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: January 20, 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 with the goal to develop a set of tools and APIs written in Java which would provide academics with an easy-to-use platform to try out experimental CAD ideas and algorithms on modern Xilinx FPGAs. RapidSmith 2 (abbreviated RS2 hereafter) represents a major addition to RapidSmith. 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

In addition, you need not start with a Vivado design — you can create a new design from scratch in RS2 and then import it into Vivado.

So, clearly a major addition with RS2 is the ability to work with Vivado. However, the other major new capability which RS2 adds over RapidSmith is that it changes RapidSmith's design representation from the Instances and Sites of ISE's XDL representation to the Cells and BELs of Vivado. 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. It is written in Java. It also depends on some understanding of Xilinx FPGAs, Vivado, and TCL. However, the goal is that this documentation provides sufficient background and detail to help bring developers up to speed on the needed topics.

RS2 by no means is a Xilinx Vivado replacement and cannot be used without a valid and current license to a Xilinx tools installation (RS2 cannot generate bitstreams for a design, for example).

1.3 Why RS2?

The Xilinx-provided TCL interface into Vivado, in theory, provides all that is needed to create any kind of CAD tool desired to augment the capabilities provided by Vivado. In practice there are a number of problems with that. First, TCL is slow — far too slow to execute a router for example. Also the Xilinx TCL interface does not manage memory well. In our experience, long-running scripts eventually cause the system to run out of memory. Brad White's MS work also determined that not 100% of the device information required to do arbitrary CAD manipulations is available through TCL. As a result, additional tools (and some small amount of manual work) are required to provide the user (and CAD tools they might like to write) with all the physical details on Xilinx parts. Simply put, some information is not available through the TCL interface. Finally, the ability to export and import designs to/from Vivado and operate on them outside Vivado using a modern high-level language such as Java is a hugely useful capability.

RS2 (in conjunction with Tincr which is described in a later section of this document) takes care of all of the generation of the FPGA part information that is required by CAD tools. It also takes care of exporting/importing designs from and to Vivado along with a myriad number of fairly arcane details associated with that process. In addition, RS2 creates special device files from the XDLRC files produced by Tincr and provides a nice API into those physical device details. All of this enables researchers to have more time to focus on what matters most: their 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/RS2 and VPR is that the RapidSmith tools aim to provide the ability to 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 are available in RapidSmith and RS2. VPR currently is limited to FPGA features which can be described using VPR's architectural description facilities.

1.6 Why Java?

We have found Java to be an excellent rapid prototyping platform for FPGA CAD tools. The Java libraries are rich with data structures useful for such applications and Java eliminates the need to clean up objects in memory. This eliminates the time needed to debug such things, leaving more time for the researcher to focus on the real research at hand. Our experience over the past decade is that for student research projects, the lack of memory management problems (dangling pointers, memory leaks,) and the associated errors has greatly improved our 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 Github.com as the project 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.

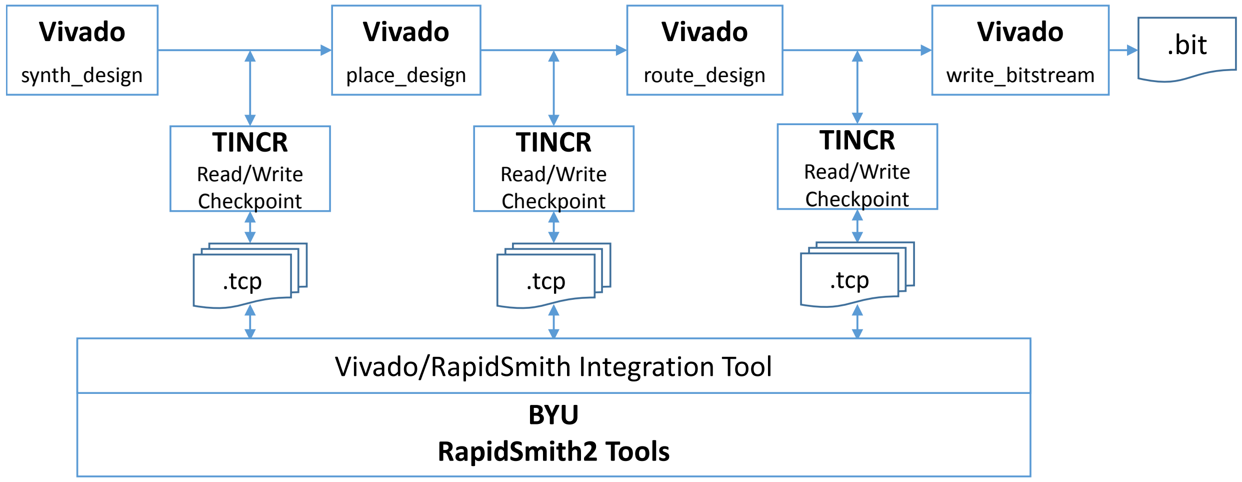


Figure 1: Vivado and RS2 Usage Model

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 you can easily build RS2 for use in Eclipse (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
- 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, the BYU Edif Tools, etc.

3.1.2 Steps for Installation For Command Line Usage

The first task is to acquire RS2 using git. You can acquire the RS2 distribution by executing the following:

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

The second task is to create an environment variable called `RAPIDSMITH_PATH` and point it at the RS2 directory thus created. This is needed so RS2 can find the required device files and other items as it runs.

The third task is to build RS2. At this point you have two choices: setting up RS2 for use with Eclipse or building RS2 manually to generate .class and .jar files which you can then use with any Java installation.

Building for Eclipse RS2 requires Eclipse Neon or later so install that. Then, create an eclipse project by executing one of the following:

```
# Will build antlr-generated files and create Eclipse project
gradlew antlr eclipse
(or gradlew.bat antlr eclipse depending on OS)
```

Executing these will create an Eclipse *.project* file. Once you have done this you can import the project into Eclipse by opening Eclipse and selecting:

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

and pointing it to the RapidSmith2 directory created when you cloned RS2 from github above. All of the Java source files will be found in Eclipse under *src/main/java*.

IMPORTANT: Eclipse does not seem to like (and instructions on the web discourage) having a git repository being inside its workspace. Put your git repository elsewhere on disk (like *MyDocuments/git*).

Building Manually Execute one of the following to build RS2:

```
gradlew build    # Will build everything needed
gradlew.bat build
```

This will produce a variety of things, any of which can be added to your `CLASSPATH` as needed:

- The resulting RS2 class file directory tree will be found in *build/classes/main*.
- A jar file of the above RS2 class files can be found in *build/libs*.
- Both tar and zip files can be found in *build/distributions*. They contain a full jar of the RS2 build along with copies of other needed jar files. You should add them all to your `CLASSPATH` except the qtjambi ones - just add the qtjambi one for your particular system (note there is no 64-bit qtjambi for windows so use the 32-bit one).

At this point you should be able to write tools that use RS2.

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 manual build process puts its .class and .jar files. Make sure you understand that before you try to combine these two build/execute methods (or better yet, don't combine them).

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 you create in Eclipse 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:

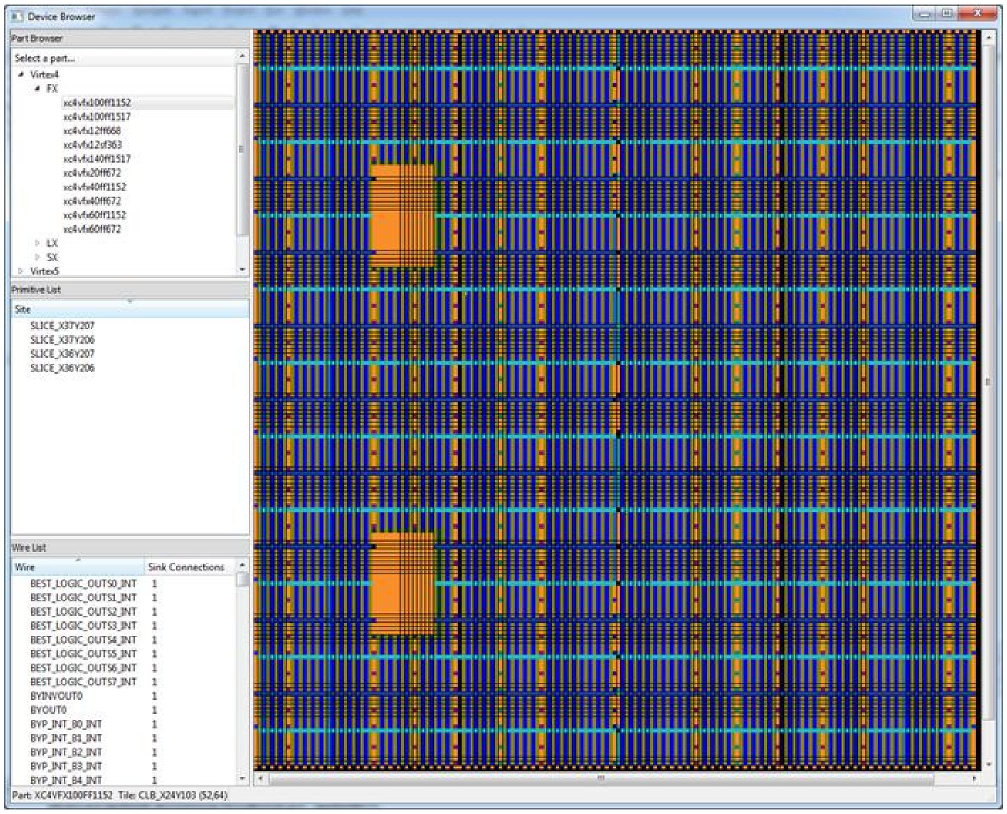
- The RS2 code base contains a number of assertions which may be helpful as you are developing code. These are not enabled by default in Java. To enable them, add *-ea* as a VM argument. This is highly recommended.

- 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 you will typically get file open failure messages as RS2 tries to load device files and the like.

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
```



Wire	Sink Connections
BEST_LOGIC_OUTS0_INT	1
BEST_LOGIC_OUTS1_INT	1
BEST_LOGIC_OUTS2_INT	1
BEST_LOGIC_OUTS3_INT	1
BEST_LOGIC_OUTS4_INT	1
BEST_LOGIC_OUTS5_INT	1
BEST_LOGIC_OUTS6_INT	1
BEST_LOGIC_OUTS7_INT	1
BYINVOUT0	1
BYOUT0	1
BYP_INT_B0_INT	1
BYP_INT_B1_INT	1
BYP_INT_B2_INT	1
BYP_INT_B3_INT	1
BYP_INT_B4_INT	1

Part: XC4VFX100FF1152 Tile: CLB_X24Y103 (52,64)

Figure 2: **DeviceBrowser** Sample Display

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.

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 using this part (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 Example RS2 Programs and Sample Vivado Designs

A variety of example programs can be found in the `edu.byu.edu.rapidSmith.examples` package in the RS2 installation. They have been heavily commented and so provide a means to learn the RS2 API by example as we believe this is much better than reading a lot of text trying to teach you what you need to know.

There is a *README.txt* file in that directory to provide an overview and suggested order for learning from the examples. In addition, the subsections below describe one or more built-in RS2 programs which you might find useful.

4.1 DeviceBrowser

Note: this is a program from the original RapidSmith, but which is discussed here because it is still very useful in RS2.

This GUI program is located in `edu.byu.ece.rapidSmith.device.browser` package. It will let you browse parts at the tile level. 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.

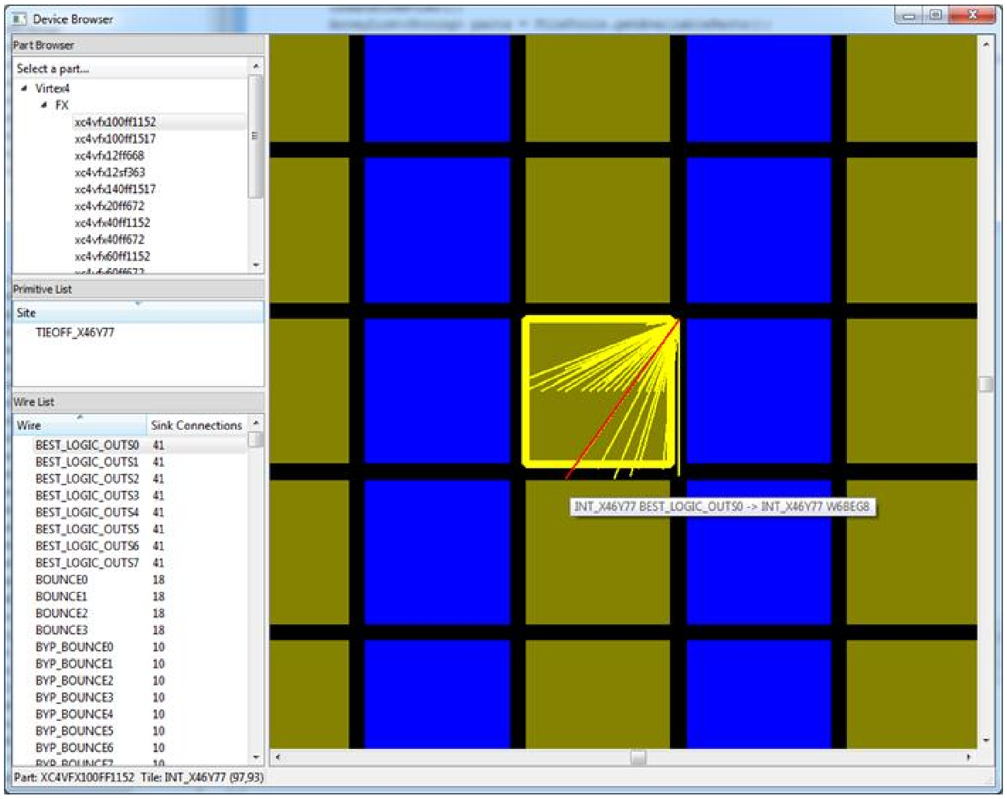
All that is required for this to operate is a valid device file (no design required). A screenshot of the **DeviceBrowser** program is shown in Figure 2.

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 the screenshot below). 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. This is shown in Figure 3. Thanks to Chris Lavin for originally creating this app.

4.2 The DesignAnalyzer Test Program

This program, along with a number of other example programs, is located in the `edu.byu.ece.rapidSmith.examples` package. After loading a design from a checkpoint, it simply walks the design data structure, printing out what it finds as it goes. As such, it provides a nice example of a number of things which would be useful for getting started with RS2:

- How to enumerate the Cells in a design, determine and print their placement information as well as their properties.
- How to enumerate the logical nets in a design and print out their source and sink pins.



xc4vfx100ff1152
xc4vfx100ff1517
xc4vfx12ff668
xc4vfx12sf363
xc4vfx140ff1517
xc4vfx20ff672
xc4vfx40ff1152
xc4vfx40ff672
xc4vfx60ff1152

Site
TIEOFF_X46Y77

Wire	Sink Connections
BEST_LOGIC_OUTS0	41
BEST_LOGIC_OUTS1	41
BEST_LOGIC_OUTS2	41
BEST_LOGIC_OUTS3	41
BEST_LOGIC_OUTS4	41
BEST_LOGIC_OUTS5	41
BEST_LOGIC_OUTS6	41
BEST_LOGIC_OUTS7	41
BOUNCE0	18
BOUNCE1	18
BOUNCE2	18
BOUNCE3	18
BYP_BOUNCE0	10
BYP_BOUNCE1	10
BYP_BOUNCE2	10
BYP_BOUNCE3	10
BYP_BOUNCE4	10
BYP_BOUNCE5	10
BYP_BOUNCE6	10

INT_X46Y77 BEST_LOGIC_OUTS0 -> INT_X46Y77 W6BEG8

Part: XC4VFX100FF1152 Tile: INT_X46Y77 (97,93)

Figure 3: **DeviceBrowser** Screen Shot Showing Wire Connections

- How to traverse and print out the physical route for a logical net (if it is routed)

4.3 The DeviceAnalyzer Test Program

This program is also located in the `edu.byu.ece.rapidSmith.examples` package. It is designed as a simple getting started program and demonstrates how to query and print tiles in a device, wires in a tile, etc.

4.4 Other Test Programs

See the README.txt file in the `edu.byu.ece.rapidSmith.examples` package directory. It outlines the other test programs there which may be useful in coming up to speed on RS2.

4.5 Sample Vivado Designs

To enable new users of RS2 to be able to quickly start running the above test programs, a small set of pre-compiled Vivado designs have been included in the RS2 distribution. They are located in the *exampleVivadoDesigns* directory and consist of 3 designs:

- *add*: synthesized only
- *cordic*: synthesized and placed

- *count16*: synthesized, placed and routed

The RS2 checkpoints are contained in the following directories: *add.tcp*, *cordic.tcp*, and *count16.tcp*. Vivado checkpoints are also included and are the *add.dcp*, *cordic.dcp*, and *count16.dcp* files.

To re-build one of the example designs in Vivado a compile script called **compile.tcl** has been included in the *exampleVivadoDesigns* directory. To re-build one of the sample designs, you would start up the Vivado Tcl shell from your Vivado distribution and then execute the following in the Tcl shell:

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

This will re-synthesize, place, and route the add design and, from that compiled design, generate the *.tcp* directory and the *.dcp* file.

5 Designs in RS2

Designs in RS2 are similar to the designs found in Vivado (which are exported as EDIF files). They are essentially logical netlists. They are represented and stored in the data structures found in the `design.subsite` package. The data structure hierarchy of the `design.subsite` package can be seen in figure 4.

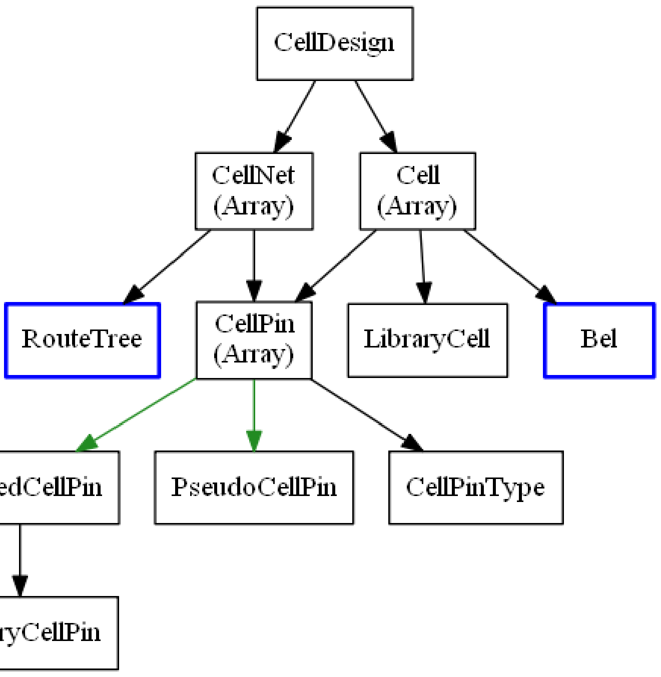


Figure 4: RapidSmith 2 `Design` data structure tree. Green arrows represent inheritance, and black arrows represent association. Blue boxes represent physical implementation components of the netlist (i.e. what physical `Bel` a `Cell` is placed on, or what physical `Wires` are used in a `CellNet`)

A `CellDesign` consists of a collection of `Cell` objects, interconnected by `CellNets`. CellNets connect to the `CellPins` on Cells. CellNets typically have one source pin and one or more sink pins. Cell objects have a name, properties, pins, a link to the library cell they are an instantiation of, etc. Cells may be placed onto `BELs` and the corresponding CellPins mapped onto `BelPins`. CellNets, when physically routed, map onto one or more `RouteTrees`.

5.1 The Cell Class

The example programs mentioned above provide examples of manipulating Cell objects. Here are a few things you should know about cells, in no particular order:

- A Cell always contains a reference to an object of type `LibraryCell`, which serves as a template for its construction.
- Cells may be physically placed onto BELs in the device. This is done by setting the Cell's anchor value to point to the BEL it resides on. If you know where you want a Cell placed

you can just place it there. On the other hand, RS2 provides a way to identify the Site/BEL combinations where a Cell could be placed. See the program **CreateDesignExample** in the *examples* directory for an illustration of how to do it both ways.

- Cell objects have pins on their periphery where CellNets connect to.
- The top-level ports of a design are tied to `IPORT`, `OPORT`, or `IOPORT` Cell objects. These are pseudo-cells (you won't find them in Vivado) and represent the terminal points for signals leaving or entering the top-level.

5.1.1 Cell Properties

Cells as represented in EDIF files coming from Vivado may contain properties. For example, a D flip flop cell (FDRE) has a `CONFIG.INIT` property, indicating what its power-up state should be. These properties can be set to modify the Cell's behavior. The **DesignAnalyzer** test program described above pretty-prints an RS2 logical design and, as a part of its operation, it lists the properties set on each Cell in the design. Here are some additional things about properties you should know:

- It might be of interest to learn what properties could be set for a given cell. This set of properties can be found in the *cellLibrary.xml* files generated for a given family (see the *$RAPIDSMITH_PATH/devices* directory and its sub-directories to find these XML files for any devices installed). The files are quite readable and from them you can learn much about the available LibraryCell types for a given FPGA family (look for the `libcellproperty` tags in the file). At some point in the future this information will be incorporated into the RS2 data structures so that user programs can query them and so RS2 can check whether they are legal values when set by a user program. For now, user code can set properties and those will be exported into EDIF when going from RS2 back into Vivado. However, no error checking will be done by RS2 as this is done.
- In a GUI view of devices in Vivado you will see polarity inverters in many sites allowing for programmable selection of a signal or its inverse. This is shown in the GUI in the form of a 2:1 MUX. The CLK signal and its inverse entering a SLICE is an example of this. However, this is not explicitly represented in the device representation. Rather, properties on the Cells driven by the mux output signals muxes indicate whether the signal is inverted or not. For example, generate a 4-bit counter using rising-edge triggered flip flops in Vivado and generate an EDIF file for it. You will see that the counter is constructed, in part from FDRE cells. Now, modify the HDL for your counter to make it a falling-edge triggered counter and compare the resulting EDIF file. The difference you will see is that the property on each of the FDRE cells called `CONFIG.IS_C_INVERTED` has been set, indicating it is a falling-edge triggered flip flop. When **bitgen** is actually run by Vivado, the corresponding clock inverter will be programmed accordingly.
- It should go without saying that since there is only one such clock inverter in a SLICE, all the flip flops in a slice must be either rising-edge triggered or falling-edge triggered (they must have the same `CONFIG.IS_C_INVERTED` control set value). If you violate this, Vivado will throw an error. Similar restrictions exist for all cells in a site driven by shared programmable

inverters. For example, flip flops in a slice (FDRE LibraryCells) may share programmable inverters on their clock, clock enable, and R inputs.

5.2 The CellNet Class

- A CellNet has a type. Legal values are WIRE, GND, VCC, and UNKNOWN. The WIRE type is the one used for normal signals. CellNets have one source pin and one or more sink pins (these are of type CellPin). The CellNet class has methods for traversing these.
- GND and VCC nets have some special characteristics. There is a single logical VCC net. It is driven by a single RapidSmithGlobalVcc cell. The output pin of that cell is the source of all VCC in the design. However, unlike other cells which get routed to, this cell is never physically placed. The situation with GND is similar.

5.2.1 Physical Routing of CellNets in RS2

- A CellNet is physically routed by determining the metal segments and intervening PIPs that are to be used to make up the route. A physical net is called a `Wire` and contains some number of RouteTree objects. A given RouteTree object has the source of the route as its root and then branches represent the branching of the route between source and sink. The physical routing of a net is represented by attaching one or more RouteTree objects to the net.
- Normal wires (CellNets of type WIRE) have only one RouteTree, reflecting the fact that they have a single source and multiple sinks. Note that a wire cannot be physical routed to the pin of a cell which has not yet been placed.
- Physically, GND and VCC nets are unique. When a circuit has been routed by Vivado the result will be multiple physical VCC routes and multiple physical GND routes in the circuit. Each route is represented by its own RouteTree object. The source for each of these RouteTree objects will be a wire which is connected to a TIEOFF. These TIEOFFs are not physically placed but their locations can be inferred by the source wire for each of the RouteTrees making up the VCC or GND route.
- Once a CellNet's physical routing has been created as a RouteTree, that is converted to a directed routing string when RS2 designs are exported from RS2 back into Vivado. The **DesignAnalyzer** program in the examples directory gives an example of tracing out the RouteTrees which represent a physically routed wire.

The program **DesignAnalyzer** in the `edu.byu.ece.rapidSmith.examples` package provides an illustration of how to traverse a physically routed CellNet. This is done in its *createRoutingString()* method. This method starts by getting the RouteTree object associated with the source pin for the given CellNet. It then recursively follows the linked set of RouteTree objects to follow the wire. Liberal comments in the **DesignAnalyzer** program illustrate how this is done. Consult it for details.

6 Devices in RS2

6.1 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 section 6.2. 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.

6.1.1 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 5 displays three types of template `Tiles` in an Artix7 part.

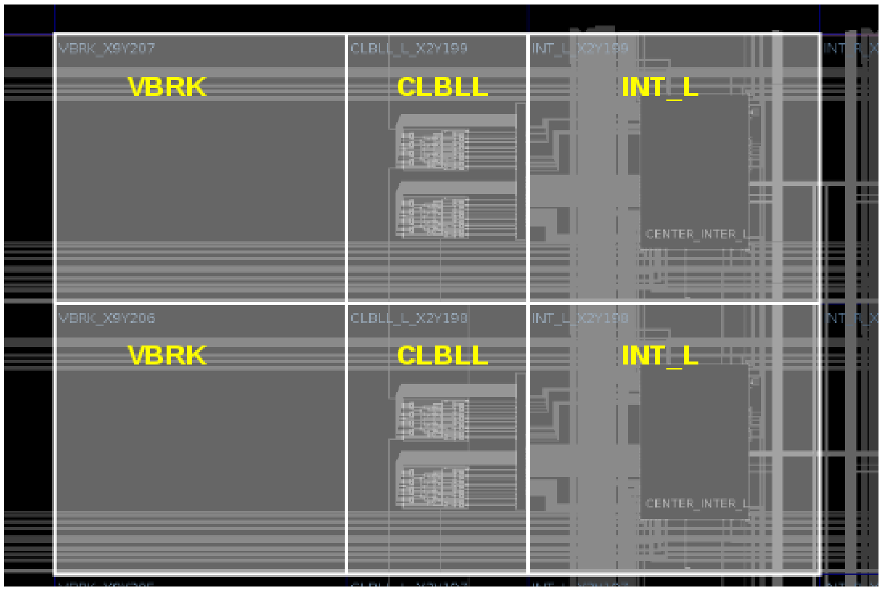
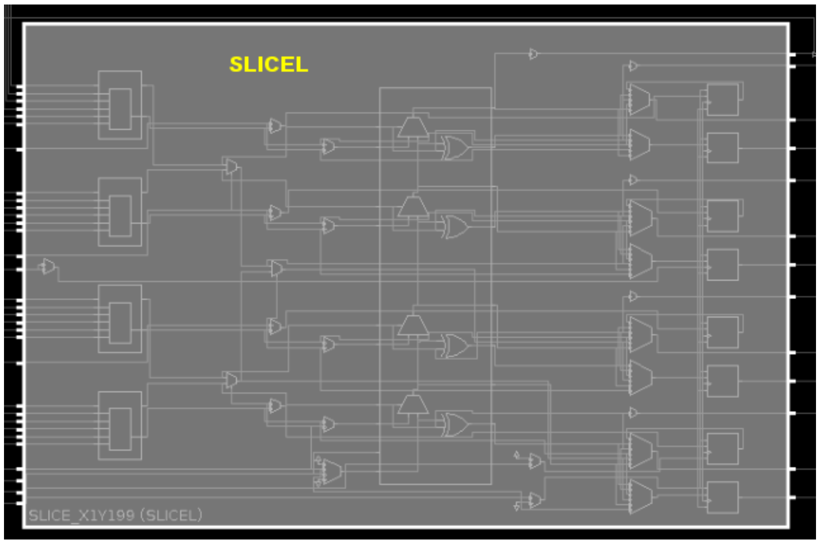


Figure 5: Highlighted example `Tiles` in an Artix7 FPGA.

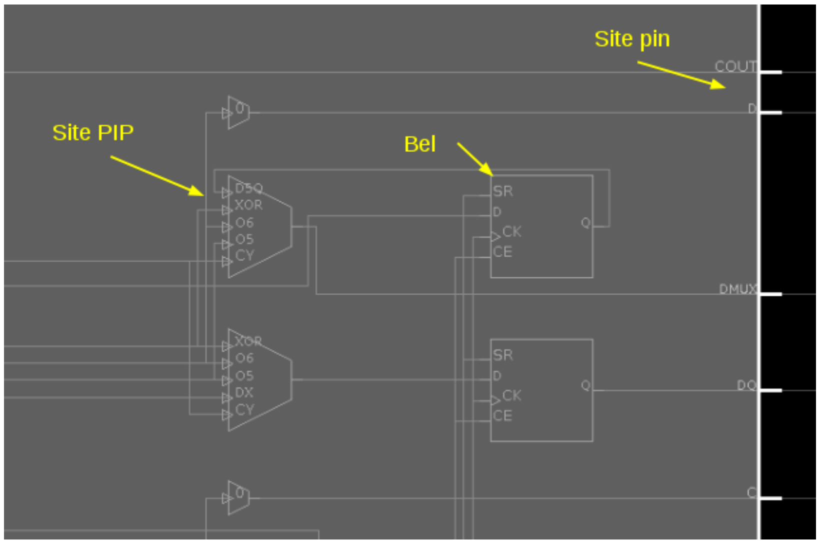
The two **VBRK** `Tiles` on the left are used for intermediate wiring. The two **INT_L** `Tiles` on the right are switchbox `Tiles`. These are reconfigurable routing `Tiles` that allow a single wire to go multiple different locations within the FPGA. The two **CLBLL** `Tiles` in the middle are used to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

6.1.2 Primitive Sites

Each `Tile` can contain one or more `Primitive Sites` (often shortened to `Site`). As figure 5 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 6 shows a SLICEL, one of the `Sites` within a CLBLL `Tile`. As the figure shows, there are three main components to



(a)



(b)

Figure 6: (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.

`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.

6.1.3 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 routed to many different locations in the FPGA. Figure 7 shows an example of a switchbox.

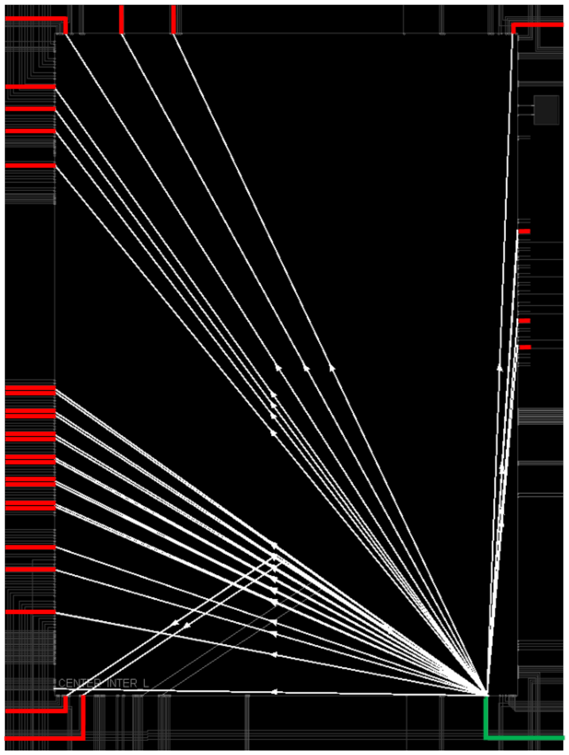


Figure 7: An example of PIP wire connections in a device. The green wire represents the source wire, and the red wires represent all possible sink wires in the Switchbox. The highlighed white sections of the figure are PIP connections.

6.2 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 8 shows the new data structure hierarchy, which can be found in the *edu.byu.ece.rapidSmith.device* package.

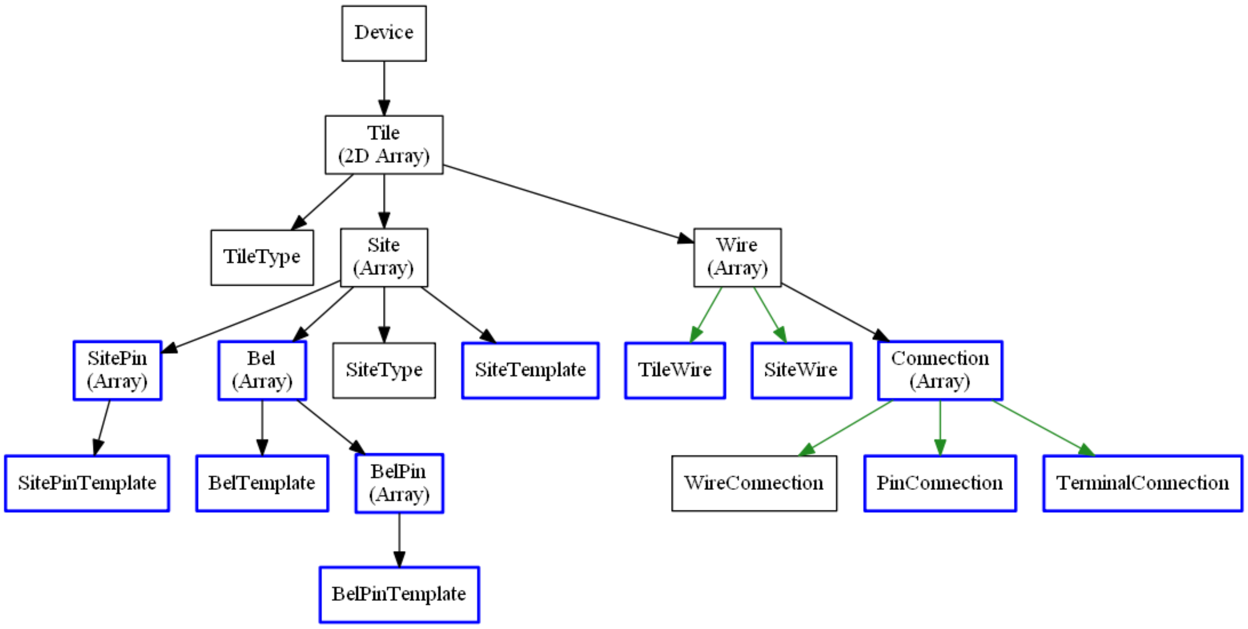


Figure 8: RapidSmith2 `Device` data structure tree. Green arrows represent inheritance, and black arrows represent association. Classes and Interfaces bolded in blue are new to RapidSmith 2.

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 4 illustrate how to load and browse a device down to the `Tile` and `Site` levels.

6.2.1 Templates

As figure 8 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 Artix7 *xc7a100tcsg324* part, there are 11,100 `Sites` of type SLICEL. Each of these SLICELs have 215 `BELs`, `Site Pins`, and `Site PIPs` 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.

6.2.2 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 as 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.

6.2.3 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.

6.2.4 Wire and Connection Interfaces

As figure 8 shows, there are two classes that inherit the `Wire` interface (described in section 6.2.3) and three classes that inherit the `Connection` interface (described in section 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 4 demonstrate the use of these interfaces.

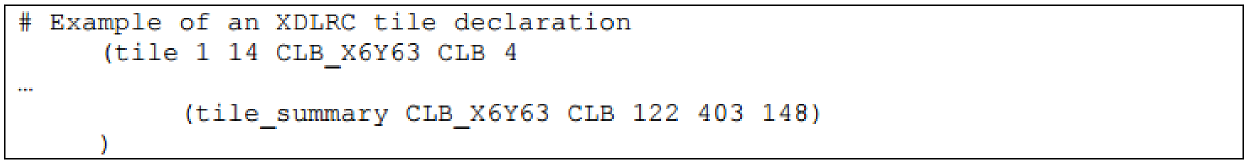
6.3 Device Files

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.

6.3.1 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 accompaning images. As you will see, much of the terminology is the same as section 6.1.

Tiles

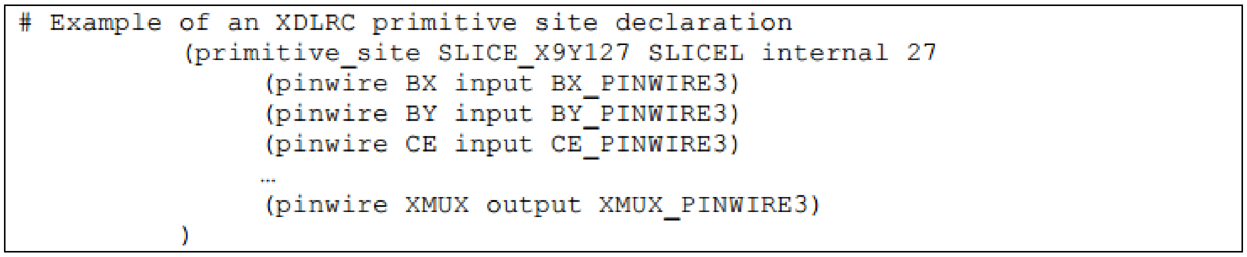


```
# Example of an XDLRC tile declaration
     (tile 1 14 CLB_X6Y63 CLB 4
...
          (tile_summary CLB_X6Y63 CLB 122 403 148)
     )
```

Figure 9: Tile syntax in XDLRC files

A tile in an XDLRC file corresponds to the same thing as the `Tiles` described in section 6.1. 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 10: 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 11: 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 section 6.2.2. The `WireConnection` objects that are created from this part of the XDLRC are described in section 7.1.

PIP

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

Figure 12: 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. Section 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 Pins`, `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 13. The sub-site data structures in RapidSmith (`Bels`, `SiteWires`, 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)
                (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 13: Primitive Def sections of XDLRC files

6.3.2 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.

7 Routing in RS2

7.1 Wires and WireConnections

Routing in RapidSmith is done using `Wire` objects, which are described in section 6.1.3. `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 7). These types of connections are important to FPGA routing, because they dynamically configure the routing network for a given design.

2. **Non-PIP Connections**: Connect the same physical wire across two different tiles. In general, wires stretch across multiple tiles in a device, having a different name in each tile. This is demonstrated in figure 14. 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.

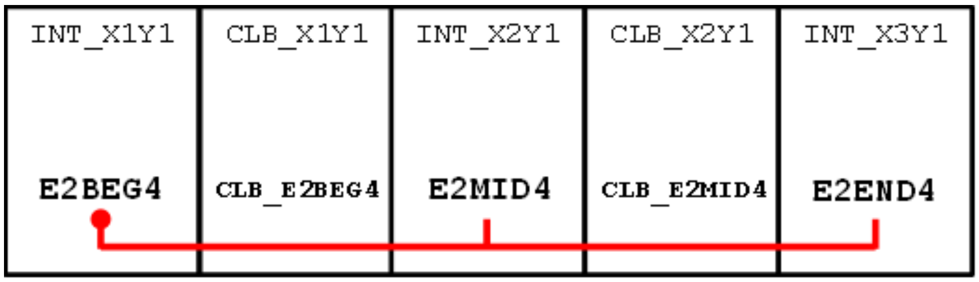


Figure 14: A wire in an FPGA illustrating how each part of the wire has a different name depending on the tile it is located in

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()`). 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.

7.4 RouteTrees

`Wires` and `WireConnections` are the fundamental objects used to specify and explore routing in RapidSmith, but they need to be encapsulated 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` data structure, which handles `Wires`, `WireConnections`, and branching while routing a `CellNet`. As figure 15 shows, it is a simple tree structure that organizes wires and wire connections. The fields in a `RouteTree` include:

- A link to the the parent `RouteTree`
- The `Connection` from the parent tree to the current tree

- A list of sink `RouteTree` children
- A cost field for routers

When a design is imported from Vivado, the routing information is representated using `RouteTrees`. On design export, the `RouteTree` for each `CellNet` is parsed 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. The **DesignAnalyzer** example demonstrates how to iterate through a `RouteTree`. The **AStarRouter** and **HandRouter** examples demonstrate how to build a `RouteTree`.

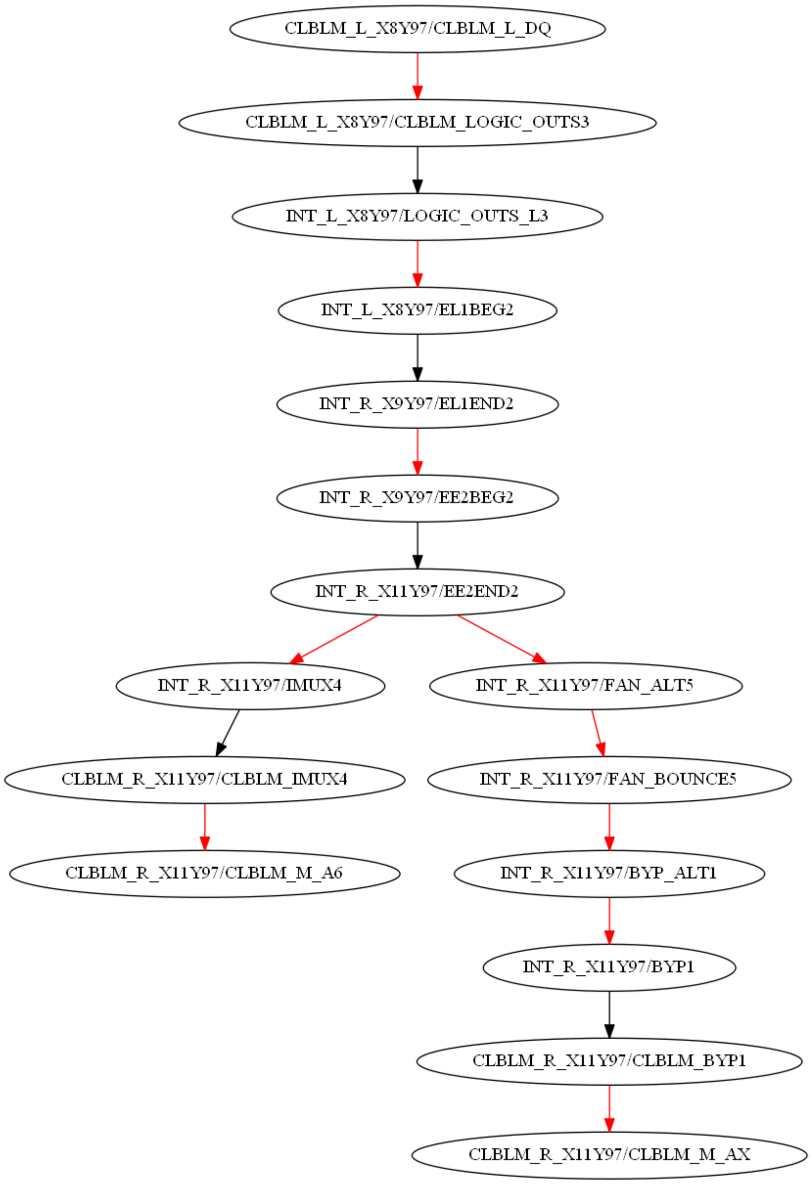


Figure 15: A DOT file representation of the RapidSmith2 RouteTree data structure. Nodes represent `Wires`, red edges represent `PIP` wire connections, and black edges represent non-PIP wire connections. This specific RouteTree was created from the **AStarRouter** example.

8 Importing/Exporting Designs Between Vivado and RS2

Importing and export designs between Vivado and RS2 is straightforward. See the program edu.byu.ece.rapidSmith.examples.ImportExportExample for an illustration of how to do this.

9 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.

10 Legal and Dependencies

RS2 is released under GPL version 3.

10.1 RapidSmith Legal Text

11 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.

12 Appendix

TBD