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: February 14, 2017

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

Contents

1 Introduction

1.1 What is RapidSmith 2?

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 RS2 hereafter) represents a major addition to RapidSmith. Specifically, Vivado designs are now supported. Using RS2 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 RS2 and then import it into Vivado if desired.

The other new major capability of RS2 is that it changes RapidSmith's design representation. Instead of using XDL's view of a design with Instances and Sites, RS2 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 RS2?

RS2 is aimed at anyone desiring to do FPGA CAD research on real Xilinx devices available in Vivado. As such, users of RS2 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. RS2 is by no means a Xilinx Vivado replacement. It cannot be used without a valid and current license to a Vivado installation (RS2 cannot generate bitstreams for example).

1.3 Why RS2?

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 RS2. 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.

RS2 (in conjunction with Tincr which is described in subsubsection 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. RS2 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 RS2 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 RS2 moving forward.

1.5 How is RS2 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 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 RS2. VPR is currently limited to FPGA features which can be described using VPR's architectural description facilities.

1.6 Why Java?

RS2 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, RS2, and Tincr

2.1 RapidSmith vs. RS2

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 RS2?

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 RS2 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 RS2 to understand the structure of and the resources available for use in a given Xilinx part). Tincr is available at https://github.com/byuccl/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 RS2: A Framework for BEL-Level CAD Exploration on Xilinx FPGAs

The second part of the development of RS2 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 RS2 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 RS2 Integration

The third part of the development of RS2 was to create the ability to export designs from Vivado and into RS2 and, correspondingly, to import RS2 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 RS2 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 RS2 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. RS2 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. RS2 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 RS2, 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 RS2 Usage Model and Structure

The usage model for RS2 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 Tincr Checkpoint which can then be imported into RS2. At those same points in the design flow, RS2 can export a Tincr Checkpoint which can then be imported back into Vivado. Thus, a complete solution involves Vivado, Tincr, and RS2.

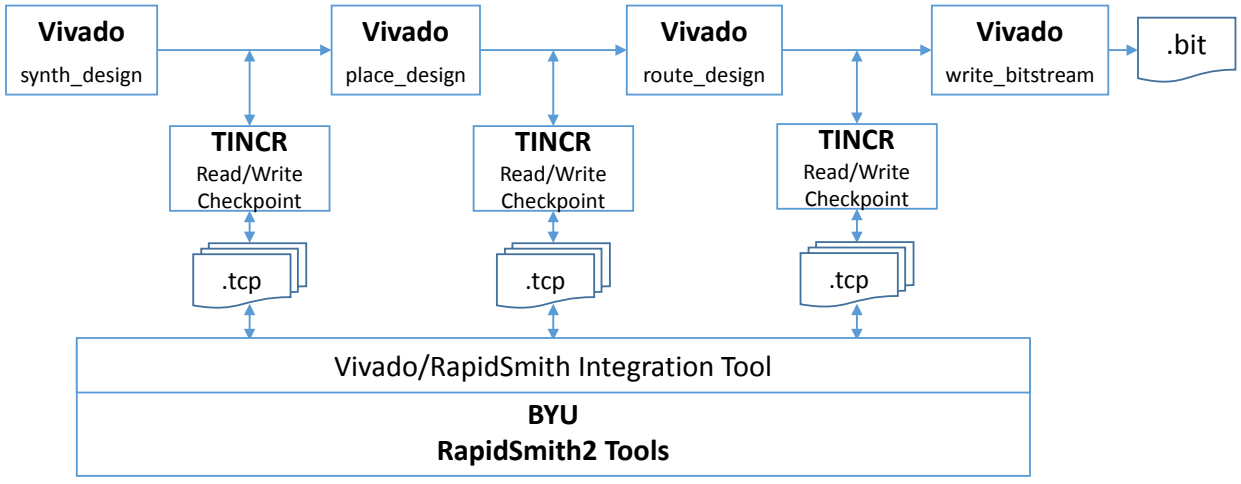


Figure 1: Vivado and RS2 Usage Model

3 Getting Started

3.1 Installation

RS2 is available on Github at: https://github.com/byuccl/RapidSmith2. You can either build RS2 into .class and .jar files for use in any Java environment, or configure RS2 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 RS2. 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 RS2 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 RS2 repository into a local directory.

2. Create a new environment variable called RAPIDSMITH_PATH, and point it to your local repository of RS2 that you setup in step (1). This is needed so RS2 can find required device files and other items at runtime.

3. Build the RS2 project. RS2 is managed using a gradle build system. To build the project, navigate to your local repository of RS2 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 RS2 for use in an IDE, or run RS2 from the command line. Both choices are detailed below.

   Running from an IDE The gradle scripts in RS2 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 RS2 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 RS2 from the command line. The following directories are created:

- *build/classes/main*: This folder contains the RS2 class file directory tree.
- *build/libs*: This folder contains a Jar file of the RS2 class files.
- *build/distributions*: This folder has both .zip and .tar files with contains all Jars needed to run RS2 from the command line. This includes a full jar of the RS2 build alond with copies of dependency Jars (such as QT-Jambi).

After adding the appropriate .class files or Jars to your `CLASSPATH`, you should be able to run RS2 tools from the command line. If you make any changes to the RS2 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 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.

3.1.4 Running RS2 Programs

Some points to keep in mind while configuring and running RS2 programs:

- The RS2 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 RS2 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 RS2 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 RS2 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) RS2 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/*
```

3.1.5 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 RS2 (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.

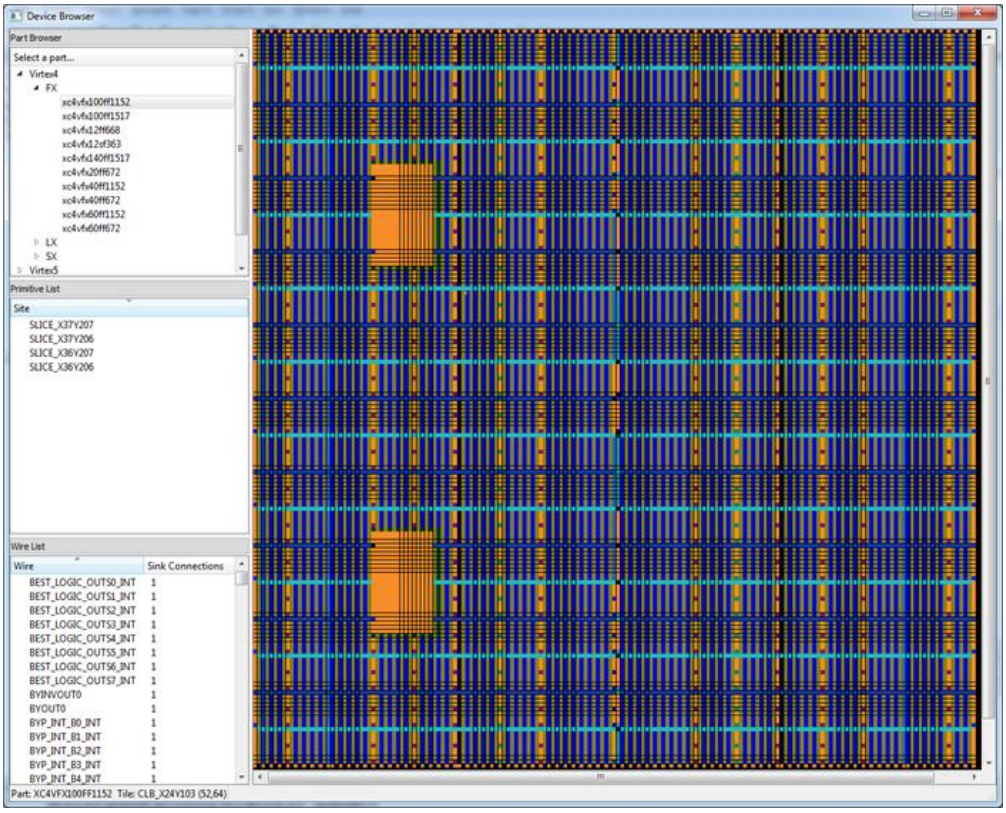


Figure 2: DeviceBrowser Sample Display

3.2 Device Files For Use With RS2

Device files for one part (the *xc7a100tcsg324*) are included in the distribution so you can immediately start working with RS2 (initially, it will be the only device available when you run the **DeviceBrowser** program above). The device files for this part can be found in the *$RAPIDSMITH_PATH/devices/artix7* directory. If you desire to work with additional parts, follow the instructions found in the file *$RAPIDSMITH_PATH/doc/InstallingNewDevices.txt*.

4 Devices in RS2

4.1 Xilinx FPGA Architecture Overview

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

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

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 Tiles

Conceptually, a Xilinx FPGA can be thought of as a two dimensional array of `Tiles`. A `Tile` is a template section within a FPGA that performs a specific function (say, implementing logic equations). Templates are then duplicated across the device (all copies are identical) and different `Tile` types are wired together. As an example, Figure 3 displays three types of template `Tiles` in an Artix7 part.

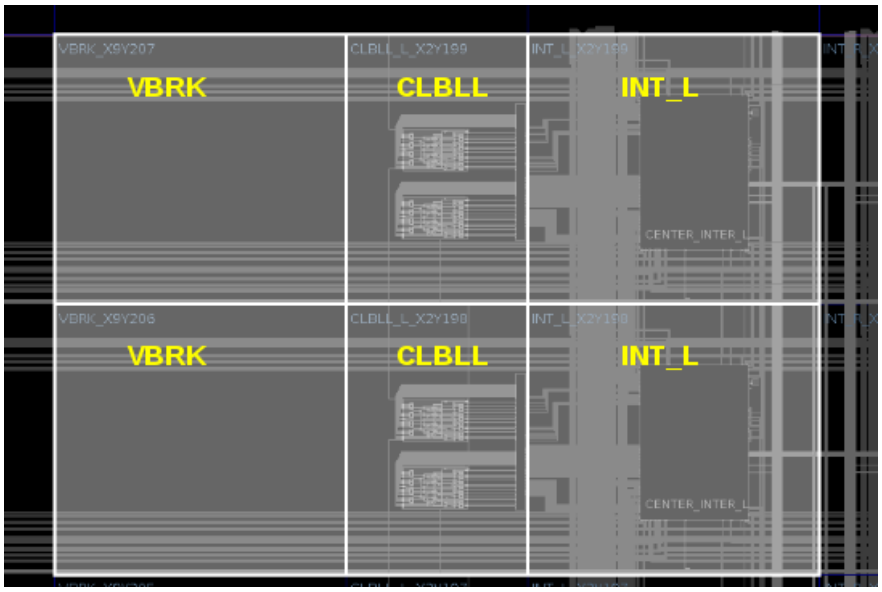


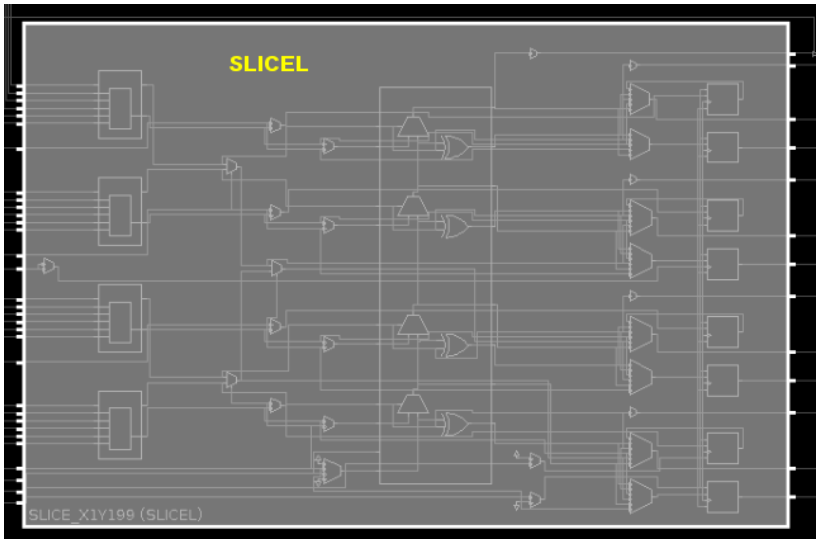
Figure 3: Highlighted example `Tiles` in an Artix7 FPGA.

The two **VBRK** `Tiles` on the left are used for intermediate wiring. The two **INT_L** `Tiles` on the right are switchbox `Tiles`. These are reconfigurable routing `Tiles` that allow a single wire to go multiple different locations within the FPGA. The two **CLBLL** `Tiles` in the middle are used to implement combinational and sequential digital logic. They are the basic building blocks of Xilinx FPGAs. Other `Tile` types include DSP, BRAM, and IOB.

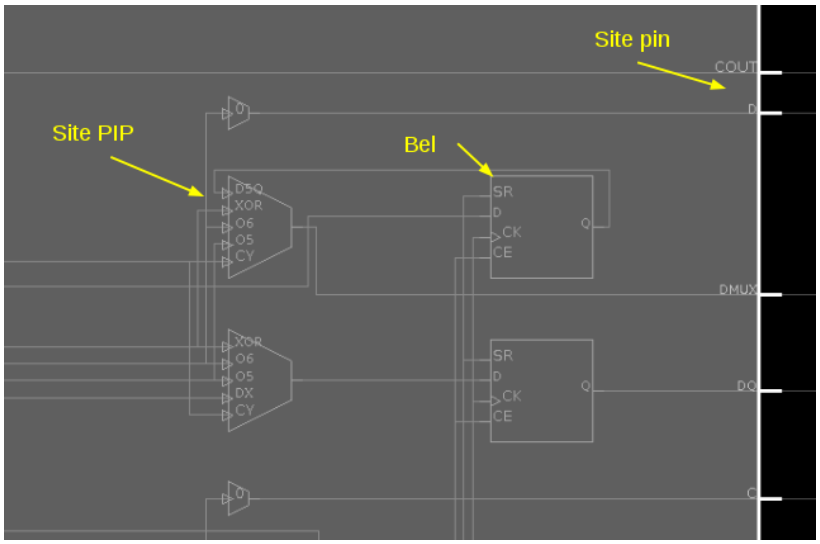
4.1.2 Primitive Sites

Each `Tile` can contain one or more `Primitive Sites` (often shortened to `Site`). As Figure 3 shows, CLBLL `Tiles` have two `Sites`, INT_L `Tiles` have one `Site`, and VBRK `Tiles` have none. A `Site` is the part within a `Tile` that actually performs its "useful" function. The remainder of the `Tile` is used to wire signals to/from its `Sites`. Figure 4 shows a SLICEL, one of the `Sites` within a CLBLL `Tile`. As the figure shows, there are three main components to `Sites` in Xilinx FPGAs:

- `Site PIPs`: Reconfigurable routing PIPS used for intrasite routing (also called routing muxes). In Vivado, `Site PIPs` are usually configured automatically as cells in a design are being placed (based on cell properties and placement location).
- `BELs`: **B**asic **EL**ements are hardware components within the `Site` for implementing digital logic. For example, LUT `BELs` within a SLICEL are used to implement logic equations, and Flip Flop `BELs` are used as storage. In a synthesized netlist, design elements are mapped to physical `BELs` during implementation.
- `Site Pins`: `Site` input and output. These pins are connected to `Wires` of the parent `Tile` and typically drive general fabric.



(a)



(b)

Figure 4: (a) A highlighted example of a SLICEL `Site` in a CLBLL `Tile` of an Artix7 FPGA. (b) The basic components of a `Site` in a Xilinx FPGA.

4.1.3 Wires and PIPs

FPGA components are connected together using metal `Wires` (called `Nodes` in Vivado). In order to make the FPGA reconfigurable, `Wires` are connected together through Programmable Interconnect Points (`PIPs`). Individual `PIPs` can be enabled or disabled as a design is being routed, and a string of enabled `PIPs` uniquely identify the used `Wires` of a physical route. `PIPs` are most commonly found in Switchbox `Tiles`, and enable a single wire to be routed to many different locations in the FPGA. Figure 5 shows an example of a switchbox.

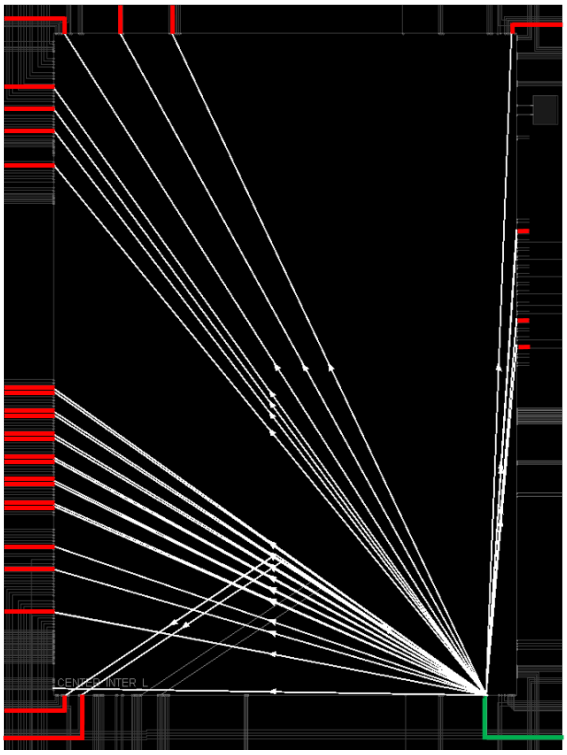


Figure 5: An example of PIP wire connections in a device. The green wire represents the source wire, and the red wires represent all possible sink wires in the Switchbox. The highlighed white sections of the figure are PIP connections.

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 components were unknown. RapidSmith 2 extends the `Device` architecture to include all components **within** a `Site` as well. Figure 6 shows the new data structure hierarchy, which can be found in the *edu.byu.ece.rapidSmith.device* package.

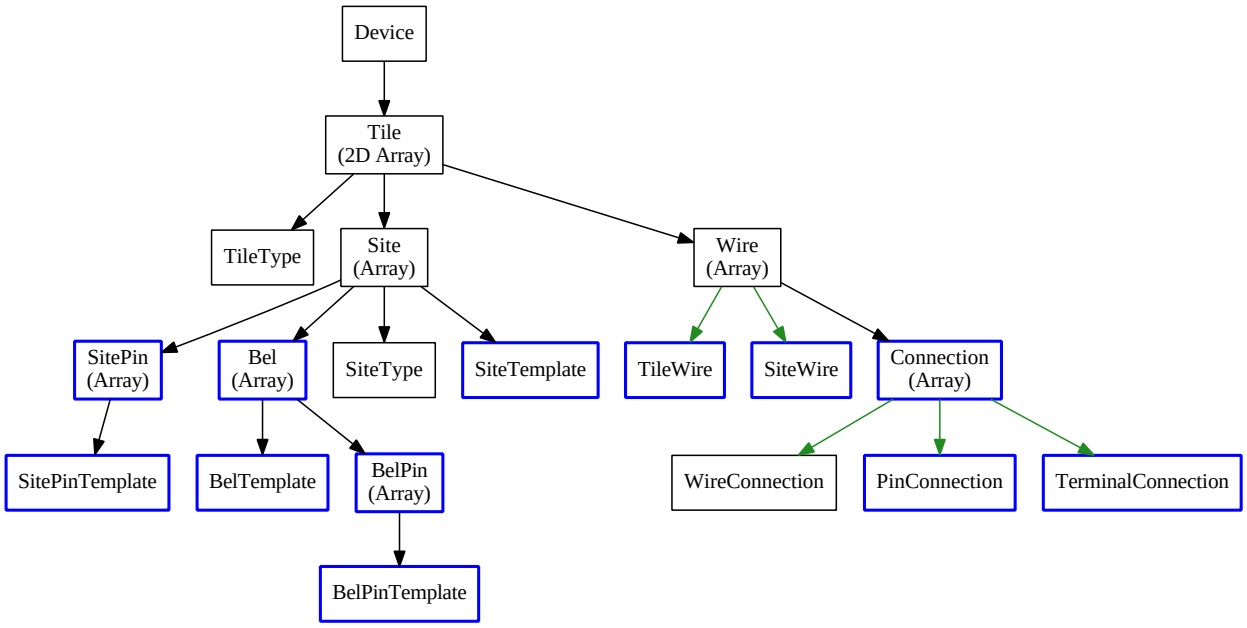


Figure 6: RapidSmith2 `Device` data structure tree. Green arrows represent inheritance, and black arrows represent association. Classes and Interfaces bolded in blue are new to RapidSmith 2.

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 RapidSmith (such as a `Tile`). Because of this, most RapidSmith data structures represent a straightforward part of a Xilinx FPGA, The reader is referred to the documentation in the source code to learn more about each data structure. Also, The **DeviceBrowser** and **DeviceAnalyzer** programs described in section 9 illustrate how to load and browse a device down to the `Tile` and `Site` levels.

4.2.1 Templates

As Figure 6 shows, there are several template classes in RapidSmith. Template classes are used to specify the configuration of certain structures only once, and then reuse the configuration 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 `BEL`s, `Site Pin`s, and `Site PIP`s combined. In order to save memory, RapidSmith lazily creates these objects only when a SLICEL `Site` is being used. The alternative would be to create each of the objects when a device is loaded. Template classes should **NOT** be used by the normal user. When creating algorithms using RapidSmith's API, use the non-template version of classes.

4.2.2 WireEnumerator

Wires with the same name can occur several times throughout a Xilinx FPGA device. For example, the wire "CLBLL_L_C2" exists in every `Tile` of type "CLBLL_L". In order to make the device files small, each uniquely named `Wire` is assigned an integer enumeration. This avoids moving strings around in memory which would be costly in terms of both space and comparison times. RapidSmith manages all uniquely named wires in an FPGA family with a `WireEnumerator`. The `WireEnumerator` class has methods that convert to and from the wire name and enumeration of a `Wire`, and also stores other `Wire` information such as direction and type. In previous versions of RapidSmith, the user had to use the `WireEnumerator` extensively while building CAD tools. RapidSmith 2 has changed this, largely abstracting the `WireEnumerator` away in favor of more convenient methods in all classes and interfaces that deal with `Wires`. For example, the name or enumeration of a `Wire` can now be obtained with the function calls *Wire.getWireName()* and *Wire.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 RapidSmith are uniquely identified not only by their name (or enumeration), but also by the `Tile` or `Site` in which they exist. RapidSmith 2 introduces the `TileWire` and `SiteWire` classes to encapsulate this information for the user. Many functions in RapidSmith now return a `TileWire` or `SiteWire` (wrapped in a generic `Wire` object) to the user instead of an integer wire enumeration.

4.2.4 Wire and Connection Interfaces

As Figure 6 shows, there are two classes that inherit the `Wire` interface (described in subsubsection 4.2.3) and three classes that inherit the `Connection` interface (described in subsection 7.3). In general, most methods in RapidSmith return the generic interface instead of the subclasses. When creating CAD tools in RapidSmith, it is suggested that the user use these interfaces. Several of the examples referenced in section 9 demonstrate the use of these interfaces.

4.3 Device Files

The `Device` data structures in RapidSmith are created from XDLRC files. For older Xilinx parts (series 7 and below), these files can be created in ISE with the *xdl* command. For newer parts (ultrascale and above), XDLRC files can be created in Vivado using the TINCR command *tincr::write_xdlrc*. Because XDLRC files can grow to be hundreds of Gigabytes in size, RapidSmith compresses them into much smaller device files (usually in the tens of Megabytes) and stores those instead. This section gives a brief overview of the syntax of XDLRC files, and how to install new device files in RapidSmith from both ISE and Vivado.

4.3.1 Basic Syntax of XDLRC files

In general, users of RapidSmith do not need to understand the syntax of XDLRC files to create CAD tools in RapidSmith. 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 7: 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 8: 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 RapidSmith, only the first pin name (i.e. BX above) is used.

Wire

```
# 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 9: Wire syntax in XDLRC files

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 RapidSmith 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.

PIP

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

Figure 10: 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 RapidSmith.

Primitive Definitions

The Primitive Definition portion of an XDLRC file textually describes the components found within a `Primitive Site` type (a SLICEL for example) and how they are connected. `BEL`s, `Site Pin`s, `Site PIP`s, configuration options, and route-throughs are the elements that are most commonly found within these parts of the file. An example of a complete primitive definition file of type BUFHCE can be seen in Figure 11. The sub-site data structures in RapidSmith (`Bel`s, `SiteWire`s, etc.) are built by parsing this section of the XDLRC file.

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

Figure 11: Primitive Def sections of XDLRC files

5 Designs in RS2

5.1 Xilinx Netlists

During the synthesis stage of implementation, a digital circuit expressed using RTL (VHDL or Verilog) is translated to a lower level Xilinx netlist. This netlist describes a digital circuit in terms of primitive elements that can directly target hardware on a Xilinx FPGA. In terms of granularity, a Xilinx netlist is more abstract than gates and transistors, but more detailed than RTL. A list of valid primitives (also called `Cells`) that can be used within a Xilinx netlist can be found here for 7 Series devices and here for Ultrascale devices. It is suggested that users of RapidSmith become familiar with the primitives found in these documents, and how they are used. The primitives of a Xilinx netlist are wired together to create a digital circuit (such as an 32-bit adder) capable of being implemented on an FPGA. Figure 12 shows an example netlist of a 3-bit counter that has been synthesized in Vivado.

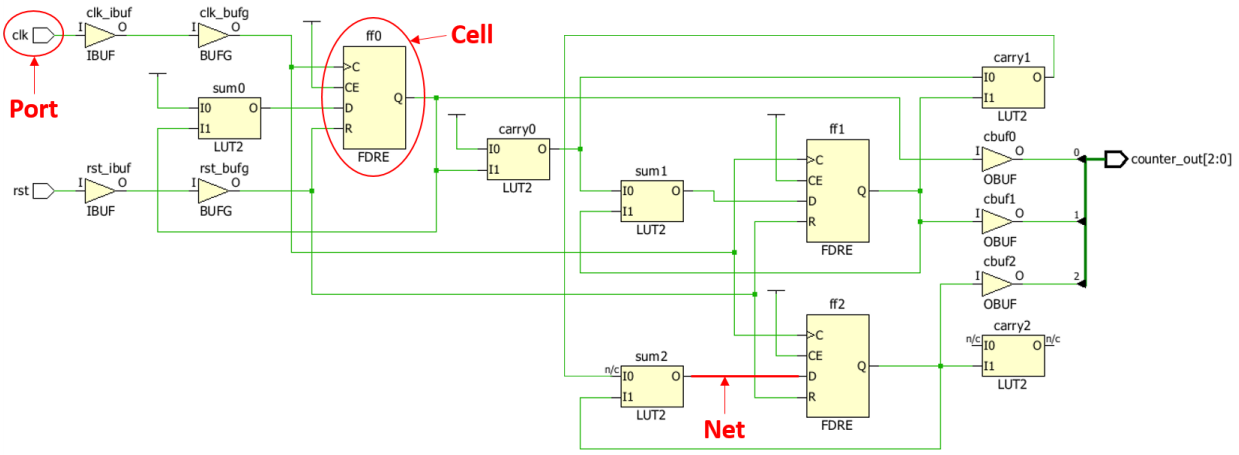


Figure 12: Schematic of a 3-bit counter in Vivado using LUT and FDRE cells. The yellow boxes are `Cells`, the green lines are `Nets`, and and the white figures on the edge of the diagram are `Ports`. An example of each of these netlist components have been highlighted.

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 12 uses an 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 signal (replace xx with the corresponding letters).
- **Block Ram** (BRAMs): On-chip FPGA memory cell.
- **Digital Signal Processing Units** (DSPs): Perform complex arithmetic functions efficiently.
- **Buffers** (BUF): IO, clock, and other types of signal buffers.

There exists several other types of `Cells`, but the ones in the list above are the most prevalent. `Nets` are used to 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 an FPGA design, `Ports` are connected 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 not 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 EDIF. In order to recreate the design in RapidSmith, the EDIF file is parsed and converted to a custom netlist structure described in the next section.

5.2 RS2 Netlist Data Structures

RapidSmith 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 RapidSmith straightforward. The data structures that constitute a RapidSmith netlist can be found in the package `byu.edu.ece.rapidSmith.design.subsite`. The package hierarchy can be seen in Figure 13.

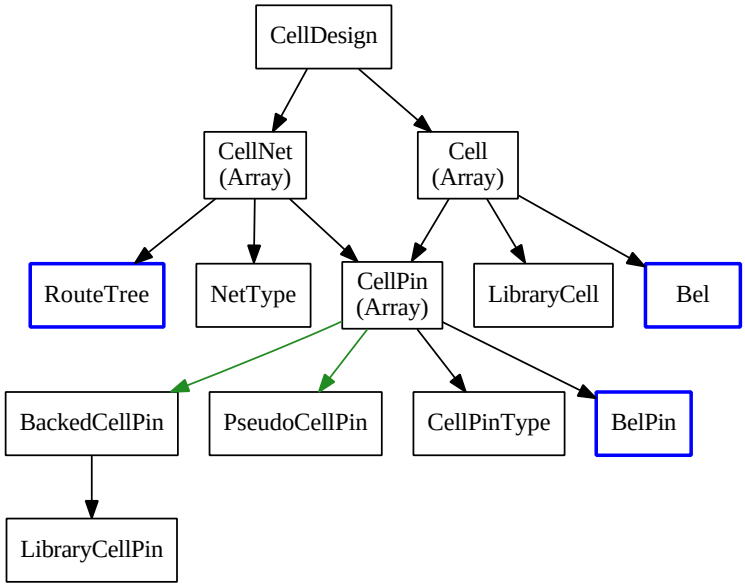


Figure 13: RapidSmith 2 design data structure tree. Black arrows represent composition (i.e. a `CellDesign` contains an array of `CellNets` and an array of `Cells`). Green arrows represent inheritance (i.e. a `CellPin` can be either a `BackedCellPin` or a `PseudoCellPin`). Blue boxes represent physical implementation components of the netlist (i.e. what physical `Bel` a `Cell` is placed on, or what physical `Wires` are used in a `CellNet`).

A `CellDesign` is the top level design object in RapidSmith. It consists of a collection of `Cell` objects, interconnected by `CellNets`. RapidSmith `Cells` are equivalent to Xilinx `Cells` and RapidSmith `CellNets` are equivalent to Xilinx `Nets`. `Cell` objects have a template `LibraryCell`, which represents a Xilinx primitive. They also have a collection of `CellPins`, which can be thought of as `Cell` IO. During the placement phase of implementation, `Cells` may be mapped onto `BELs` and the corresponding `CellPins` mapped onto `BelPins`. Placement is described in more detail in section 6. `CellNet` objects connect `Cells` together via their `CellPins`. During the routing phase of implementation, a `CellNet` maps onto one or more `RouteTrees`. Routing is described in more detail in section 7. The best way to learn how to use these classes is to 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 structure. A Vivado design is converted from a Tincr checkpoint to a `CellDesign` on import, and converted from a `CellDesign` to a Tincr checkpoint on design export. It contains the following:

- A list of uniquely named `Cells` in the design.
- A list of uniquely named `CellNets` in the design
- Global GND and VCC `CellNets`
- `Cell` placement information (where each cell is placed)
- Internal routing configuration for each **used** `Site` (in the form of `Site PIPs`)

- A list of XDC constraints imported from Vivado. See section 8 for more information about XDC constraints and how they are represented in RapidSmith.

The `CellDesign` class has a variety of methods to retrieve and manipulate the `Cells` and `CellNets` of a design, place `Cells` onto physical `BELs`, configure sub-site routing, and perform several other tasks. See the Javadocs for the exact API.

5.2.2 `Cell`

This section contains a few things you should know about `Cells` in, no particular order.

- A `Cell` always contains a reference to a backing `LibraryCell`. 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. The method call `CellDesign.getCellsOfType(String,CellLibrary)` can be used to get all `Cells` in the current design with a specific `LibraryCell` type.
- A `Cell` has a list of `CellPins` that can connect to `CellNets`. The methods `Cell.getPins()`, `Cell.getInputPins()`, and `Cell.getOutputPins()` can be used to get a handle to the pins of a `Cell`. If more pins are needed on a `Cell`, `PseudoCellPin` objects can be attached (see subsubsection 5.2.3).
- `Cells` can be placed onto `BELs` 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 RapidSmith. Specifically, there are three types of port cells: IPORT, OPORT, and IOPORT. The method `CellDesign.getPorts()` can be used to iterate through the port `Cells` in a design, and the method `Cell.isPort()` can be used to determine if a given `Cell` is actually a port.

5.2.3 `CellPin`

There are two types of `CellPin` objects in RapidSmith: (1) `BackedCellPins` and (2) `PseudoCellPins`. `BackedCellPins` are straightforward. They represent a `LibraryCellPin`, which is a pin attached to the corresponding `LibraryCell` of a `Cell`. These pins are used as signal input and signal output. On the other hand, `PseudoCellPins` are "fake" `CellPin` objects which are added to a `Cell` after the `Cell` has been created. They don't affect the functionality of the `Cell`, but are needed for physical implementation. An example of where a `PseudoCellPin` is needed can be seen in Figure 14.

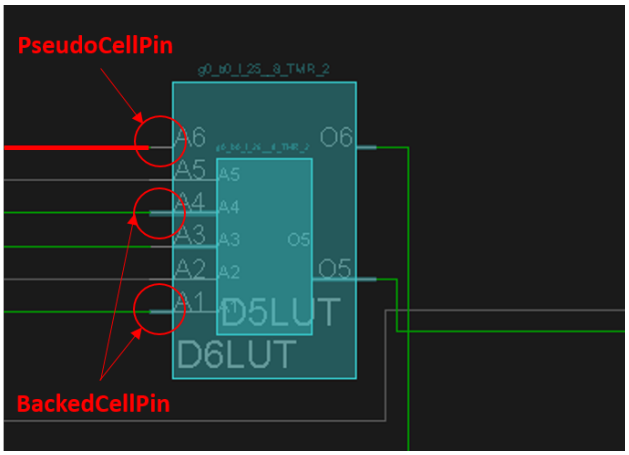


Figure 14: An example where a `PseudoCellPin` in RapidSmith may be needed. Both a `PseudoCellPin` and `BackedCellPin` are pointed out. The red net in the image (going into the A6 pin) is the power net (VCC).

In the figure, a cell of type LUT2 has been placed onto a D6LUT BEL of a SLICEL. The two CellPins of the LUT2 have been mapped to the A1 and A4 pins of the BEL (which have been highlighted in blue), connected to Nets, and routed. This is expected behavior based on the netlist. Also, the global VCC net has been routed to the A6 BelPin. This is necessary due to the hardware implementation of LUTs. However, in the netlist view of the circuit, the fact that VCC needs to be routed to the A6 pin is not at all represented (a CellNet can only be connected to CellPins). This is where PseudoCellPins are helpful. In RapidSmith, a PseudoCellPin can be added to the LUT2 cell (called "VCCPin"), mapped to the A6 BelPin, and attached to the global VCC CellNet. This gives a more complete view of how the netlist needs to be physically implemented. PseudoCellPins are created and added automatically to Cells when importing a Tincr checkpoint. If you are creating Cells from scratch, a PseudoCellPin can be added with the method call Cell.attachPseudoPin(string,PinDirection). To determine if a CellPin is a PseudoCellPin, the method call CellPin.isPseudoPin() can be used. That being said, in general **users do not need to worry about the different between pseudo and backed CellPins** when creating a CAD algorithm. They function identically, so you can use the higher level handles to the CellPin interface instead.

Each CellPin object also has a CellPinType and PinDirection enumeration. Table 1 displays the possible values for both of these enumerations. The CellPinType field can be used to find all RESET CellPins in a design, determine if a CellNet is a clock net (it connects to pins of type CLOCK), and other useful functions. The PinDirection field is typically used to filter a list of CellPins by their direction. It it especially useful in determining if a pin on a Cell is of type INOUT.

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

Table 1: Enumerations for the CellPin class

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. The remainder of this section details important aspects about CellNets in RapidSmith.

- CellNets are routed using the RouteTree data structure. Routing in RapidSmith 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. Other NetTypes could be added to this list in the future (such as CLOCK), but for now they are not differentiated.

- VCC and GND nets (also called static nets) are treated specially in RapidSmith in two ways. (1) Vivado designs can have multiple nets that correspond to VCC or GND, but they all logically represent the same thing. This representation can be challenging to work with when designing CAD tools. RapidSmith takes a different approach. When a design is imported from Vivado, all VCC nets are collapsed into a single VCC net that is sourced by a global VCC cell (the same applies to GND nets). We have found this representation much easier to work with. The function calls `CellDesign.getVccNet()` and `CellDesign.getGndNet()` can be used to retrieve these special nets from the design. If you add a VCC net to a design that already has a VCC net, the old net will be overridden. (2) When a static net is routed, it doesn't have a single source driving it. Rather, the net is partitioned into sections with each section having a single source. In an FPGA, each of these sources corresponds to a TIEOFF within a switchbox tile. This is shown in Figure 15.

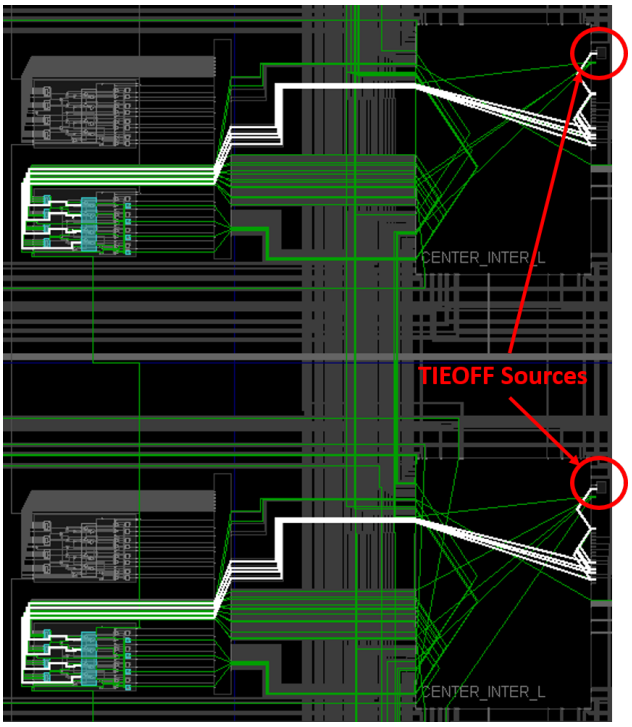


Figure 15: Static net multiple sources example. The higlighted white wires are a part of the same GND net, and the circled red areas are the source TIEOFFs for the GND net. In this image, only two sources are shown.

  RapidSmith handles this oddity by allowing `CellNets` to have more than one RouteTree object associated with it. In the case of Figure 15 the GND net would have two `RouteTree` objects associated with it, one for each source TIEOFF.

- Most `CellNets` have a single source pin and multiple sink pins (referred to as the "fanout" of the net). When routing a net that fits this description, only one RouteTree needs to be specified. However, there are two noticable exceptions to the common case: (1) static nets (described in the bullet point above) and (2) nets with multiple drivers (involving IO pins). In a Xilinx FPGA, this is most often seen in IOB pads. An example is shown in Figure 16. The highlighted net can be driven by both the OBUF output, and from an external source. RapidSmith 2 supports nets with multiple drivers. The function call `CellNet.getAllSources()` can be used to retrieve all possible source `CellPins` of a net.

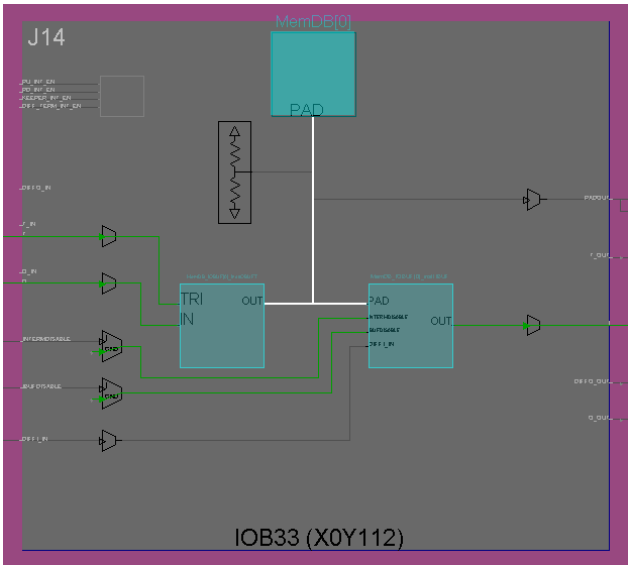


Figure 16: An example of net (highlighted in white) that can be driven by more than once source.

- After a design has been placed, `CellNets` fall into one of two categories: INTRASITE or INTERSITE. Figure 17 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`. These types of nets are handled differently during routing. To determine if a `CellNet` is an INTRASITE net, the method call `CellNet.isIntrasite()` can be used.

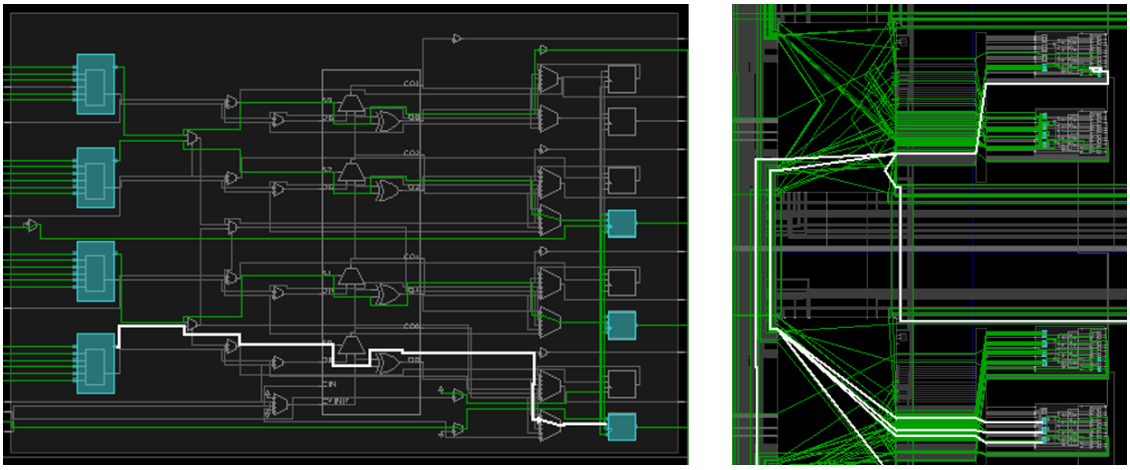


Figure 17: Example of an INTRASITE net (left) and an INTERSITE net (right).

5.2.5 `PropertyList`

In Vivado the top-level design, cells, and nets all have properties associated with them. The Tcl command `[report_property $object]` can be used to view the properties of a Vivado Tcl object. When a design is exported from Vivado, these properties (**if they are not the default value**) are stored in the EDIF netlist file. When RapidSmith parses the EDIF file, the properties are stored in a data structure called a `PropertyList`. Each `CellDesign`, `Cell`, and `CellNet` in RapidSmith has an associated `PropertyList` object, which can be retrieved using the method `getProperties()`. `PropertyList` implements the `Iterable` interface, so properties of an object can be iterated through easily using a for-loop. Also, properties can be retrieved by their string name. Because EDIF properties only support String, Integer, and Boolean types, any properties imported from Vivado will have one of these

types.

Cells are the most interesting objects in terms of properties. This is 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. Possible values include 1, 2, 4, 9, 18, 36 and 72. Another example is a D flip flop cell (FDRE) and its "CONFIG.INIT" property. This property indicates what the flip flops power-up state should be. Some additional things about cell properties that may be helpful include:

- To learn what configuration properties a Cell has, open up a Vivado design and type the following into the console,

```
Vivado% link_design -part xc7a100tcsg324-3 -quiet
Vivado% set lib_cell [get_lib_cells RAMB36E1]
Vivado% report_property $lib_cell
```

  replacing "xc7a100tcsg324-3" with the part you are using and "RAMB36E1" with your cell of interest. All properties that start with "CONFIG" represent configuration properties that a user can set.

- To learn what configuration properties **affect the pin mappings of a cell**, open the *cellLibrary.xml* file you have, and search for the tag "libcellproperty." Pin mappings are described in more detail in section 6.

- Only properties of type EDIF will be exported from RapidSmith. If you create cells in your design and add properties to them, make sure that you mark the properties as EDIF properties by setting their type. All other properties will be ignored.

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

5.3 The Cell Library

The CellLibrary in RapidSmith stores important information about each of the LibraryCells compatible with a specific Xilinx FPGA part. This includes:

- The name of the library cell
- The name, direction, and type of each library cell pin
- Valid placement locations for instances of the library cell
- Possible logical-to-physical pin mappings for every library pin
- Configuration properties of the library cell (TODO)

When doing any type of netlist modification, the CellLibrary is needed. Currently, each CellLibrary corresponds to a specific Xilinx part. This means that for each part you use in RapidSmith, a new CellLibrary needs to be generated [2]. The rest of this section demonstrates how to get access to a cell library in RapidSMith, and how to create a new cell library XML file from Tincr.

[2]This may change to be family-specific in the future. Usually, parts in the same family can use the same CellLibrary. The part-specific mention is very conservative, but is one that we know is right

5.3.1 Getting a handle to the Cell Library

When a Tincr checkpoint is imported into RapidSmith, a `CellLibrary` is automatically loaded. To get a handle to the `CellLibrary`, you can use the method call `TincrCheckpoint.getLibCells()` on the Tincr checkpoint object that was loaded. If you only have a handle to a `Device` object, then the code below can be used to load a cell library from disk:

```
CellLibrary libCells = new CellLibrary(RSEnvironment.defaultEnv()
                       .getPartFolderPath(device.getPartName())
                       .resolve(CELL_LIBRARY_NAME));
```

5.3.2 Generating a new Cell Library

During runtime, a `CellLibrary` data structure is created by parsing a *cellLibrary.xml* file into memory. So, to create a new `CellLibrary` all you have to do is generate a new *cellLibrary.xml* file for the Xilinx part you are interested in. This can be done in a few easy steps.

1. Open Vivado in Tcl mode, and type the following into the console:

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

   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.

2. Open the generated XML file in a text editor and search for the "CARRY" cell (could be the CARRY4 or CARRY8).

3. Scroll down to the "bels" XML element within the CARRY cell, and add the following lines to each pin that is named "CI":

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

   If the cell is a CARRY4, you should have to insert this line in two places only.

4. Save your changes and exit the text editor

5. Copy the XML file to *$RapidSmith_Path/device/family* directory where "family" is replaced by the family of your part (such as Artix7), and "$RapidSmith_Path" is your RapidSmith repository location. Once this is done the new `CellLibrary` should be ready to use.

NOTE: Usually, parts in the same family can use the same `CellLibrary`.

6 Placement in RS2

In the original RapidSmith, placement occurred at the `Site` level. A collection of `Cells` were grouped together in what was called an `Instance`, and the `Instance` was assigned to a compatible site type. Where the individual `Cells` were actually placed within the `Site` was a blackbox. Because RapidSmith 2 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 grain control during the placement of a design, and allows sub-site algorithms (such as packing algorithms) to be explored. Code Sample 1 demonstrates the basic steps to placing a `Cell` in RapidSmith.

Code Sample 1: Steps for placing a Cell in RapidSmith

```
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 RapidSmith `Cell`. The first is straightforward: get a handle to a `Bel` object, and use the method call `CellDesign.placeCell(Cell, Bel)` (line 11). This will map a logical `Cell` onto a physical `Bel`. Once a `Bel` has been used, no other `Cells` can be mapped to it. No error checking is performed to ensure that a `Cell` is compatible with the `Bel` it is being placed on, that is up to the user. The second step is more complicated: each logical `CellPin` of the `Cell` needs to be mapped to a physical `BelPin`. This can be done by either (a) specifying the `BelPin` name (lines 19-20), or (b) using the function call `CellPin.getPossibleBelPins(Bel)` (line 23) to get a list of valid `BelPin` mappings. For most `Cells` each `CellPin` only maps to a single `BelPin`, and so option (b) is preferable. However, there are two noticable exceptions to this rule.

1. LUT input pins are permutable. This means that an input `CellPin` attached to a LUT `Cell` can be mapped to any input pin of a LUT `Bel`. Figure 18 shows an example of this functionality in Vivado. Once the input pins have been mapped, a configuration equation is created for and applied to the LUT `Bel` using the appropriate `BelPin` inputs. In order for a LUT `Cell` to be physically implemented, these pins need to be mapped. The function call `CellPin.getPossibleBelPins(Bel)` in this case will return all of the input `BelPins` 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`, the least-significant data input pins map to different physical `BelPins` when the width of the BRAM is set to 72. This is an important concept when performing netlist modifications in RapidSmith. The function call `CellPin.getPossibleBelPins(Bel)` only returns the pin mappings for the **default** `Cell` configuration. The correct pin mappings are always used when a Tincr checkpoint is imported into RapidSmith, but if

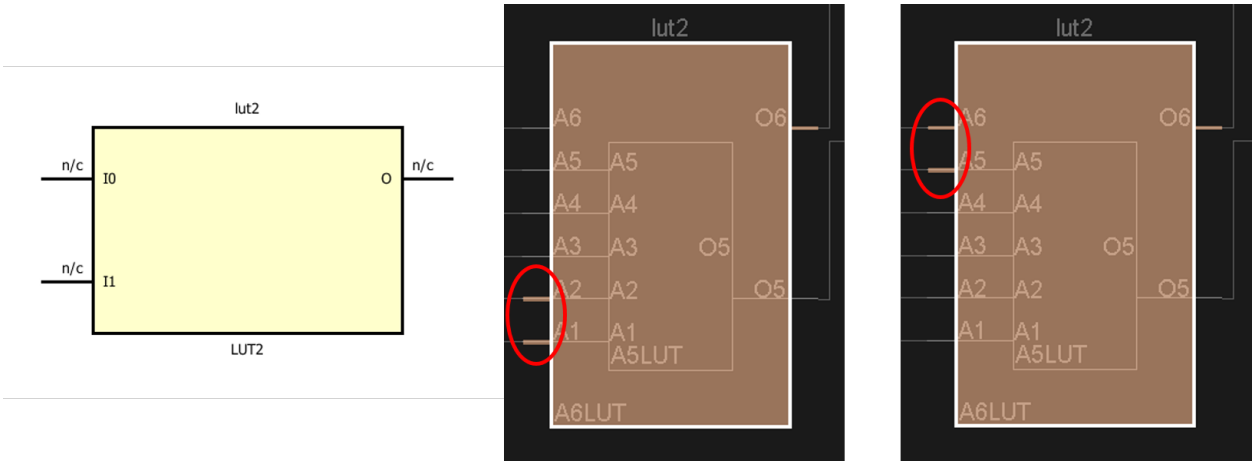


Figure 18: An example of LUT pin permutations. The left figure shows the schematic of a LUT2 cell. The middle figure shows the two input pins of the LUT cell being mapped to the A1 and A2 pins of a LUT6 bel. The right figure shows the same input pins being mapped to the A6 and A5 pins instead.

new logic is being added to a design it is up to the user to determine the proper pin mappings. Adding support for automatically determining a `Cells` pin mappings based on a its configuration is currently being worked on, but is not yet ready. For now, users can configure a cell in Vivado, place it, and use the TCL function shown in Code Sample 2 to determine the correct pin mappings for a `Cell`.

Code Sample 2: TCL script to print all logical-to-physical pin mappings of a cell

```
1
2  proc print_pin_mappings{cell} {
3
4    foreach cell_pin [get_pins -of $cell] {
5      puts "$cell_pin -> [get_bel_pins -of $cell_pin]"
6    }
7  }
```

Some additional points about placement to be aware of include:

- VCC and GND `Cells` are not placed when implementing a design in Vivado. This distinction is applied to RapidSmith 2 as well. Rather than placing VCC or GND explictly, `RouteTrees` that are sourced by switchbox TIEOFFs are used to express their placement implicitly (VCC/GND is "placed" on the TIEOFF).
- A list of valid placement options for a `Cell` can be obtained with the function call `Cell.getPossibleAnchors()`. The sample program **CreateDesignExample** (described in subsection 9.6) gives a good example of how to use this function. The reader is referred to the source code and Javadocs for more details.
- If you plan on writing a placer in RapidSmith, there are several placement rules for a given device. A few examples include (a) CARRY4 `Cells` that are connected through the carry chain need to placed vertically to one another so they can use the dedicated carry wires, and (b) 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. It is the responsibility of the user to determine all relevant placement rules because error checking is not performed on design export. The source code for a sample simulated annealing placer for the Artix-7 part *xc7a100tcsg324* can be found in the package `edu.byu.ece.rapidSmith.examples.placerDemo`.

7 Routing in RS2

After placement, all `Cells` of a design have been mapped to `Bels`, and their corresponding `CellPins` mapped to `BelPins`. The next (and final) step of the FPGA implementation flow is to physically wire together the used `BelPins`. This is known as routing. Routing involves taking each logical `Net` of a design, determining the `BelPins` they are connected to (based on the `CellPins`), and finding a list of physical wires that electrically connect the pins together. This section details how routing is done in RS2.

7.1 Wires and WireConnections

Routing in RapidSmith is done using `Wire` objects, which are described in subsubsection 4.1.3. `Wires` are uniquely identified by their corresponding tile and wire name (i.e. tile1/wire1), and are connected through `WireConnection` 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 an FPGA part (as shown in Figure 5). These types of connections are important to FPGA routing, because they dynamically configure the routing network for a given design.
2. **Non-PIP Connections**: Connect the same physical wire across two different tiles. In general, wires stretch across multiple tiles in a device, having a different name in each tile. This is demonstrated in Figure 19. The example wire shown in the figure spans 5 different tiles, but has a different name in each. To save space, only the source and sink wire segments are kept in RapidSmith data structures (`INT_X1Y1/E2BEG4`, `INT_X2Y1/E2MID4`, and `INT_X3Y1/E2END4`). The source segment is connected to each sink segment through a non-PIP wire connection. It is also possible to have non-PIP connections within a `Tile`, but this is rare.

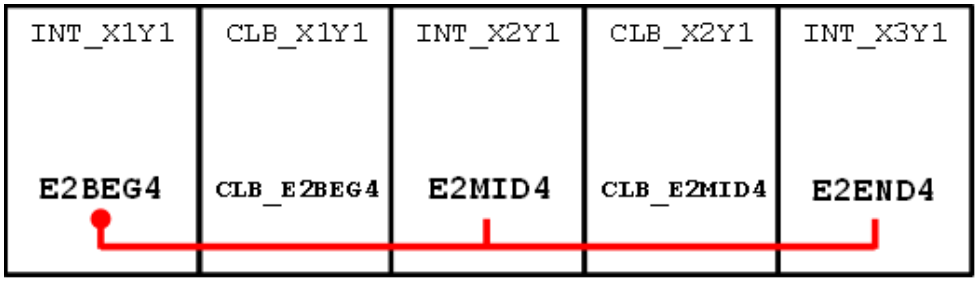


Figure 19: A wire in an FPGA illustrating how each part of the wire has a different name depending on the tile it is located in

7.2 Traversing Wire Objects

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

- `mywire.getWireConnections()`: Returns a collection of all `WireConnections` with the source wire of mywire. This collection can be iterated through to find all places a specific wire goes (i.e. what wires it connects to).
- `wc.isPip()`: Returns true if the wire connection “wc” is a PIP connection. Returns false otherwise.
- `wc.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 has to be either (a) created using a `TileWire` constructor, or (b) retrieved from a function call of another object (such as `SitePin::getExternalWire()`). Code Sample 3 demonstrates how to iterate through `WireConnection` objects. To gain a better understanding of how to use `Wires` and `WireConnections`, see the **HandRouter** example in the RapidSmith repository.

7.3 Other Types of Connections

Along with PIP and non-PIP wire connections, there are several other types of connections in RapidSmith. 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 `conn.getSitePin()` can be used to get the sink `SitePin` object.
- **Terminal Connections**: Connects a `Wire` to a `BelPin`. The function call `conn.getBelPin()` can be used to get the sink `BelPin` object.
- **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 `SitePin` directly to an output `SitePin`. These connections are represented as routethroughs in RapidSmith and can be determined with the function call `conn.isRoutethrough()`. **NOTE**: Before using this type of connection when building a routing data structure, make sure the `Site` is unused.
- **BEL Routethrough Connections**: Connects an input BEL `Wire` to an output BEL `Wire`. Similarly to a `Site` routethrough, LUTs in Vivado can also be configured as routethroughs. In this case, the signal on one of the input pins of the LUT is passed directly to the output pin of the LUT. A BEL routethrough connection can be used to configure a LUT as a routethrough in RapidSmith. **NOTE**: Before using this type of connection, make sure the LUT is unused (i.e. no cell is placed on it).

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. Code Sample 3 demonstrates how to iterate through different `Connection` types in RapidSmith.

Code Sample 3: How to iterate over Connections in RapidSmith

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

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

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

// Iterate over all Pin Connections
for (Connection conn : wire.getPinConnections() ) {
  SitePin pin = conn.getSitePin();
  // Do something with the SitePin
}

// Iterate over all Terminal Connections
for (Connection conn : wire.getTerminals()) {
  BelPin bp = conn.getBelPin();
  // Do something with the BelPin
}
```

```
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 RapidSmith, 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. RapidSmith 2 introduces the `RouteTree`, which can be see in Figure 20.

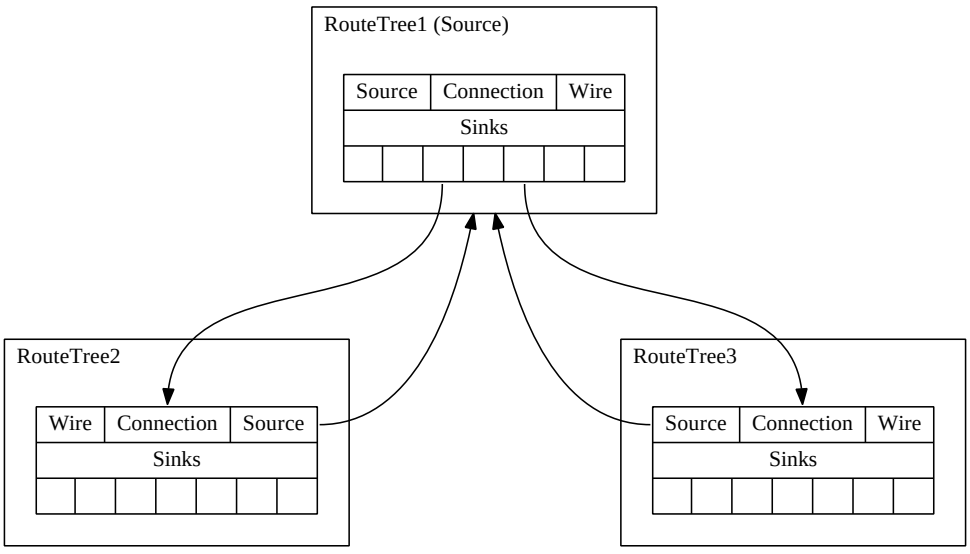


Figure 20: Visual representation of the `RouteTree` data structure.

As the figure shows, a `RouteTree` is a simple tree data structure. Each node in the tree represents a physical wire, 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 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 WireConnection 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 node 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 21 shows an example of a complete RouteTree in RapidSmith.

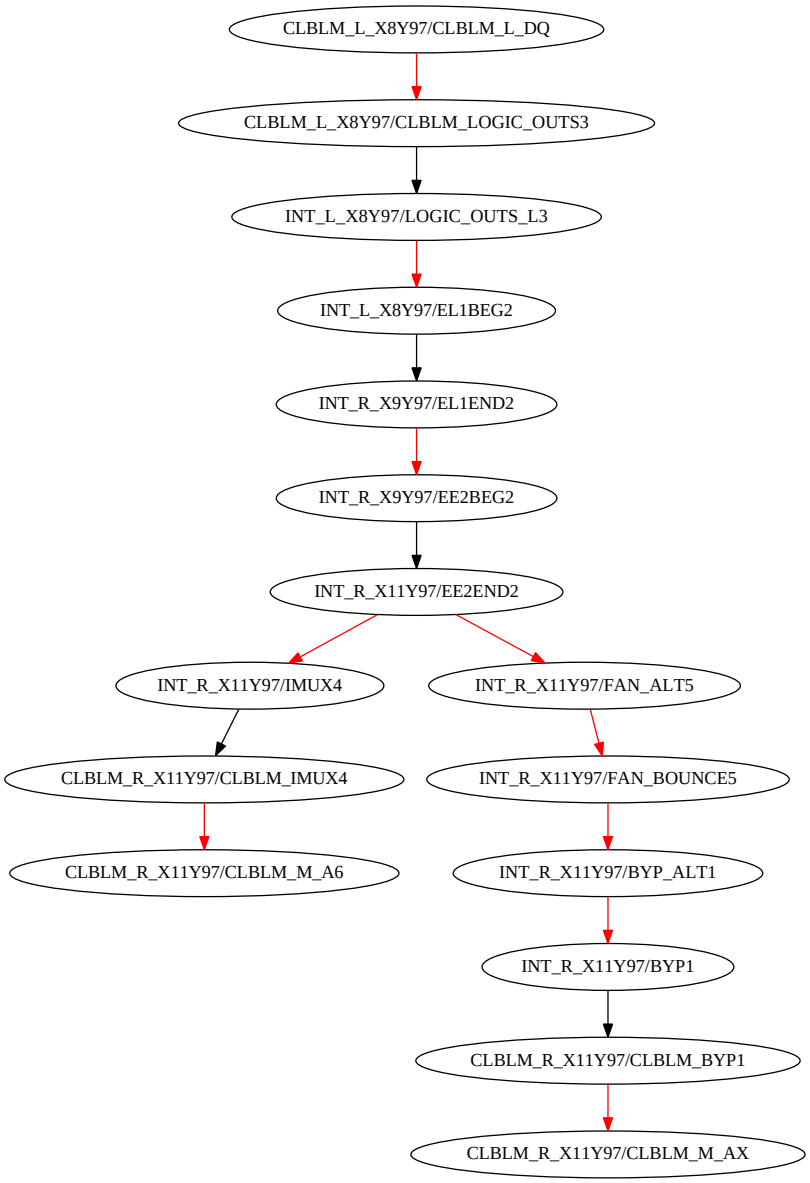


Figure 21: A visual representation of a RapidSmith2 RouteTree. Nodes represent Wires, red edges represent PIP wire connections, and black edges represent non-PIP wire connections. This specific RouteTree was created from the **AStarRouter** example.

In this case, the CellNet that is being routed has once source pin, and two sink pins. The source pin is connected to wire "CLBLM_L_X8Y97/CLBLM_L_DQ", and the sink pins are connected to the wires "CLBLM_R_X1Y97/CLBLM_M_A6" and "CLBLM_R_X11Y97/CLBLM_M_AX". Starting from the source, wires are traversed downward (via wire connections) until the target wires are reached. Code Sample 4 demonstrates the basic usage of RouteTrees in RapidSmith. The **DesignAnalyzer**, **AStarRouter**, and **HandRouter** examples described in section 9 also demonstrate how to traverse and build a RouteTree.

Code Sample 4: Building a RouteTree

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

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

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

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

When a design is imported from Vivado, 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. A user can use custom data structures to route a design, but it needs to be **converted to an equivalent RouteTree data structure** before exporting the design to Vivado.

7.5 Three Part Routing

In RapidSmith 2, there are three sections to a routed CellNet.

1. The portion of the net that starts at the source BelPin, and is routed to an output SitePin. This part of the route exists completely inside of Site boundaries.
2. The portion of the net that starts at the output SitePin of part (1), and is routed to several sink SitePins. This part of the route is called the "INTERSITE" route because it connects Sites together. A typical router is responsible for routing this section of the net[3].
3. The portion of the net that start at an input SitePin from part (2), and is routed to a sink BelPin. Since there can be several sink pins in a CellNet, this section of the net can have more than one component. This part of the route also exists completely inside of Site boundaries.

Figure 22 shows an example of the three parts of a routed CellNet.

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

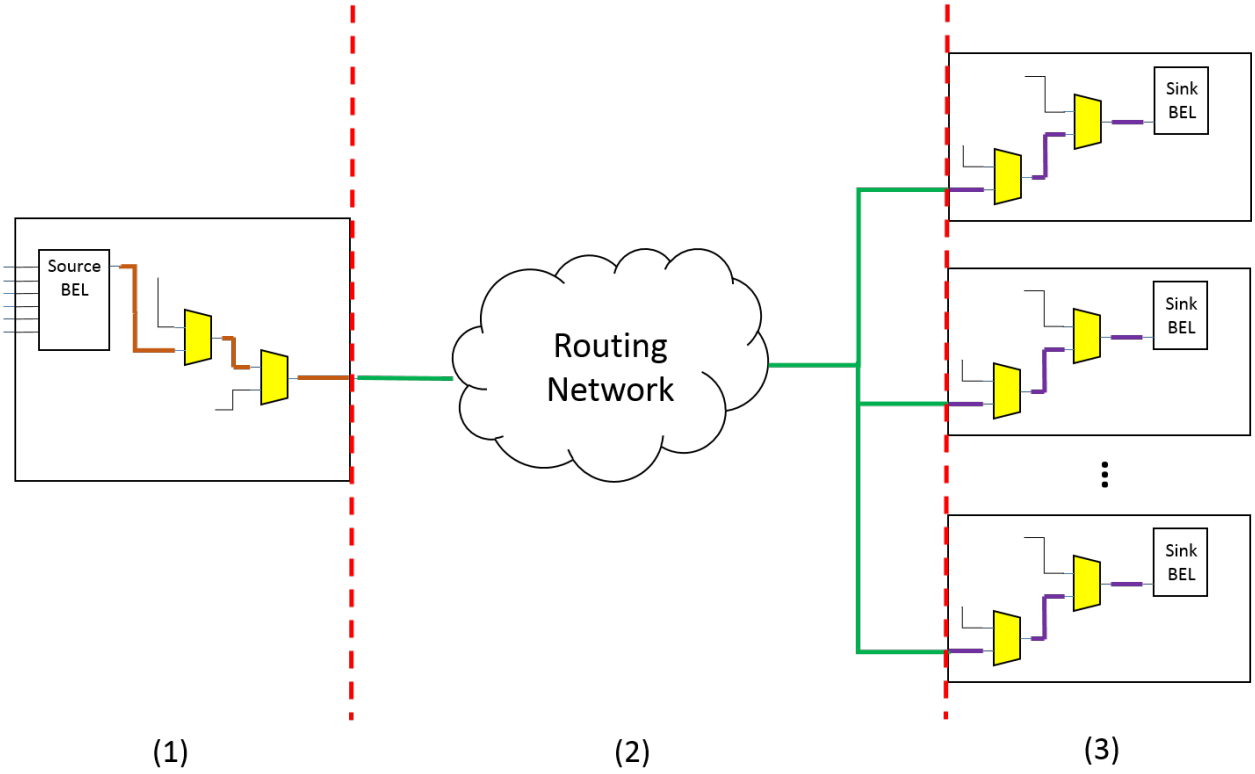


Figure 22: The three sections of a RapidSmith route.

There is a `RouteTree` object associated with each section of the three-part routing. A source `RouteTree` represents the orange wires in the figure above (part 1). An intersite `RouteTree` represents the green wires in the figure above (part 2). A list of sink `RouteTrees` represents the purple wires in the figure above (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`. Code Sample 5 demonstrates how to utilize three part routing in RapidSmith. When a design is imported into RapidSmith, the routing sections of each `CellNet` are created automatically.

Code Sample 5: Three-part routing in RapidSmith

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

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

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

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

// Or, get a RouteTree based on a SitePin
for(SitePin sitePin : net.getSitePins()) {
  if (sitePin.isInput()) {
```

```
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
27  net.addSinkRouteTree(sitePin, createSinkRouteTree(sp));
```

7.6 Routing in Vivado

The reason a three-part routing distinction is necessary in RapidSmith, is due to how routing is represented in Vivado. Inside `Site` boundaries, a route is represented using `SitePIPs`. 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 `SitePIPs`, 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 IntrasiteRouting

On design import, the `SitePIP` information extracted from Vivado is stored into RapidSmith data structures, and used to resconstruct the three-part routing view. This gives the user two options when dealing with intrasite routing in RapidSmith: (1) use the three-part routing data structures, or (2) use the set of enabled `SitePIPs` that exist in each `Site`. It is up to user preference for which representation to use when writing a CAD tool, but in order to export a design back into Vivado the three-part routing data structures need to be converted to `SitePIPs` and assigned to their corresponding `Site`. Code Sample 6 demonstrates how this is currently done. This step needs to be taken **only when you have modified the intrasite routing** for a `Site`. We are working on improving the API for `SitePIPs`, and creating a way to automate conversion from three-part routing to `SitePIP` information.

Code Sample 6: Configuring 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 Importing/Exporting Designs Between Vivado and RS2

Importing and exporting designs between Vivado and RS2 is fairly straightforward. It is done using Tincr Checkpoint (TCP) files, which contain a variety of information about a digital design. This section describes in detail the following:

- The contents of TCP files
- The small differences between TCP files for design import and export
- How to load a TCP into RS2, and how to export a RS2 design to a TCP
- Some additional information that is imported with a TCP

8.1 RapidSmith Checkpoints

RapidSmith checkpoints (RSCP) are Tincr checkpoints generated in Vivado from the command `::tincr::write_rscp`. These checkpoints are capable of representing a Vivado design at any stage of implementation (post-synthesis, post-place, and post-route), and can be parsed and imported into RapidSmith. As we will see in subsection 8.2 below, the format of these checkpoints differ slightly than those of regular TCPs. This is because a great deal of additional information needs to be added to RapidSmith checkpoints in order to import a complete representation of a Vivado design. The remainder of this subsection describes each of the file types within a RapidSmith checkpoint, and what they represent.

8.1.1 design.info

The *design.info* file within a RSCP is used to include any useful information about a design. Currently, it only stores the part name that the design is implemented on. This file is reserved to add more additional information in the future.

8.1.2 netlist.edf

The *netlist.edf* file within a RSCP is an EDIF netlist representing the logical portion of a design. It details all of the `Cells`, `Nets`, and `Ports` within a design, and is generated from Vivado using the Tcl command `write_edif`. A RapidSmith `CellDesign` is created by parsing the EDIF file, and converting it into the appropriate RS2 data structures described in subsection 5.2. Below is an example of a RSCP EDIF file.

```
(Library work
 (edifLevel 0)
 (technology (numberDefinition ))
(cell add (celltype GENERIC)
 (view add (viewtype NETLIST)
  (interface
   (port a (direction INPUT))
   (port b (direction INPUT))
   (port cin (direction INPUT))
   (port cout (direction OUTPUT))
   (port s (direction OUTPUT))
  )
  (contents
   (instance GND (viewref netlist (cellref GND (libraryref hdi_primitives))))
   (instance VCC (viewref netlist (cellref VCC (libraryref hdi_primitives))))
   (instance a_IBUF_inst (viewref netlist (cellref IBUF (libraryref
       hdi_primitives))))
   (instance b_IBUF_inst (viewref netlist (cellref IBUF (libraryref
       hdi_primitives))))
   (instance cin_IBUF_inst (viewref netlist (cellref IBUF (libraryref
       hdi_primitives))))
   (instance cout_OBUF_inst (viewref netlist (cellref OBUF (libraryref
       hdi_primitives))))
```

```
    (instance cout_OBUF_inst_i_1 (viewref netlist (cellref LUT3 (libraryref
       hdi_primitives)))
     (property INIT (string "8'hE8"))
    )
```

8.1.3 placement.rsc

The *placement.rsc* file within a RSCP stores all of the placement information of a Vivado design. This includes which package pin every `Port` is mapped to, which `Bel` each `Cell` is placed on, and the logical-to-physical pin mappings for each `CellPin`. If a design has not yet been placed, this file will be empty. Below is an example of a RSCP placement file.

```
LOC a_IBUF_inst R10 IOB33 INBUF_EN LIOB33_SING_X0Y50
PINMAP a_IBUF_inst O:OUT I:PAD
LOC b_IBUF_inst T10 IOB33 INBUF_EN LIOB33_X0Y51
PINMAP b_IBUF_inst O:OUT I:PAD
LOC cin_IBUF_inst T9 IOB33 INBUF_EN LIOB33_X0Y51
PINMAP cin_IBUF_inst O:OUT I:PAD
LOC cout_OBUF_inst U13 IOB33 OUTBUF LIOB33_X0Y53
PINMAP cout_OBUF_inst O:OUT I:IN
LOC cout_OBUF_inst_i_1 SLICE_X0Y51 SLICEL A6LUT CLBLL_L_X2Y51
PINMAP cout_OBUF_inst_i_1 O:O6 I0:A4 I1:A5 I2:A6
LOC s_OBUF_inst T13 IOB33 OUTBUF LIOB33_X0Y53
PINMAP s_OBUF_inst O:OUT I:IN T:TRI
PACKAGE_PIN R10 a
PACKAGE_PIN T10 b
PACKAGE_PIN T9 cin
PACKAGE_PIN U13 cout
PACKAGE_PIN T13 s
```

8.1.4 routing.rsc

The *routing.rsc* file within a RSCP stores all of the routing information of a Vivado design. This includes:

- The used `Site PIPs` within each `Site` (which specifies the internal routing)
- A list of LUT `Bels` that are acting as static sources (see subsection 8.3 for more details)
- A list of LUT `Bels` that are being used as routethroughs (see subsection 8.3 for more details)
- The name of each INTRASITE net
- The name, connecting `SitePins`, and used `TileWires` of each INTERSITE net.
- Special routing information for the VCC and GND nets.

If Vivado has not yet been routed, this file will be mostly empty. Below is an example of the contents within a RSCP routing file.

```
SITE_PIPS SLICE_X9Y80 SRUSEDMUX:0 CEUSEDMUX:IN COUTUSED:0 CLKINV:CLK DCY0:DX ...
SITE_PIPS SLICE_X13Y80 PRECYINIT:AX SRUSEDMUX:0 CEUSEDMUX:IN COUTUSED:0 ...
SITE_PIPS SLICE_X15Y80 SRUSEDMUX:0 CEUSEDMUX:IN COUTUSED:0 CLKINV:CLK DCY0:DX ...
SITE_PIPS M17 OUSED:0

...

STATIC_SOURCES SLICE_X2Y106/D6LUT/O6 SLICE_X2Y106/C6LUT/O6 SLICE_X4Y106/D6LUT/O6 ...
```

```
LUT_RTS SLICE_X5Y101/B6LUT/A6/O6 SLICE_X2Y100/A5LUT/A4/O5 SLICE_X5Y100/C6LUT/A6/O6 ...

...

INTRASITE AddSub[10]
INTERSITE AngStep1[0] SLICE_X2Y82/AX SLICE_X2Y87/AQ SLICE_X2Y84/A1
ROUTE AngStep1[0] CLBLM_R_X3Y87/CLBLM_M_AQ CLBLM_R_X3Y87/CLBLM_LOGIC_OUTS4 ...

...

VCC INT_L_X2Y107/VCC_WIRE INT_L_X2Y107/VCC_WIRE INT_L_X2Y107/VCC_WIRE ...
START_WIRES INT_L_X2Y107/VCC_WIRE INT_L_X2Y106/VCC_WIRE INT_R_X3Y106/VCC_WIRE ...
GND INT_R_X3Y106/GND_WIRE INT_R_X3Y106/GND_WIRE INT_R_X3Y106/GFAN1 ...
START_WIRES INT_R_X3Y106/GND_WIRE INT_L_X4Y106/GND_WIRE INT_R_X3Y105/GND_WIRE ...
```

8.1.5 contraints.rsc

The *contraints.rsc* file within a RSCP stores all XDC constraints on a Vivado design. XDC constraints are similar to UCF files for ISE designs. They can be used to set the clock frequency, constrain a top-level port to a specific package pin on the device, and set other physical implementation details. An example of a RSCP constraints file can be seen below.

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

As can be seen, a Vivado constraints file is essentially a list of Tcl commands to execute before generating a bitstream. When importing a RSCP into RapidSmith, each constraint is parsed into a `XdcConstraint` object and stored in the top-level `CellDesign`. Each `XdcConstraint` has two fields, a String command and a String representing all command arguments. Code Sample 7 demonstrates how to manipulate XDC constraints in RapidSmith. Future RS2 plans include creating a more intelligent way of handling XDC constraints.

Code Sample 7: How to manipulate XDC constraints in RS2

```
1  // Getting a list of all constraints in the current design
2  List<XdcConstraint> xdcConstraints = design.getVivadoConstraints();
3
4  // Look for all of the "create_clock" constraints in the design
5  List<XdcConstraint> crtClks = xdcConstraints.stream()
6                        .filter(xdc -> xdc.getCommandName().equals("create_clock"))
7                        .collect(Collectors.toList());
8
9  // Add a new XDC constraint to the design
10 XdcConstraint xdc = new XdcConstraint("set_property", "IOSTANDARD LVCMOS18 [get_ports
       clk]")
11 design.addVivadoConstraint(xdc);
```

8.2 Tincr Checkpoints

Tincr checkpoints are very similar to RapidSmith checkpoints, but they can be used to import designs back into Vivado after they have been modified in RapidSmith. The main different between TCPs and RSCPs, is that the *placement.rsc*, *routing.rsc*, and *constraints.rsc* files turn into *placement.xdc*, *routing.xdc*, and *constraints.xdc* respectively. Vivado is able to parse the XDC files and apply physical information to the design. On design export, RapidSmith produces a

TCP that is compatible with Vivado. The file formats of the three XDC files will not be included in this documentation. If you are interested in their format, the best way to learn is to consult the Vivado user guide and generate a TCP from RapidSmith and view the individual files.

8.3 Additional Information

The reason RSCP and TCP checkpoints are distinguished, is that there are several design implementation aspects in Vivado that aren't explictly represented. In order to accurately represent a design in RapidSmith, they must be included somehow. This section describes the parts of the design that aren't explicitly represented in Vivado, and how RapidSmith handles them. subsection 8.4 describes how to get a handle to the additional information after a checkpoint has been imported.

8.3.1 LUT Routethroughs

Besides their use in implementing logic equations, LUT BELs can also act as signal routethroughs. In Vivado, 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. These LUTs are not explicitly represented in the logical netlist because there is no cell placed on the corresponding BEL. Figure 23 shows two examples of LUTs in Vivado that are configured as routethroughs.

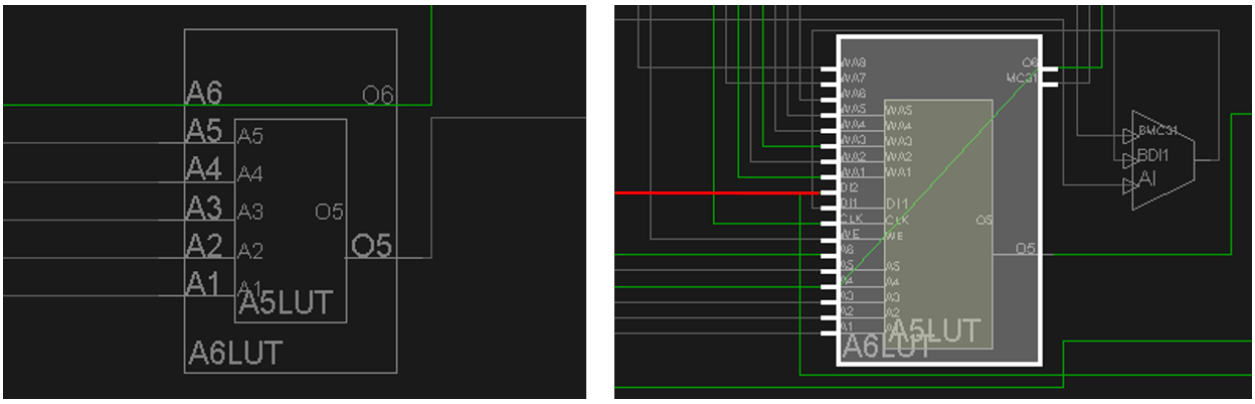


Figure 23: Two examples of LUTs being used as routethroughs in the Vivado GUI. The RT on the left uses the A6 input pin, while the RT on the right uses the A4 input pin. The net highlighted in red represents VCC.

When a design is exported from Vivado, all LUTs that are acting as routethroughs are included in the *routing.rsc* file of a RSCP. When the file is parsed, each routethrough is stored in a `BelRoutethrough` object which contains the physical `Bel`, the input `BelPin`, and the output `BelPin` that uniquely describes the routethrough. It is important to note that if a `Bel` is being used as a routethrough, no `Cell` can be placed there. When creating CAD tools, `BelRoutethrough`s give the user a more complete picture of what `Bel`s are actually available.

As described in subsection 7.3, there is a `Connection` object in RapidSmith from every input LUT pin to the LUTs output pin. These `Bel` routethrough connections, are actually just `WireConnection`s with the `isRoutethrough()` method returning true. In order to use a LUT as a routethrough in RapidSmith, you simply have to include the corresponding `WireConnection` within an INTRASITE `RouteTree`. When a design is exported from RapidSmith, the routethrough `WireConnection`s are automatically detected and converted to a passthrough LUT `Cell`. The netlist is rewired appropriately.

8.3.2 Static Source LUTs

LUTs can also act as a static power or ground source. This occurs when the configuration equation is set to 1 or 0. An example of a LUT that exhibits this behavior is shown in Figure 24. When a design is exported from Vivado, all LUTs that are acting as static sources are included in the *routing.rsc* file of a RSCP. As the file is being parsed, a list of static sources `Bel`s are recorded and stored in a list data structure. Just like routethroughs, if a `Bel` is being used as a static source no `Cell` should be placed on it.

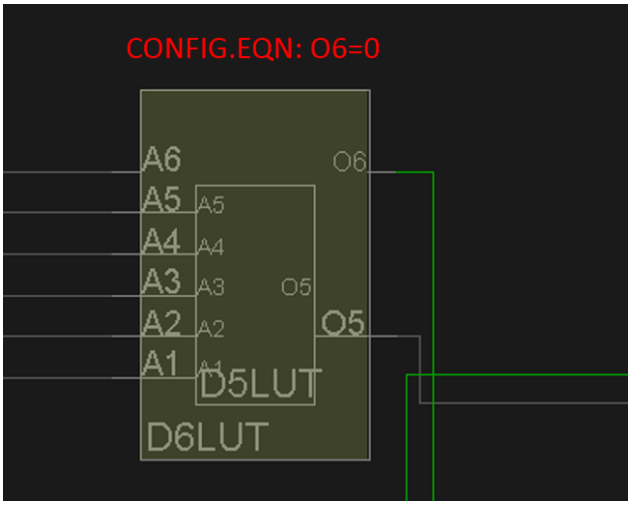


Figure 24: A LUT `Bel` that has been configured as a GND source. Notice that there are no input pins being driven.

8.3.3 Pseudo CellPins

`PseudoCellPins` are described in great detail in subsubsection 5.2.3. When a RSCP is loaded, `PseudoCellPins` are automatically detected, created, and attached to their corresponding `Cells`.

8.4 Importing and Exporting TCPs

After a Vivado design has been converted to a TCP using the `::tincr::write_rscp` command, it can be loaded into RapidSmith. Code Sample 8 demonstrates the basix syntax of design import and export.

Code Sample 8: How to import and export TCP files to and from RS2

```
// Loading a Tincr Checkpoint
TincrCheckpoint tcp = VivadoInterface.loadTcp("pathToCheckpoint.tcp");
CellDesign design = tcp.getDesign();
Device device = tcp.getDevice();
CellLibrary libCells = tcp.getLibCells();

// Insert CAD Tool Here

// Exporting the modified design to a Tincr Checkpoint
VivadoInterface.writeTCP("pathToStore.tcp", design, device, libCells);
```

While a design is being imported into RapidSmith, several useful data structures are built up. If you want to gain access to those data structures, you can pass an additional argument into the `VivadoInterface.loadTCP` method. This is shown in Code Sample 9.

Code Sample 9: Importing a TCP with additional information

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

9 Example Programs

A variety of example programs can be found in the `edu.byu.edu.rapidSmith.examples` package of the RS2 installation. They have been heavily commented to provide a means to learn the RS2 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 RS2 programs which you might find useful.

9.1 Sample Vivado Designs

To enable new users of RS2 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.tcp**: synthesized only
- **cordic.tcp**: synthesized and placed
- **count16.tcp**: synthesized, placed and routed

There is a sample design to demonstrate each possible tool flow between Vivado and RS2. 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 *.tcp* 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 25). 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.

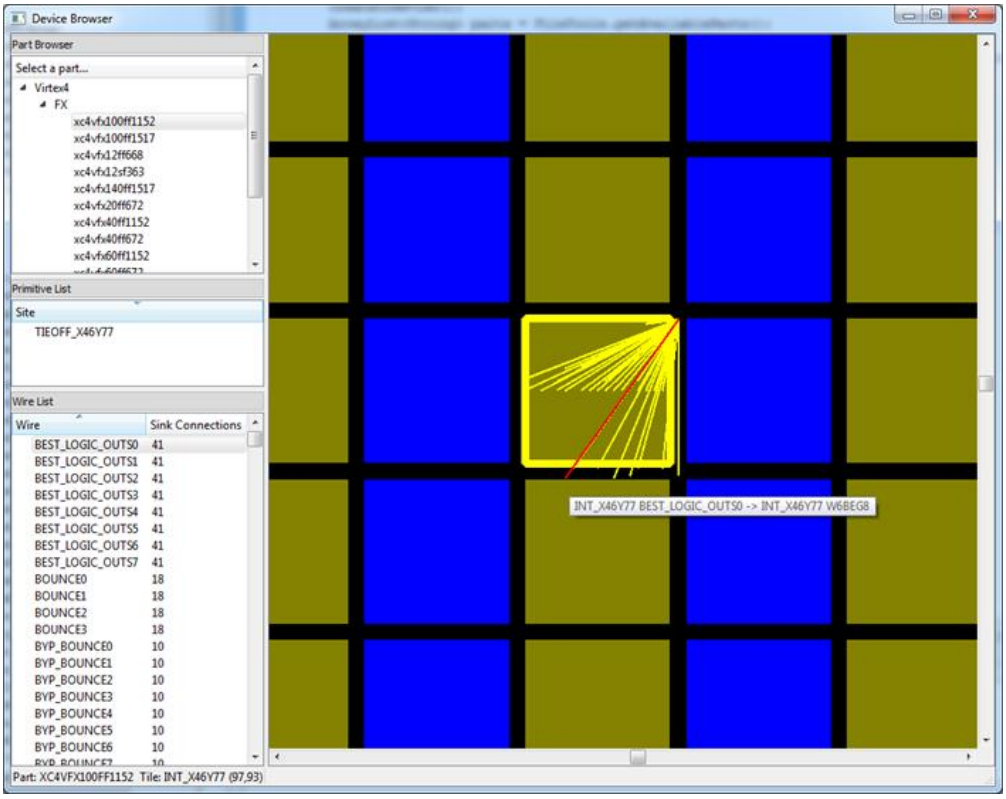


Figure 25: DeviceBrowser Screen Shot Showing Wire Connections

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 RS2. 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 Tincr checkpoint into RapidSmith, and how to export a RapidSmith netlist back into a Tincr checkpoint. This is a very important step in passing digital designs back and forth between Vivado and RapidSmith.

9.5 DesignAnalyzer

The DesignAnalyzer loads a Tincr checkpoint into RapidSmith, 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 RS2 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 RapidSmith netlist from scratch (using `Cell`s and `CellNet`s) and then places the design. While this is certainly not recommended for substantial designs, it does demonstrate how to do the following useful tasks in RapidSmith:

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

9.7 Other Test Programs

The programs introduced in this section are designed for beginners of RS2. Once you start becoming more comfortable with RapidSmith 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 Bitstreams in RS2

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 RS2 (the RapidSmith bitstream packages have been removed from RS2). 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.

11 Installing New Device Files

By default, RapidSmith includes an Artix7 *xc7a100tcsg324* part. This part has been well tested, and is a good place to start if you are learning how to use RapidSmith. However, different devices can also be created. This section describes the necessary steps to create new device files.

TODO section.

12 Legal and Dependencies

RS2 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

RS2 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

RS2 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/

RS2 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

RS2 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.

14 Appendix

TBD