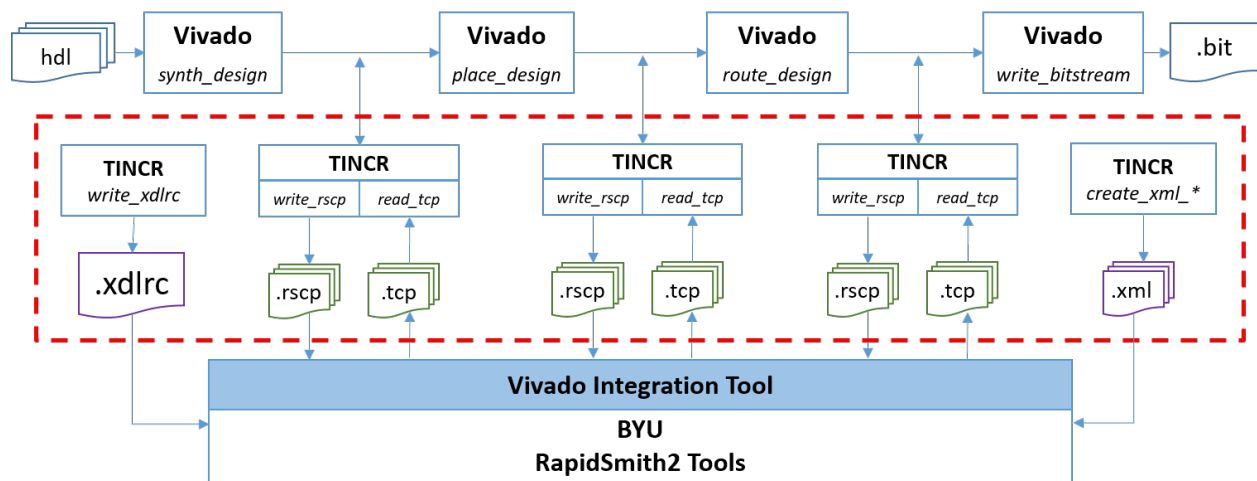


Figure 1: Vivado and RapidSmith2 Usage Model

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. 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 along 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 Alternative Installation and Use

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 2. You may initially be zoomed far in and might want to zoom out to see the entire chip layout.

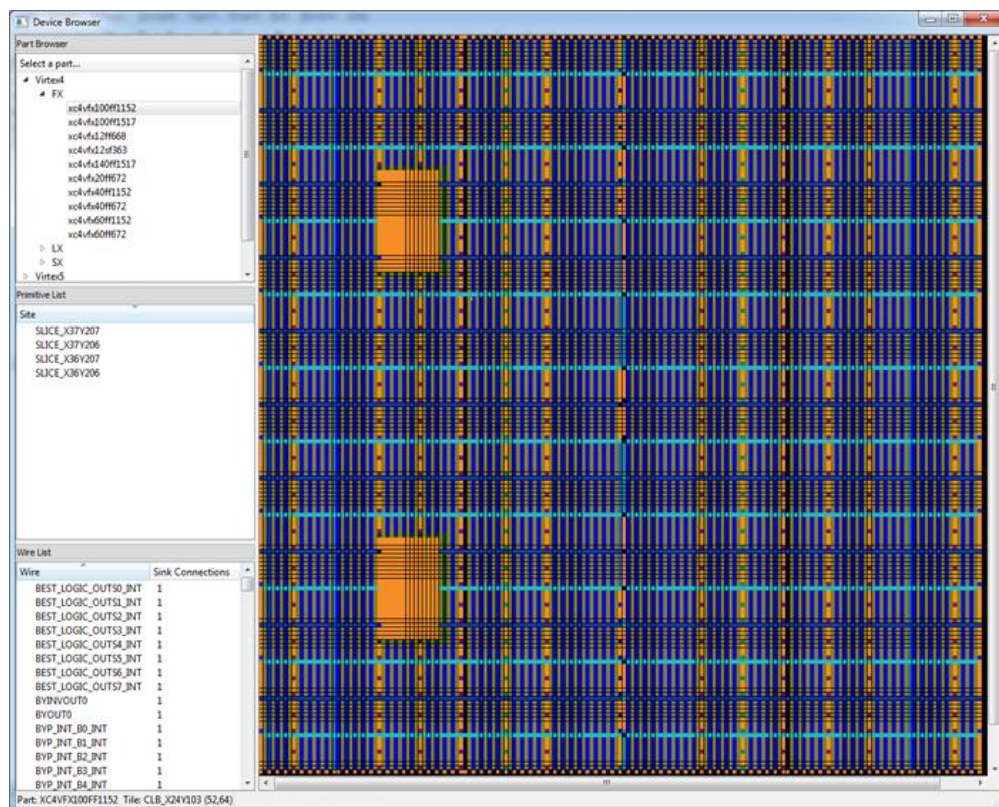


Figure 2: DeviceBrowser Sample Display

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-back-to 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 Synthesize, Place, and Route a Design in Vivado and Then Export It As a RSCP

If you desire, you may do this step any way you are used to using Vivado (with or without using the Vivado GUI for example). In this step, we will do it using the Vivado Tcl interface.

First, start up Vivado in command line mode. This can be done using the “Vivado Tcl Shell” program item or by executing “vivado -mode tcl”. Once the shell has started up, you may execute the following:

```
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. Importantly, you must flatten the design hierarchy when running the synthesis step as shown above. The result of executing this will be a new directory which contains the RapidSmith Checkpoint.

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.

Finally, note that a set of sample HDL designs are provided in the RapidSmith2 distribution (in *exampleVivado-Designs*) and which you could use for your learning. You can read about them as well as a script which implements a version of the above compilation steps in Section ?? of this report. Note that the compilation script described there expects a particular directory structure to hold your source files. Otherwise, the script implements essentially what is shown above.

3.2.2 Import the RSCP into RapidSmith2

Once you have a RSCP you are ready to create a Java program for RapidSmith2 that will be able to import that RSCP into RapidSmith2 so you can do something with it.

As described in Section 9 of this report (and Section 9.4 in particular) a sample import/export program is provided in the RapidSmith2 distribution which illustrates how to import and export designs from RapidSmith2. This program, when run, will read in a RapidSmith Checkpoint (RSCP), walk the resulting data structures and then prettyprint the design contents, and then write out the corresponding Tincr Checkpoint.

You should examine the code for that program to understand its operation and the subroutine calls that can be used to do those steps. The net result from running that program is to export a Tincr Checkpoint, stored in a directory with a “tcp” extension.

3.2.3 Importing a Tincr Checkpoint Back into Vivado

At this point you can import the resulting Tincr Checkpoint back into Vivado. Assuming the original RSCP was “add.rscp”, after running the import/export example program you will have a “add.rscp.tcp” directory. After starting Vivado up in Tcl mode (“vivado -mode tcl”) you can execute the following commands to import the checkpoint:

```
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
```

At this point the Vivado GUI will open and you will see that there are cells and nets associated with this design. If you want to save the imported design for later you could also do the following command:

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

You could then later re-load that into Vivado using the following commands from the Tcl window:

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

You will notice that these instructions focus on running Vivado in Tcl mode to do all the commands given. The Vivado GUI is then started once the commands have executed. Why not just always run in GUI mode since those same commands could have been typed into the command window in the GUI? It turns out that running commands such as these while the GUI window is open makes them run many times slower than if they were run just in Tcl mode. So, a typical use case you might find helpful is to always start Vivado in Tcl mode, do the commands needed to export or import things, and then open the GUI (using “start_gui”) and close the GUI (using “stop_gui”) as needed.

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 terminology 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 Virtex Artix Spartan Zynq	Kintex Virtex	Kintex Virtex

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 "xc7a100tcsg324-3" as an example. This part is within the Kintex UltraScale family, uses a "tcsg324" package type, and has a speed grade of "3".

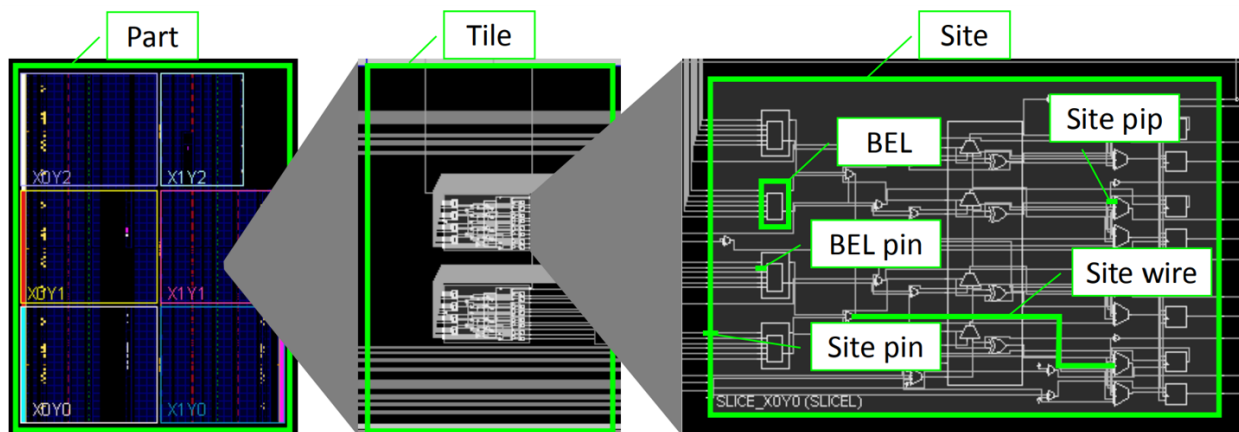


Figure 3: Xilinx Device Hierarchy

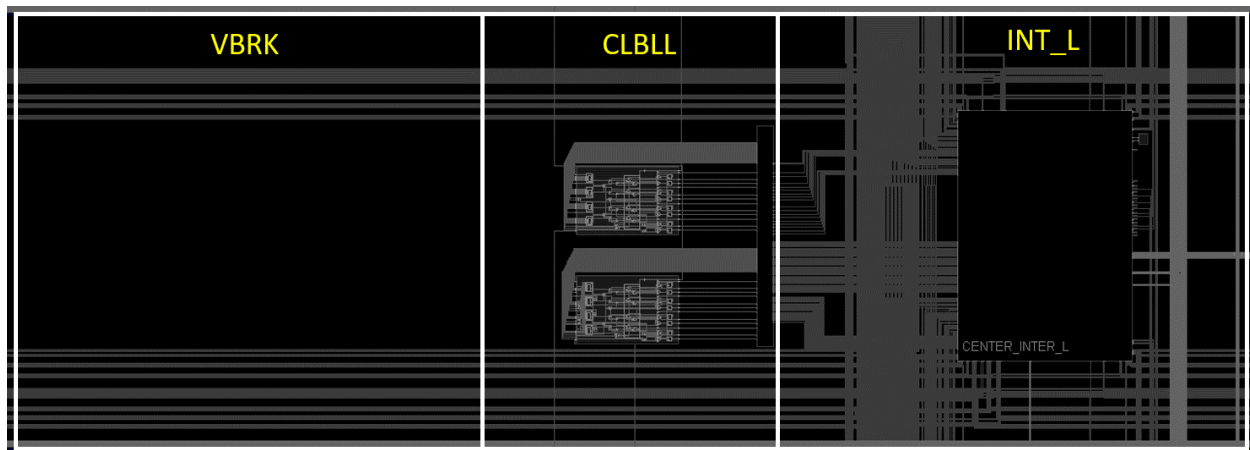


Figure 4: Artix7 Tiles

Figure 3 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 4 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.

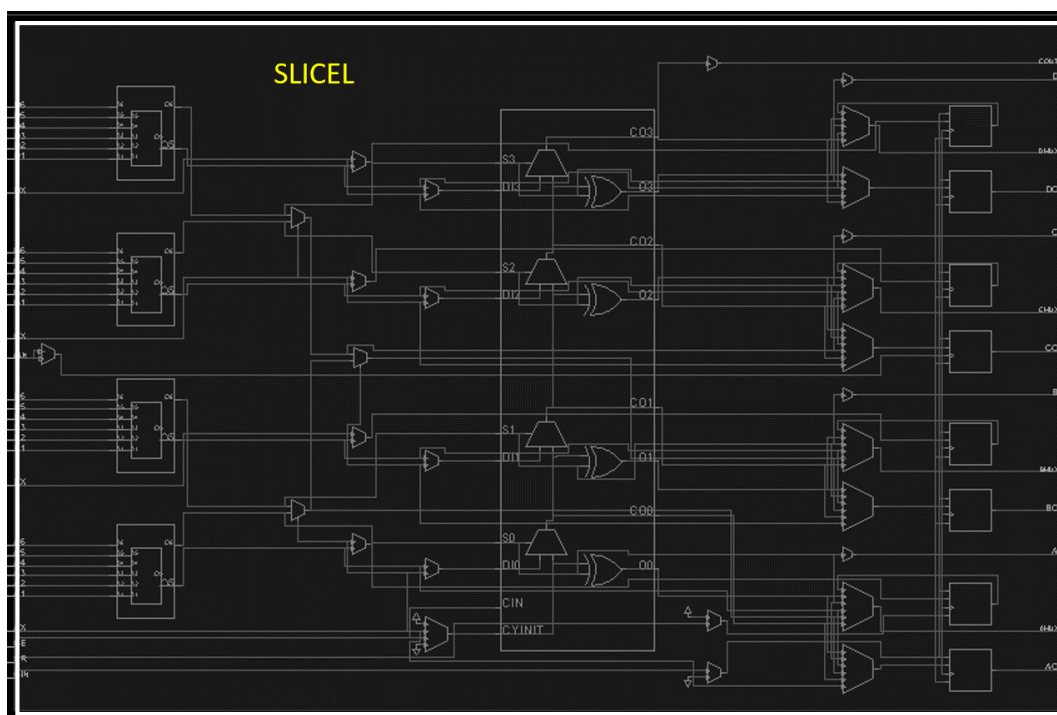
4.1.3 Sites

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 5 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:

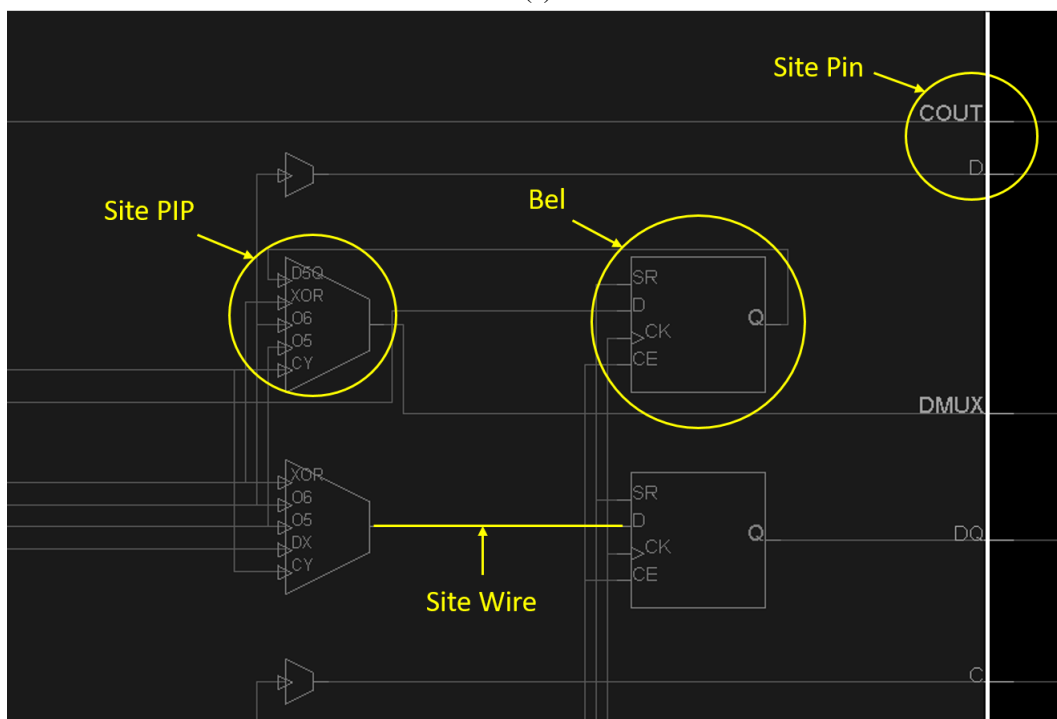
- **Site PIPs:** Also called routing muxes, these are reconfigurable routing PIPs used to specify the internal routing of a site. In Vivado, site PIPs are usually configured automatically as cells in a design are placed (based on cell properties and placement locations).
- **BELs:** **B**asic **E**lements 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 Pins:** 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 5: Series7 SLICEL Site (a), and Highlighted Site Components (b)

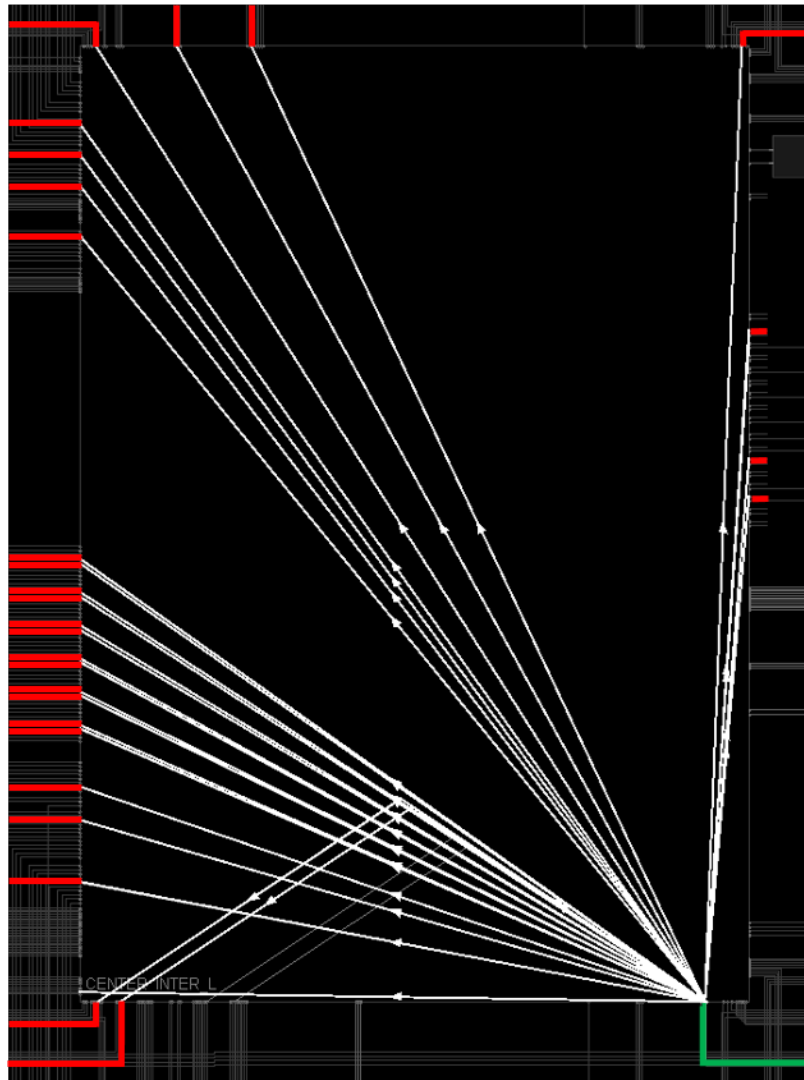


Figure 6: An example switchbox tile. The green wire represents a source wire, and the red wires represent all possible sink wires in the switchbox. The highlighted white sections of the figure are PIP connections.

PIPs are most commonly found in switchbox tiles, and enable a single wire to be routed to several locations on the chip. Figure 6 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 7 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.

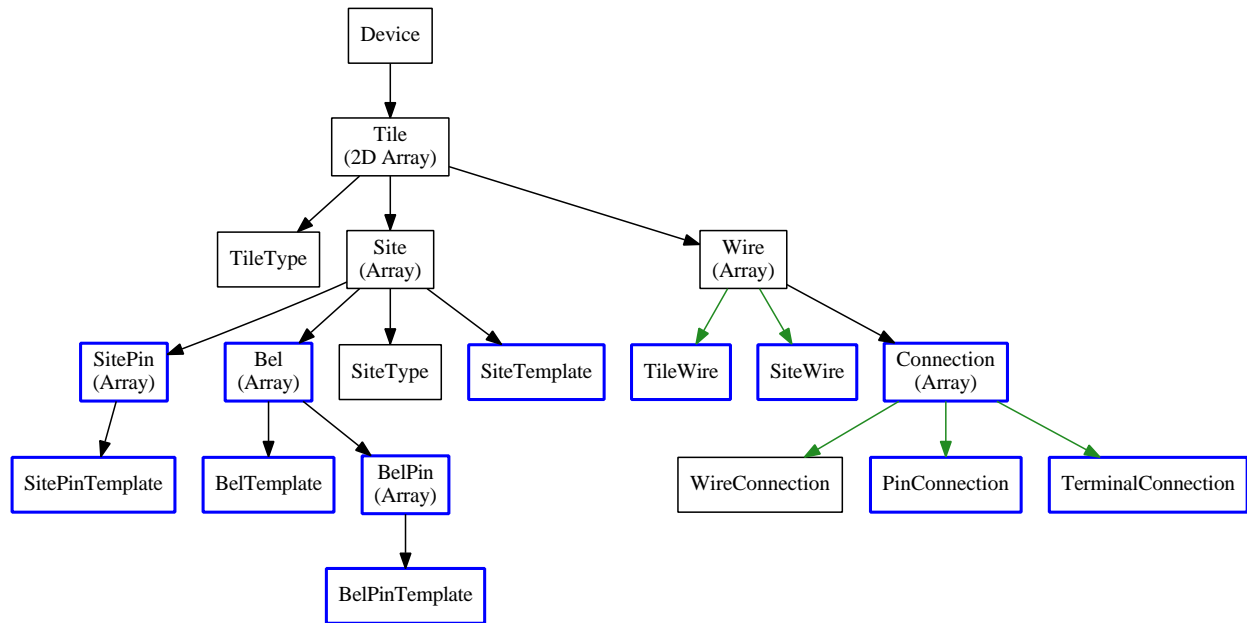


Figure 7: RapidSmith2 Device data structure tree. Green arrows represent inheritance, and black arrows represent association. Classes and Interfaces bolded in blue are new to RapidSmith 2.

Listing 1: Basic device function calls

```

1  // Get a handle to a device
2  Device device = Device.getInstance("xc7a100tcsg324-3");
3
4  // Get device components by name
5  Tile tile = device.getTile("CLBLL_R_X27Y130");
6  Site site = tile.getSite("SLICE_X44Y130");
7  Bel bel = site.getBel("D6LUT");
8
9  // Get all device components
10 device.getTiles();
11 tile.getSites();
12 site.getBels();

```

4.2.1 Templates

As Figure 7 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

```

1  // Get a list of all package pins
2  device.getPackagePins();
3
4  // Get a list of clock package pins
5  device.getClockPads();
6
7  // Get an individual package pin based on the BEL
8  Bel bel = device.getSite("IOB_X0Y10").getBel("PAD");
9  PackagePin pin = device.getPackagePin(bel);
10
11 // Package pin functions
12 pin.getName();
13 pin.getSite();
14 pin.getBel();
15 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

```

1 Device device = RSEnvironment.defaultEnv().getDevice("xc7a100tcsg324");
2 //or
3 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 8 shows an example netlist for a 3-bit counter created in Vivado.

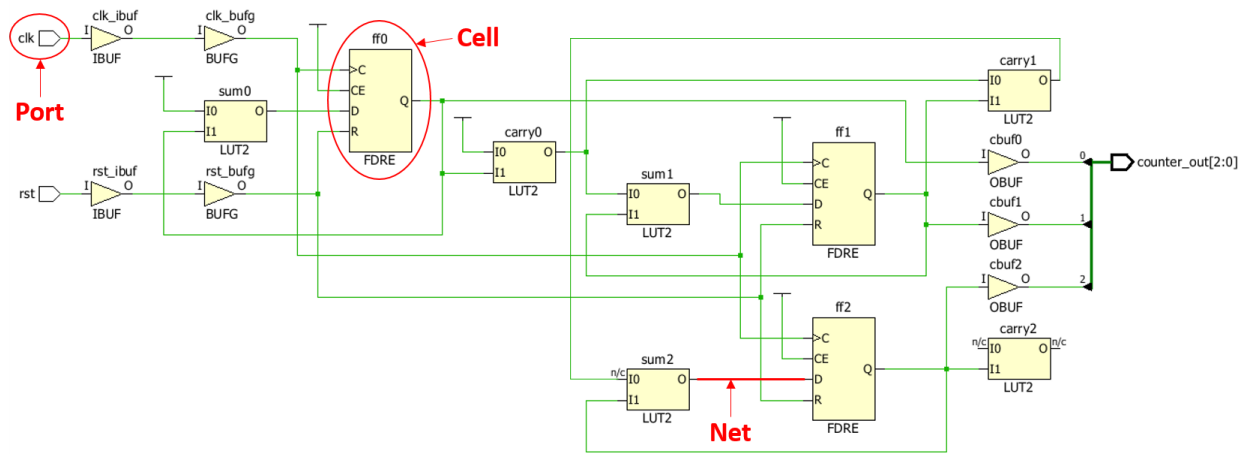


Figure 8: Schematic of a 3-bit counter in Vivado using LUT and FDRE cells. The yellow boxes are cells, the green lines are nets, and the white figures on the edge of the diagram are ports.

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 8 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 9.

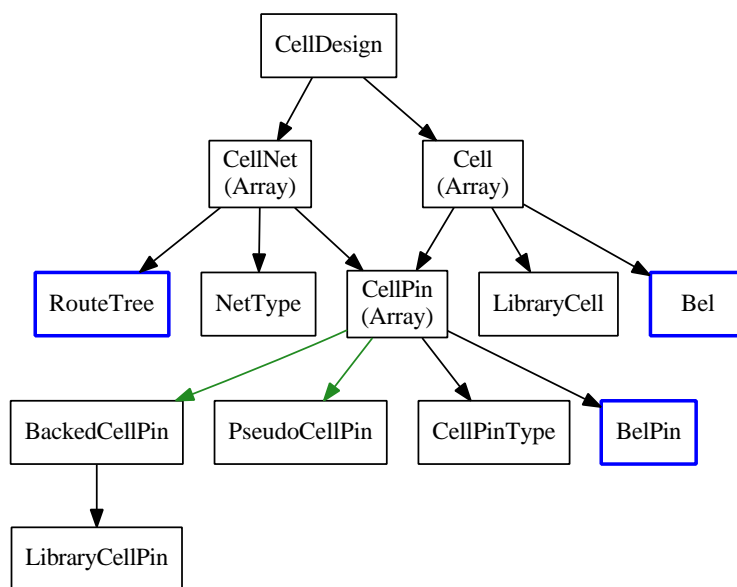


Figure 9: RapidSmith2 design data structure tree. Black arrows represent composition, green arrows represent inheritance, and blue boxes are physical implementation components of the netlist.

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 9 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 `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 Cells.

- 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 Cells 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

```

1 // You need to have a cell library to create a cell
2 CellLibrary libCells = new CellLibrary(RSEnvironment.defaultEnv()
3     .getPartFolderPath("xc7a100t-csg324")
4     .resolve("cell_library.xml"));
5
6 // How to create a new cell. The first argument is the name of the cell, the
7 // second argument is the library cell
8 Cell cell = new Cell("myCell", libCells.get("LUT6"));
9
10 // Get all cells in a design of a certain type
11 CellDesign design = getCellDesign();
12 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 subsection 5.3.1).
- Cells can be placed onto `Bel` objects of the current `Device`. See section 6 for more information about cell placement.
- Top-level Ports in Vivado (design input/output) are represented as Port Cells in RapidSmith2. Specifically, there are three types of port cells: `IPORT`, `OPORT`, and `IOPORT`. The method `CellDesign::getPorts()` can be used to iterate through the ports in a design, and `Cell::isPort()` can be used to determine if a given Cell is actually a port.
- RapidSmith2 supports both Xilinx macro and leaf cells. More information about these is given in subsection 5.1.

5.2.3 CellPin

CellPins in RapidSmith2 are attached to Cell objects and are equivalent to the cell pins found in Vivado. Each CellPin has an associated CellPinType and PinDirection. Table 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 subsection 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

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

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

```

1  // get a handle to a design
2  CellDesign design = getCellDesign();
3
4  // creating a new net
5  CellNet net = new CellNet("myNet", NetType.Wire);
6  design.addNet(net);
7
8  Cell cell1 = design.getCell("cell1");
9  Cell cell2 = design.getCell("cell2");
10
11 // connecting nets to cell pins
12 net.connectToPin(cell1.getSourcePin());
13 net.connectToPin(cell2.getpin("a"));

```

- CellNets are routed using RouteTree data structures. RapidSmith2 routing is described in more detail in section 7.
- All CellNets have a NetType enumeration. Possible values for NetType include VCC, GND, and WIRE. VCC is reserved for power nets, GND is reserved for ground nets, and WIRE represents all other nets in the design.
- The suggested approach to working with static nets in RapidSmith2 is to have only one VCC and GND net in a CellDesign. In general, this representation is much easier to work with and the special nets can be obtained with the functions CellDesign::getVccNet() and CellDesign::getGndNet(). When a design is imported from Vivado through a RSCP, all VCC and GND nets are collapsed automatically. Having multiple VCC and GND nets, however, is still supported if desired.
- Most nets have a single driver, but some can be sourced in multiple locations. Figure 10 shows an example for a GND net. RapidSmith2 handles this oddity by allowing CellNets to have more than one RouteTree object associated with it. In the case of Figure 10 the net would have two RouteTrees, one for each source TIEOFF.

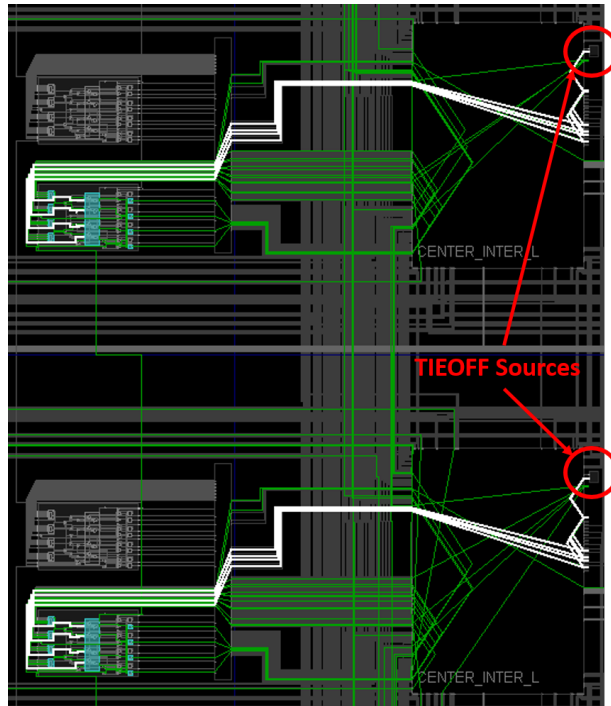


Figure 10: Example of a Vivado net with multiple sources. The highlighted wires in white are all part of the same net.

- Figure 11 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()`.

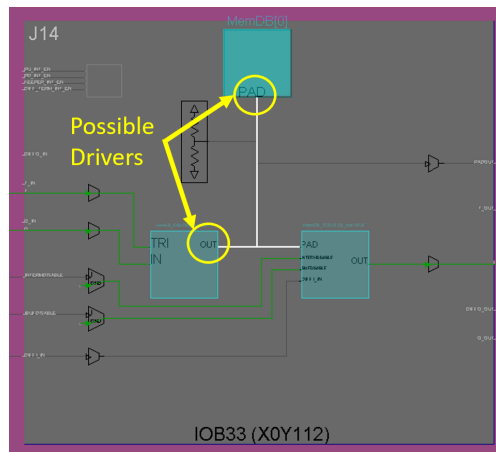


Figure 11: Bidirectional Net

- After a design has been placed, `CellNets` fall into one of two categories: **intrasite** vs. **intersite**. Figure 12 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.

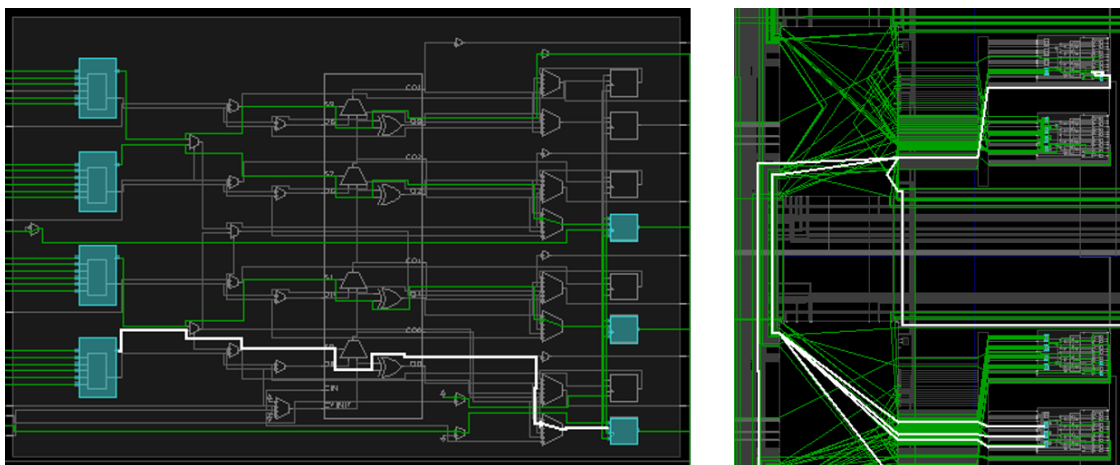


Figure 12: Example INTRASITE Net (left) and INTERSITE Net (right)

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 13 for an IOBUF cell. An IOBUF macro cell contains two **internal** cells: one of type OBUFT and the other of type IBUF. It also contains one internal net that connects the two internal cells together. External cell pins of the macro connect to one or more internal cell pins.

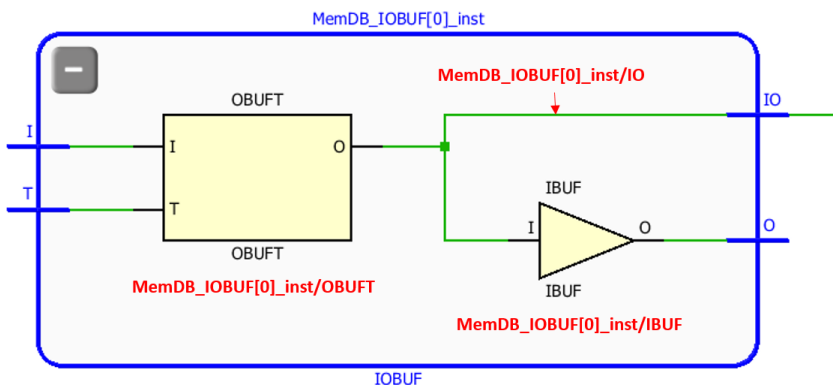


Figure 13: Vivado Macro Cell

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

```

```

10 CellPin pin = macro.getPin("IO");
11 design.getNet("TmpNet").connectToPin(pin);
12
13 // Iterate through a list of all cells (macro and leaf cells) of a design
14 for (Cell cell : design.getCells) {
15     if ( cell.isMacro() ) {
16         List<Cell> internalCells = cell.getInternalCells();
17         List<CellNet> internalNets = cell.getInternalNets();
18         List<CellPin> externalPins = cell.getPins();
19         // do something with the macro info
20     }
21     else {
22         // do something with a regular leaf cell
23     }
24 }

```

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 14 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 14: 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

```

1 // Create a new FF cell with default properties
2 CellLibrary libCells = getCellLibrary();
3 LibraryCell libCell = libCells.get("FDRE");
4 Cell cell = new Cell("myCell", libCell);
5
6 // Get a handle to the cells properties
7 PropertyList properties = cell.getProperties();
8
9 // Print the configurable properties of the cell
10 for(String propName : libCell.getConfigurableProperties()) {
11     Property prop = properties.get(propName);
12     System.out.println(propName + ":");
13     System.out.println("\tDefault -> " + prop.getStringValue());
14     System.out.println("\tPossible -> " + libCell.getPossibleValues(propName));
15 }
16
17 // Iterating over a PropertyList
18 for (Property prop : properties) {
19     System.out.println(prop.getKey() + " -> " + prop.getStringValue());
20 }
21
22 // Change the FF to be falling edge triggered...this will override the default
23 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 strings².
- 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.

²RapidSmith makes no attempt to parse the Vivado properties into their corresponding data structures. All Vivado properties are represented using Strings, and it is currently up to the user to parse the properties if they need to

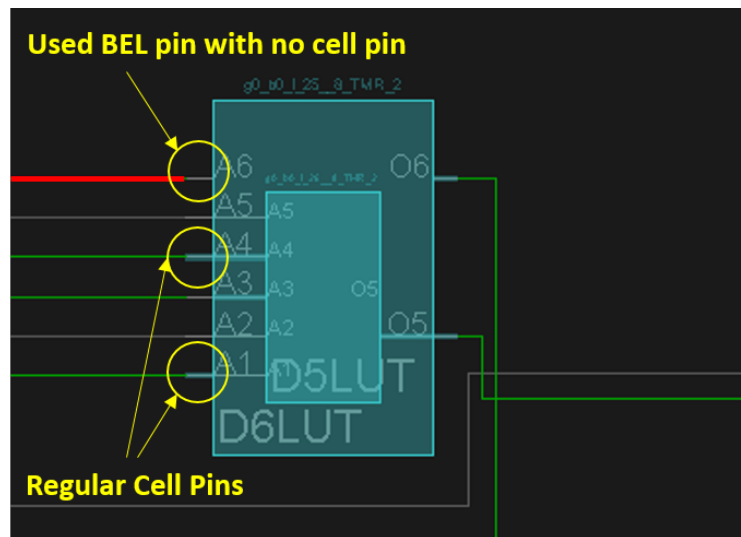


Figure 15: An example of VCC routing to an unused BEL pin (A6)

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 15. 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?

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 15, 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
```

```

2  CellDesign design = loadDesign();
3  Device device = loadDevice();
4
5  // get a handle to the appropriate cells, nets, and bel pins
6  Cell cell = design.getCell("foo");
7  CellNet vcc = design.getVccNet();
8  BelPin bp = device.getSite("SLICE_X0Y179").getBel("D6LUT").getBelPin("A6");
9
10 // create and attach the psuedo cell pin, and map it to the BelPin
11 CellPin pseudo = cell.attachPseudoPin("VccTmpPin");
12 vcc.connectToPin(pseudo);
13 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 16 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.

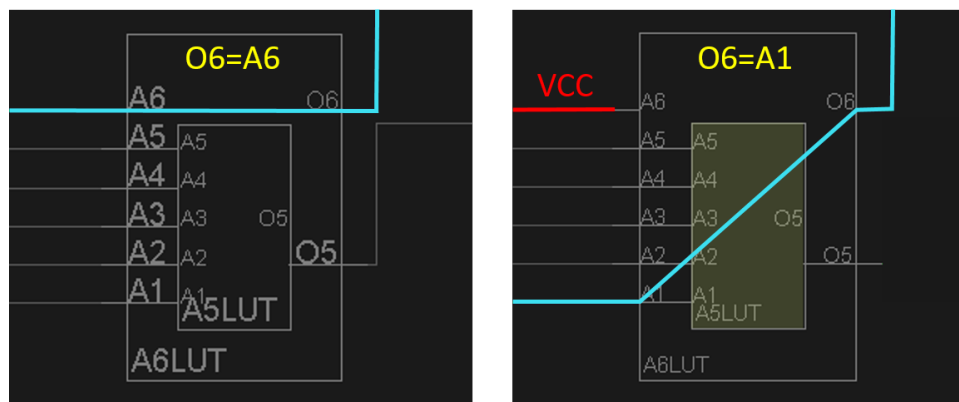


Figure 16: Two examples of LUTs configured as routethroughs in Vivado. The net highlighted in red represents VCC.

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

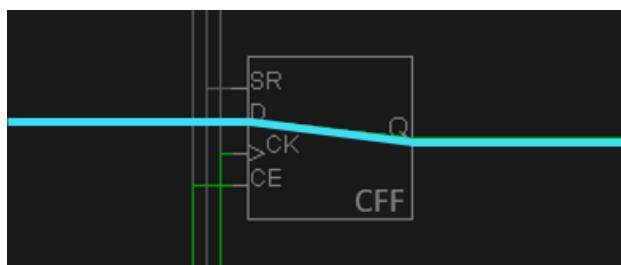


Figure 17: Flip-Flop BEL Configured as a Permanent Latch in Vivado

is retained. An example is shown in Figure 17. As the figure shows, permanent latches look very similar to LUT routethroughs described in the previous section. Because of this similarity, RapidSmith2 treats 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 18. 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.

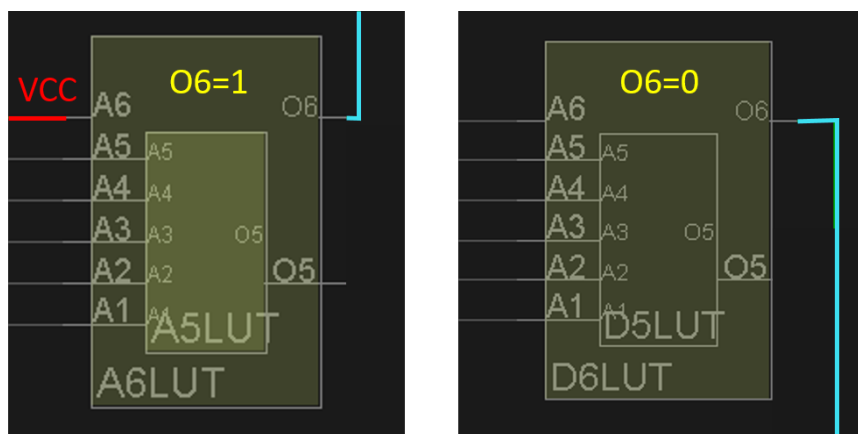


Figure 18: Two LUTs Configured as Static Sources in Vivado

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 19 where the used site PIPs are circled in red. As can be seen, the ACY0 : 05 site PIP is enabled and connects the 05 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)`

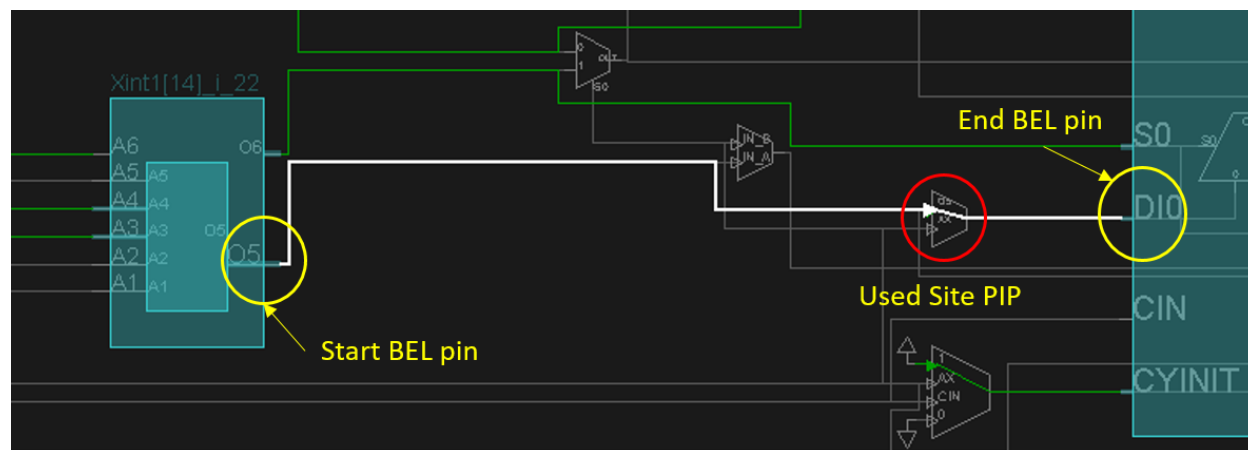


Figure 19: Site PIP Usage

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.

5.4 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 the format for a **cell library XML**. A cell library contains the following information for each library cell that can target a specific device:

- Type
- Group (i.e. SLICE, DSP, IOB, BRAM, etc.)
- Name, direction, and type for each library cell pin
- Valid placement locations for instances of the library cell
- Default logical-to-physical pin mappings for each cell pin
- Configurable properties with default values
- Macro templates

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 generated³. 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

```

1 // First way, load a Tincr Checkpoint
2 TincrCheckpoint tcp = VivadoInterface.loadTcp("checkpoint.tcp");
3 CellLibrary libCells1 = tcp.getLibCells();
4
5 // Second way, directly load the cell library from disk
6 CellLibrary libCells2 = new CellLibrary(RSEnvironment.defaultEnv()
7     .getPartFolderPath("xc7a100t-csg324-3")
8     .resolve("cellLibrary.xml");

```

5.4.1 Generating A New Cell Library

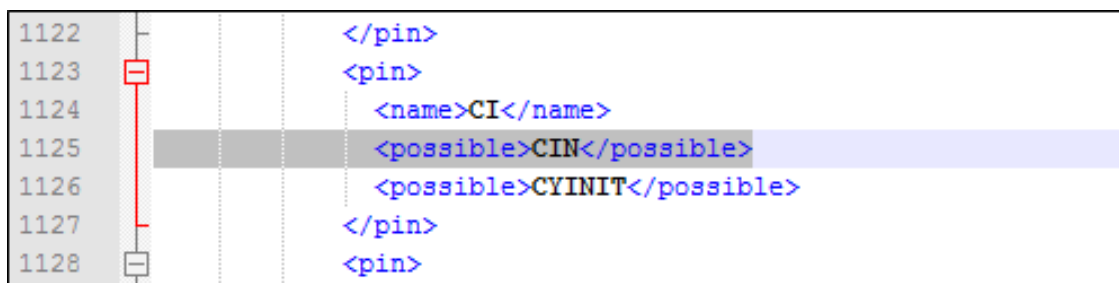
The Tincr command `[tincr::create_xml_cell_library]` can be used to generate a new cell library for a device. Specifically, follow the steps listed below to create a new cell library (the items marked with **SERIES7** only need to be done for series7 families).

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
```

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

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”:



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

3. 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.

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:

```
<?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 10.

Listing 10: 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 11 demonstrates the basic steps to placing cells in RapidSmith2.

Listing 11: Steps for placing a Cell in RapidSmith2

```

1  // Load the device and design
2  TincrCheckpoint tcp = VivadoInterface.loadTcp("myCheckpoint.tcp");
3  Device device = tcp.getDevice();
4  CellDesign design = tcp.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 20 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 12. The correct pin mappings are always used when a RSCP is imported into RapidSmith2.

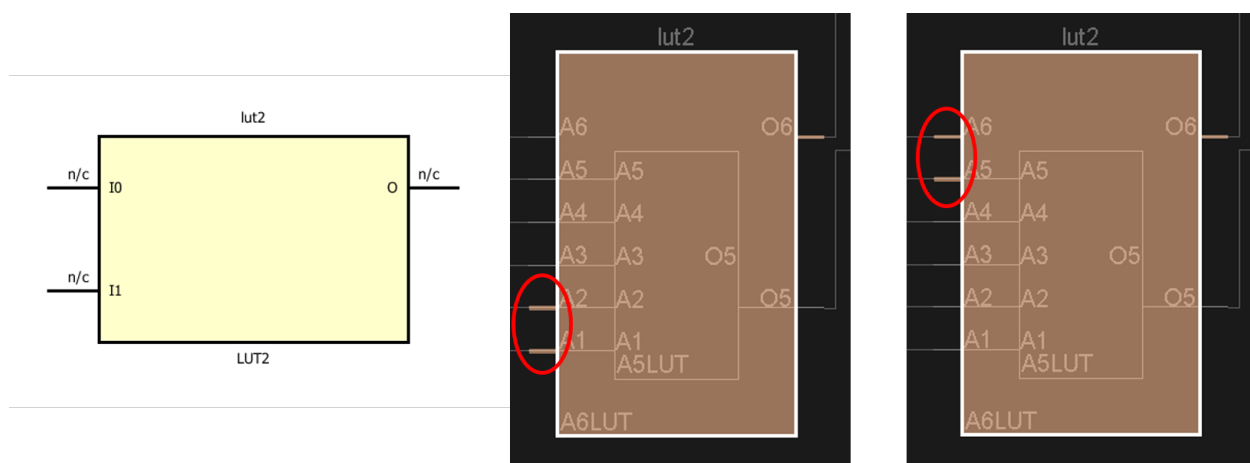


Figure 20: LUT Pin Permutation Example

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

```

1  proc print_pin_mappings{cell} {
2      foreach cell_pin [get_pins -of $cell] {
3          puts "$cell_pin -> [get_bel_pins -of $cell_pin]"
4      }
5  }

```

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

- VCC and GND cells are not placed when implementing a design in Vivado. This distinction is applied to RapidSmith2 as well. Rather than placing VCC or GND explicitly, `RouteTrees` that are sourced by switchbox TIEOFFs are used to express their placement implicitly (VCC/GND is “placed” on the TIEOFF).
- A list of valid placement 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 be 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 6). 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 21. 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.

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 13 demonstrates how to iterate over `Connection` objects. To gain a better understanding of how to use Wires and Connections, see the **HandRouter** example in the RapidSmith2 repository.

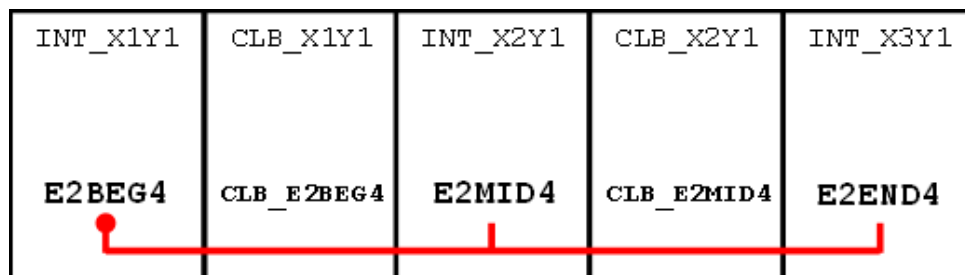


Figure 21: Multi-Tile Xilinx Wire

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

Listing 13: 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

Wires and WireConnections are the fundamental objects used to specify and explore routing in RapidSmith2, but they need to be organized in a higher-level data structure to give meaning to the route of a CellNet. In the original RapidSmith, creating this data structure was up to the user. RapidSmith2 introduces the `RouteTree`, which can be seen in Figure 22.

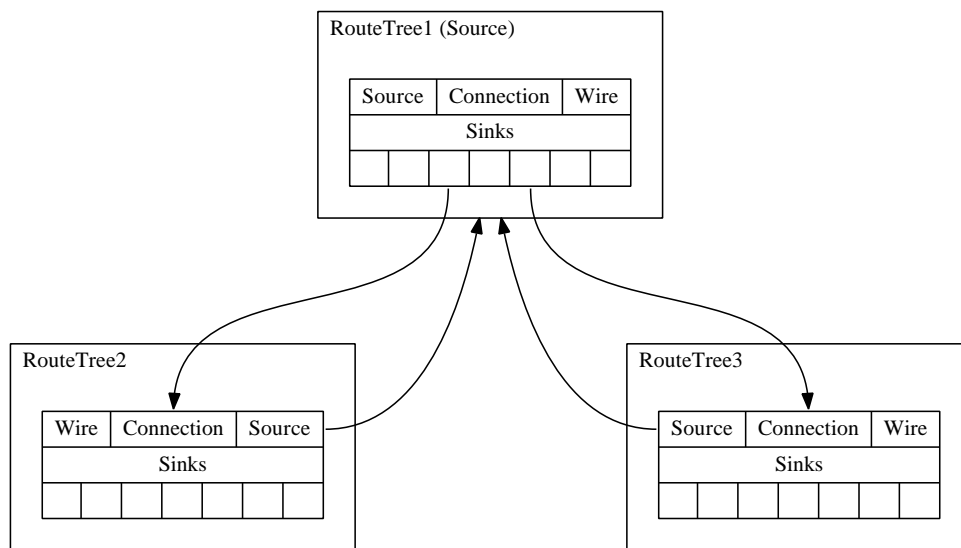


Figure 22: Visual Representation of a RapidSmith2 RouteTree

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 23 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

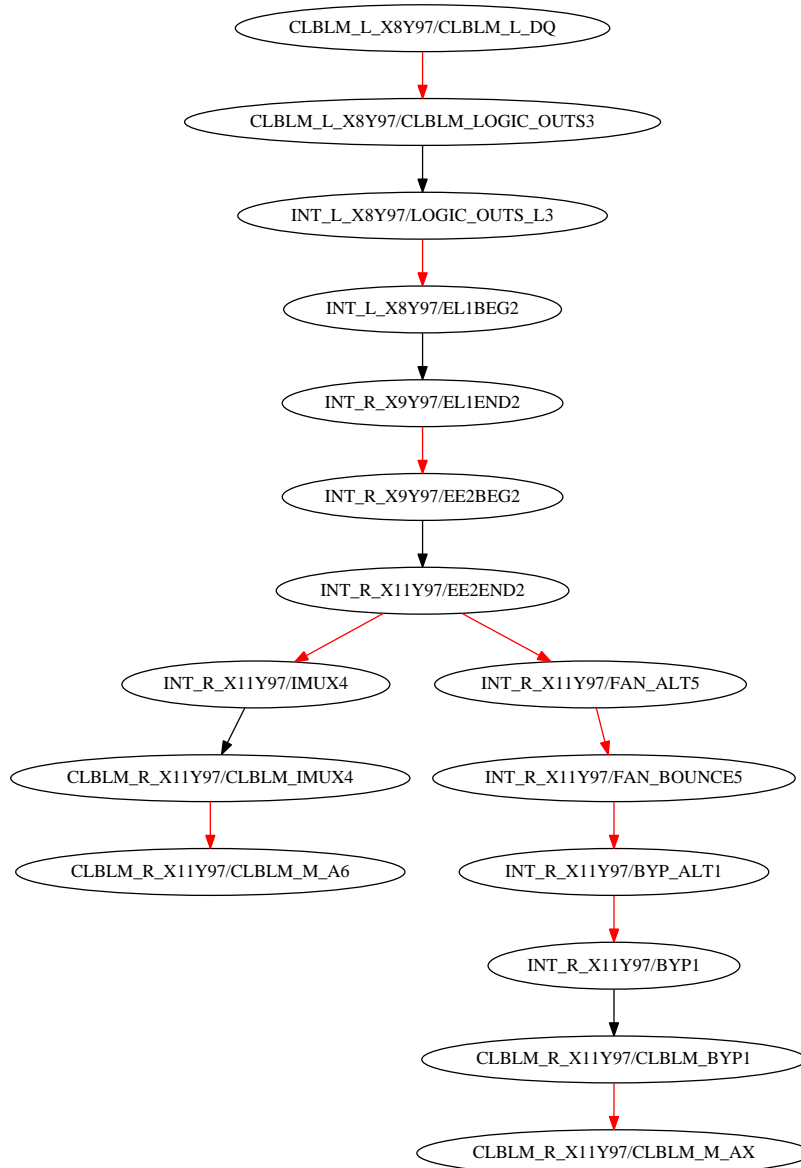


Figure 23: Sample RapidSmith2 RouteTree (red edges represent PIP connections)

are reached. Listing 14 demonstrates the basic usage of `RouteTrees` in `RapidSmith2`. The **DesignAnalyzer**, **AS-tarRouter**, and **HandRouter** examples in the `RapidSmith2` repository also demonstrate how to traverse and build a `RouteTree`.

Listing 14: Building a `RouteTree`

```

1  // Find a Wire to start the RouteTree at
2  Site site = device.getSite("SLICE_X5Y84");
3  SitePin pin = site.getSitePin("DQ");
4  Wire startWire = sink.getExternalWire();
5
6  // Create the first node in the RouteTree
7  Queue<RouteTree> rtQueue = new LinkedList<RouteTree>();
8  RouteTree start = new RouteTree(startWire);
9  rtQueue.add(start);
10
11 // Build up the RouteTree somehow
12 while (!amDone()) {
13     RouteTree current = rtQueue.poll();
14     Wire wire = route.getWire();
15
16     for (Connection conn : wire.getWireConnections()) {
17         // add qualified connections to the RouteTree
18         if (isQualified(wire)) {
19             RouteTree tmp = current.addConnection(conn);
20             rtQueue.add(tmp);
21         }
22     }
23 }

```

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 need to be **converted to an equivalent `RouteTree` representation** before exporting the design to Vivado.

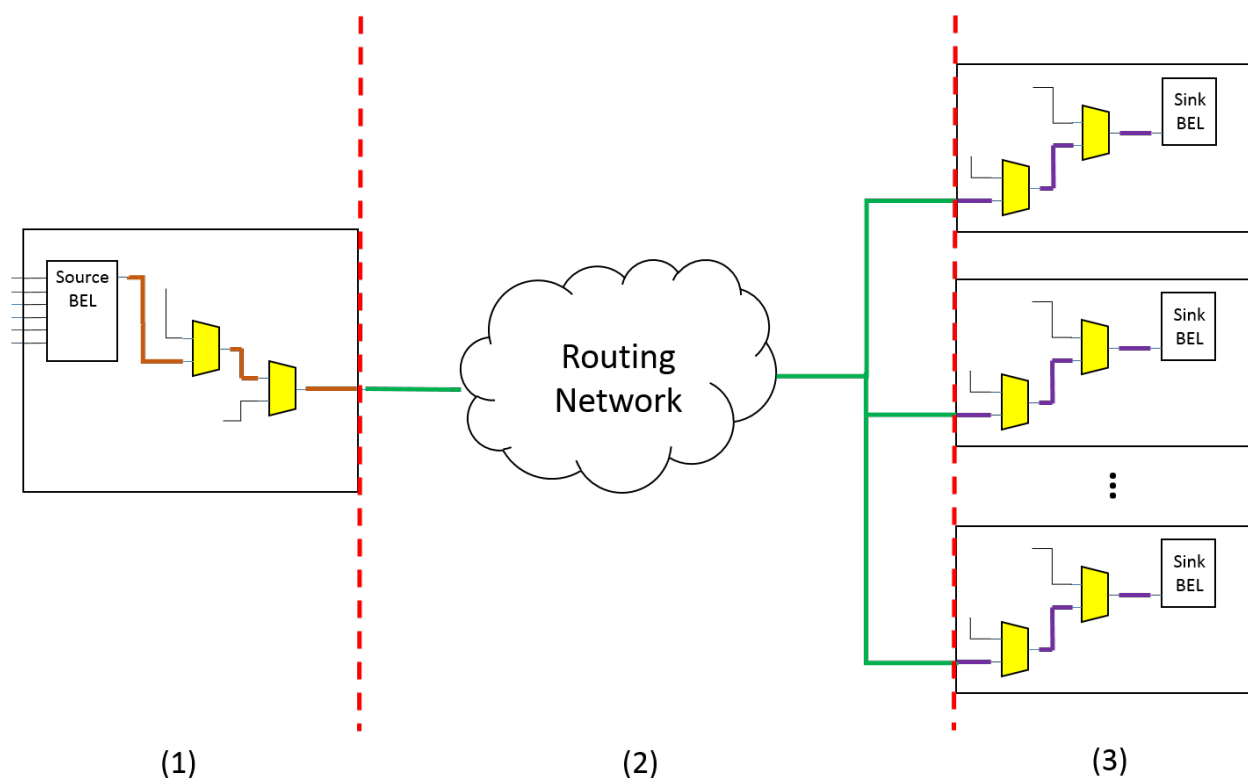
7.5 Three Part Routing

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 net⁴.
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 24 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 15 demonstrates how to utilize three-part routing in `RapidSmith2`. On design import, the routing sections of each `CellNet` are created automatically.

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

**Figure 24:** Three-Part Routing**Listing 15:** Demonstration of three-part routing in RapidSmith2

```

1  // Get a handle to a routed net in the design
2  CellNet net = design.getNet("myNet");
3
4  // Handling the source RouteTree
5  RouteTree source = net.getSourceRouteTree();
6  net.setSourceRouteTree(createSourceRoute());
7
8  // Handling the intersite RouteTree
9  RouteTree intersite = net.getIntersiteRouteTree();
10 net.addIntersiteRouteTree(createIntersiteRoute());
11
12 // Iterate over a list of sink RouteTrees
13 for (RouteTree rt : net.getSinkSitePinRouteTrees()) {
14     // do something with the RouteTree
15 }
16
17 // Or, get a RouteTree based on a SitePin
18 for (SitePin sitePin : net.getSitePins()) {
19     if (sitePin.isInput()) {
20         RouteTree sinkTree = net.getSinkTree(sitePin);
21         // do something with the RouteTree
22     }
23 }
24
25 // Add a new sink RouteTree that starts at a SitePin
26 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 16 demonstrates how a set of used site PIPs can be created and added to a site. This step needs to be taken **only when you have modified the intrasite routing** for a site.

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

```
1 // Get a handle to a Design and a Site
2 CellDesign design = tcp.getDesign();
3 Device device = tcp.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 modifying Vivado designs post-synthesis, post-place, and post-route as shown in Figure 25. 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.

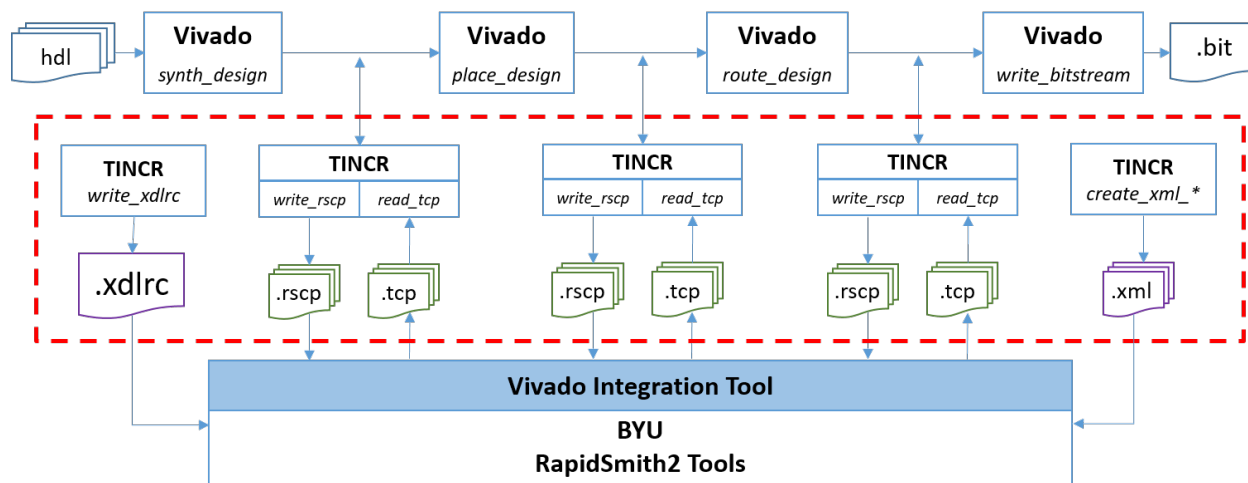


Figure 25: RapidSmith2 Design Flows

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 17.

Listing 17: How to import and export TCP files to and from RapidSmith2

```

1 // Loading a Tincr 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::loadTCP()`, as shown in Listing 18.

Listing 18: Importing a TCP 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 17 demonstrates how to export a design from RapidSmith2, which produces a Tincr Checkpoint

(TCP). To import the TCP back into Vivado, simply open Vivado in Tcl mode and run the command `[tincr::read-tcp myCheckpoint.tcp]`

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 subsection 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 `PseudoCellPins`, are added to the global VCC/GND net, attached to the cell placed at the corresponding BEL, and then mapped to the BEL pin.
- 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.

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, functionally equivalent**.

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 following commands to re-build one of the example designs.

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

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.

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 2. 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 26). 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.

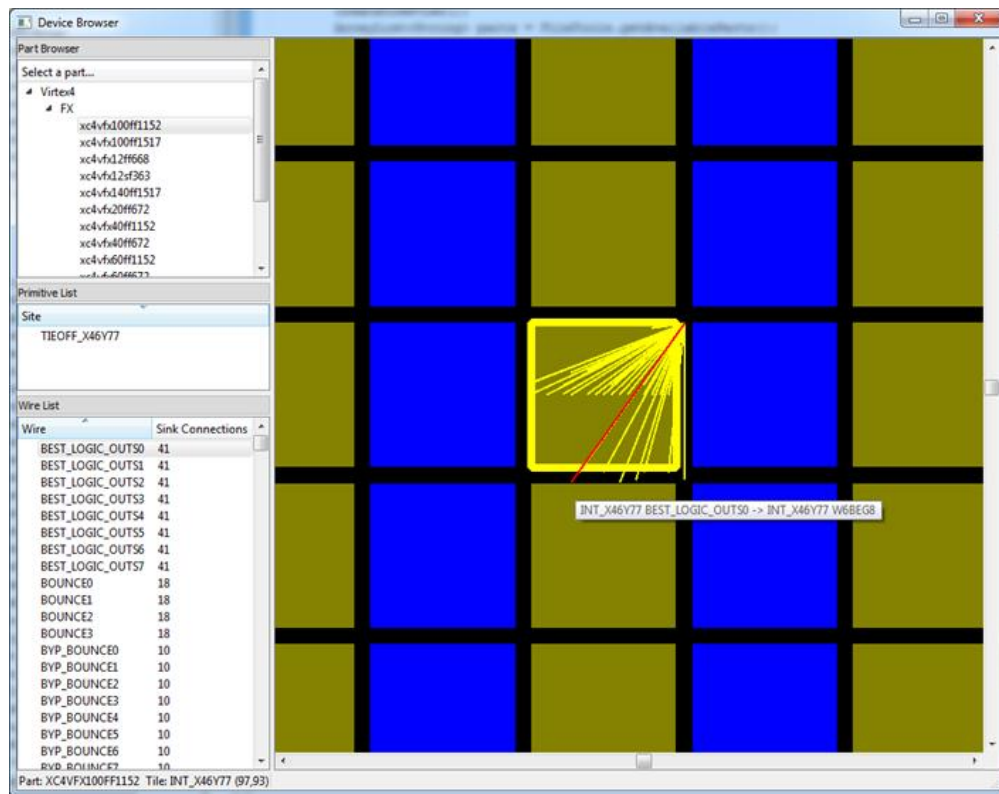


Figure 26: DeviceBrowser Screen Shot Showing Wire Connections

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 `Cells` and `CellNets`) 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 `Cells` and add them to an existing netlist
- Create new `CellNets`, and connect them to `CellPins`
- Modify the properties on `Cells`
- Place `Cells` onto a `Bels`
- 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, three `Tincr` intermediate files are required:

1. XDLRC
2. Family Info XML
3. Device Info XML

The contents and format of these files are described in great detail in Appendix A, Appendix B, and Appendix C respectively, and will not be described here. For those that are curious about what each file represents, refer to the listed appendices. 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 19 for the Artix7 part `xc7a100tcsg324-3`. The “-max_processes” option is used to parallelize the operation so that it will execute faster. This Tcl command can take a very long time to run (more than 24 hours for very large devices), 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 19: XDLRC Generation Example

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

2. Run the `Tincr` command `[tincr::create_xml_device_info]` to create a `deviceInfo.xml` file for the part. Copy the 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 20. The device installer creates compact device files that represent a Xilinx device from the XDLRC and `deviceInfo.xml` generated in the previous steps. Notice the two JVM command line arguments used in the command. The first option (“-ea”) enables assertions for the code. It is important to include this flag so that device file errors can be caught during parsing. The second option (“-Xmx4096m”) sets how much memory the JVM can use while running the installer. Since XDLRC files are quite large, the memory usage of the installer grows very quickly. If the device installer fails with an out of memory exception, you will need to increase the memory and re-run the installer (up to 32 GB of memory may be required).

Listing 20: RapidSmith2 device installer example usage

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

4. Run the family builder in RapidSmith2 and pass the name of the newly created part as a command line argument. An example usage is shown in Listing 21 for an Artix7 device. An `Artix7.java` file (or whatever family your device is in) will already exist, but will be updated with new sites and tile types from the newly installed part.
5. The final step is to create a *cellLibrary.xml* file for the device. This file details all Xilinx primitives that can be used for implementing designs on the given device. subsection 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 21: Family builder example usage

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

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

Listing 22: 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 supports implementing FPGA designs on devices for the following families (also called architectures):

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

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

1. Create the primitive definitions of the family using VSRT. The VSRT user guide is given in 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 23 for Kintex UltraScale. As the listing shows, there are three arguments to the command:
 - **familyInfo.xml**: The file name to store the generated family info. The file ending “.xml” will be appended if it is not included.
 - **kintexu**: The Vivado family name.

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

Listing 23: Family info example usage

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

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

Listing 24: How to access SiteTypes and TileTypes in RapidSmith2

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

Listing 25: 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 33). 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 26 can be used to select a specific site type in Vivado to view its alternate types.

Listing 26: Tcl commands to select a Vivado Site object

```

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

```

2. The second hand edit is to add alternate type pin mappings. When a site is changed to one of its alternate types in Vivado, the site pins can be renamed. An example is shown in Figure 27 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 27 shows how to add pin renamings to the family info XML using the "pinmaps" tag. To determine the actual pin mappings, the first step is to open two instances of the Vivado GUI. For each site in the family info, load the default type in one Vivado instance, load each alternate type in the other instance, and visibly check what pins are renamed in the alternate type (as demonstrated in Figure 27). Table 3 gives a list of all alternate types that rename pins for Artix7 devices.

Listing 27: Sample pinmaps in a family info file

```

<name>ILOGICE3</name>
<alternatives>
  <alternative>
    <name>ILOGICE2</name>

```

```

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

```

Table 3: Artix7 Alternate Pin Mappings

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

- 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.
- 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.

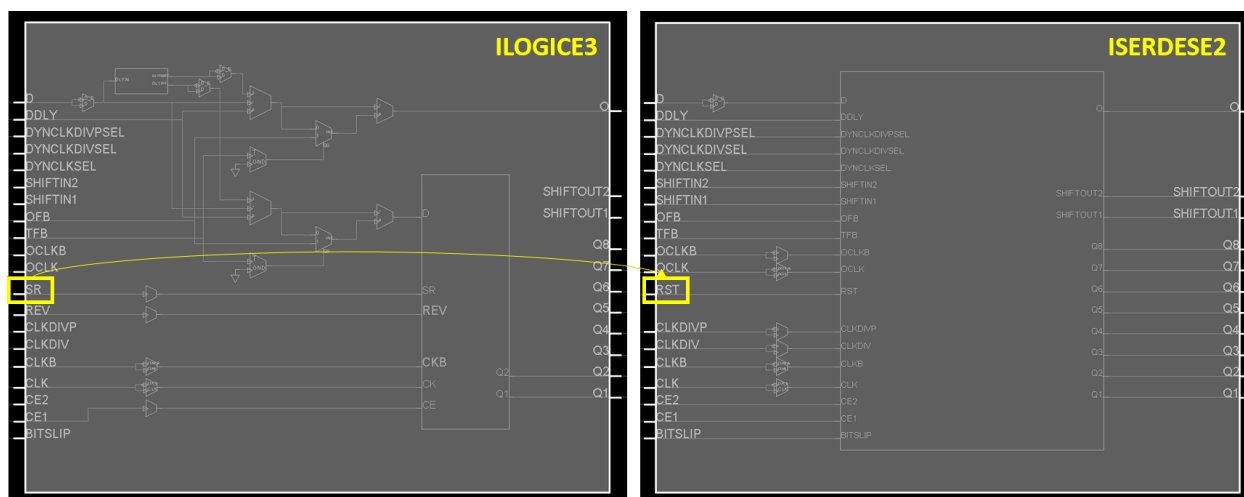


Figure 27: Example Alternate Site Pin Renaming

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 28, shows the two compatible types that were manually added to complete the Kintex UltraScale family info. Other device families may require additional compatible site hand edits. It is up to the user to determine what compatible sites need to be added through experimentation.

Listing 28: Manually added compatible sites for UltraScale devices

```
<site_type>
  <name>SLICEL</name>
  <is_slice/>
  <compatible_types>
    <compatible_type>SLICEM</compatible_type>
  </compatible_types>
  ...
</site_type>
<site_type>
  <name>HRIIO</name>
  <is_iob/>
  <compatible_types>
    <compatible_type>HPIIOB</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 28: 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 29: 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 30: Wire syntax in XDLRC files

PIP

```
# Example of an XDLRC PIP declaration
      (pip INT X7Y63 BEST LOGIC OUTS0 -> BYP INT B5)
```

Figure 31: 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)
  )
  (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 32: 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 32. 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
- Site PIP Corrections
- Compatible Types
- Pin Direction Corrections
- BEL Routethroughs

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 33. 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 29 shows how alternate types are represented in a family info XML. Pinmap tags are included for alternate pin names changing as described in subsection 10.3.

Listing 29: Example Alternate Type XML

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

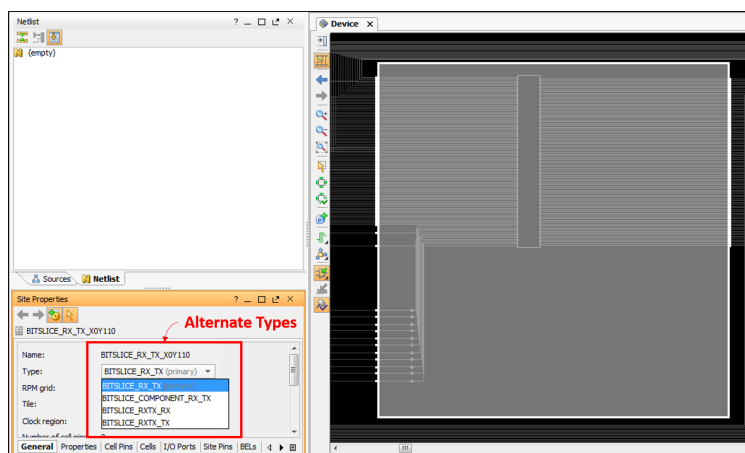


Figure 33: `BITSLICE_RX_TX` Alternate Types

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 34, 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.

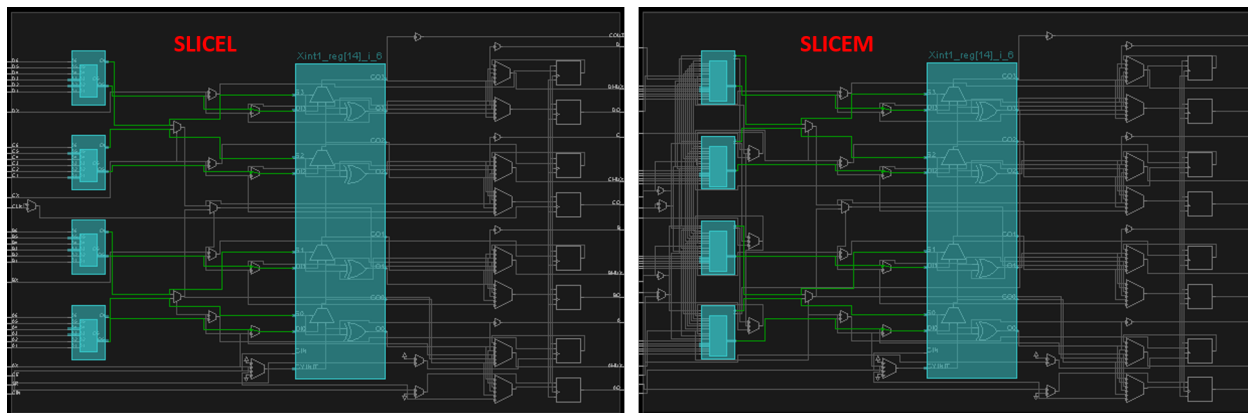


Figure 34: A group of cells placed on a SLICEL site (left) and a SLICEM site (right).

A visualization of compatibility is also given in Figure 35. 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

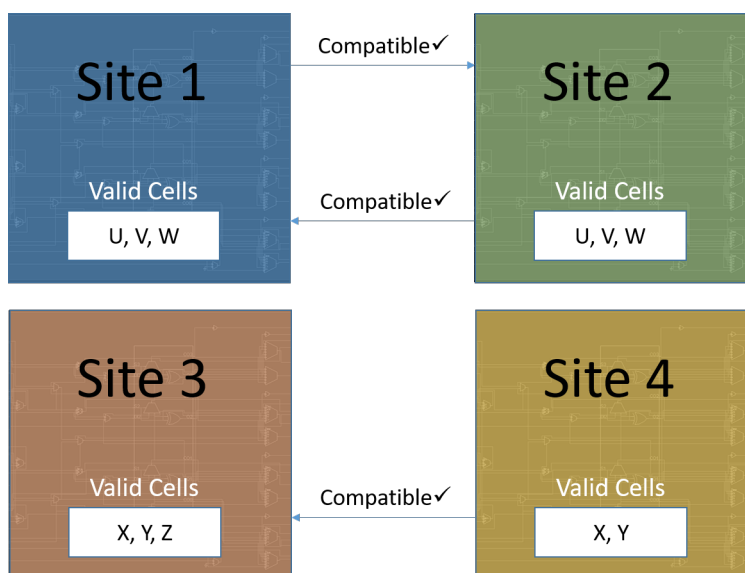


Figure 35: Compatibility Testing for Single-BEL Sites

site type for each group of cells, lowering the quality of results. Listing 30 shows example XML for compatible types in a family info file.

Listing 30: 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 31 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 ??.

Listing 31: 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 32 shows how site PIP corrections are represented in a family info file.

Listing 32: 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 36. 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”.

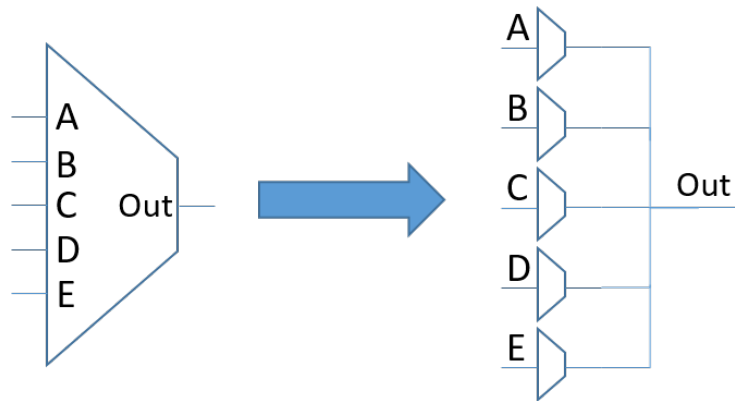


Figure 36: Site PIP Decomposition

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 33 shows how pin direction corrections are represented in XML.

Listing 33: 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 34. 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 34: 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 RSCP and TCP Format

As shown in Figure 1, RapidSmith2 parses RapidSmith Checkpoints (RSCP) generated from `Tincr`, and creates Tincr Checkpoints (TCP) that can be loaded back into Vivado. This documentation does not have a detailed discussion of the contents and format of RSCPs and TCPs. For advanced users, a very detailed description of these checkpoint formats is given in Thomas Townsend's masters thesis located for download at <http://scholarsarchive.byu.edu/etd/6492/>.