

**RapidSmith 2**

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

**Technical Report and Documentation**

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

Revised: 12 July 2016

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

# Introduction

## 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.

The 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 language to the Cells and BELs of Vivado.

This opens up a world of new CAD research opportunities which were difficult to perform using Rapidsmith.

## 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.

## 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 (this has been unofficially confirmed by reading between the lines of the responses we have received to bug reports we have filed). 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.

## Which Xilinx Parts does RS2 Support?

As of the writing of this document, Virtex 7 has been tested the most and are 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 and, at some point, Ultrascale should be fully supported[[1]](#footnote-1).

As will be seen later, to generate additional device files for additional parts *within a supported family* is relativel straightforward and can be done by any user. As will also be seen later, new families can also be supported but this requires more work.

## How is RS2 Different than VPR and VTR?

[VPR (Versatile Place and Route)](http://www.eecg.utoronto.ca/vpr/) has been an FPGA research tool for several years and has led to hundreds of 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.  Our understanding is that VPR currently is limited to FPGA features which can be described using VPR's architectural description facilities.

## Why Java?

We have found Java to be a 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 in other development platforms, 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 and its associated errors has greatly improved our student productivity and led to far more stable CAD tools.

# Vivado, RS2, and Tincr

## RapidSmith vs. RS2

### 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 as 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.

### 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. However, with Vivado Xilinx has provided an extensive TCL scripting capability which it 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. The development of RS2 consisted of two parts.

#### 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 simpler, (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 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

#### 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, an initial 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.

### What is All This About XDL 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 use it to support the creation 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 primitive 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. Cells are placed onto ***BEL*** objects 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 using 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.

## 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

# Getting Started

## Installation

### Getting RS2

RS2 is available on Github at:

[https://github.com/byuccl/RapidSmith2](https://github.com/xrtc/RapidSmith2)

The repository contains all the files you need (including supporting JAR files). While project files are not included in the Github repo, we highly recommend you use something like Eclipse as your IDE.

### Requirements for Installation

* Windows, Linux or Mac OS X all will work (see notes below for Mac OS X)
* Vivado
* JDK 1.8 or later NOTE: If you plan on using the Qt Jambi framework in a Windows environment, you will need the 32-bit JRE (Qt Jambi 4.6.3 had yet to be compiled in 64-bit Windows as of the writing of this document).
* Supporting JARs
  + INCLUDED: [Caucho Hessian Implementation](http://hessian.caucho.com/) [JAR v.4.0.6](http://caucho.com/download/hessian-4.0.6.jar) (Used for compressing database device files)
  + INCLUDED: [Qt Jambi](http://doc.qt.nokia.com/qtjambi-4.4/html/com/trolltech/qt/qtjambi-index.html) (Qt for Java) for the Part Tile Browser example. Just adding the [jars](http://qt.nokia.com/downloads/) to the CLASSPATH variable is adequate.
  + INCLUDED: [JOpt Simple](http://jopt-simple.sourceforge.net/) for use by some examples in the bitstream tools packages.
  + OPTIONAL: [JavaCC](https://javacc.dev.java.net/) if the user wants to change the XDL design parser. There is also a [good plugin for Eclipse for JavaCC](http://eclipse-javacc.sourceforge.net/) which makes it easier to modify and compile .jj files.

### Steps for Installation For Command Line Usage

1. Clone the Github byuccl/RapidSmith2 project to a local place. This can be done by executing the following command from the command line of a machine with *git* installed: git clone [https://github.com/xrtc/RapidSmith2](https://github.com/srtc/RapidSmith2)  
   This will create a clone of the Github repository for RapidSmith2 in the current directory (it will create a RapidSmith2 directory and put it into that).
2. Add all the jar files in the jars folder (hessian-4.0.6.jar, qtjambi-4.6.3.jar, qtjambi-<your\_platform>-4.6.3.jar, jopt-simple-3.2.jar) to your CLASSPATH environment variable.
3. Add the location of your Java project to your CLASSPATH environment variable.
4. Create an environment variable called RAPIDSMITH\_PATH and set its value to the path where you have the Java project located on your computer.
5. Compile all of the Java classes.
6. Test your installation by running any of the programs such as:

|  |
| --- |
| java edu.byu.ece.rapidSmith.device.browser.DeviceBrowser |

If all goes well you should see a graphical representation showing the details of a physical FPGA device as shown in Figure 2.

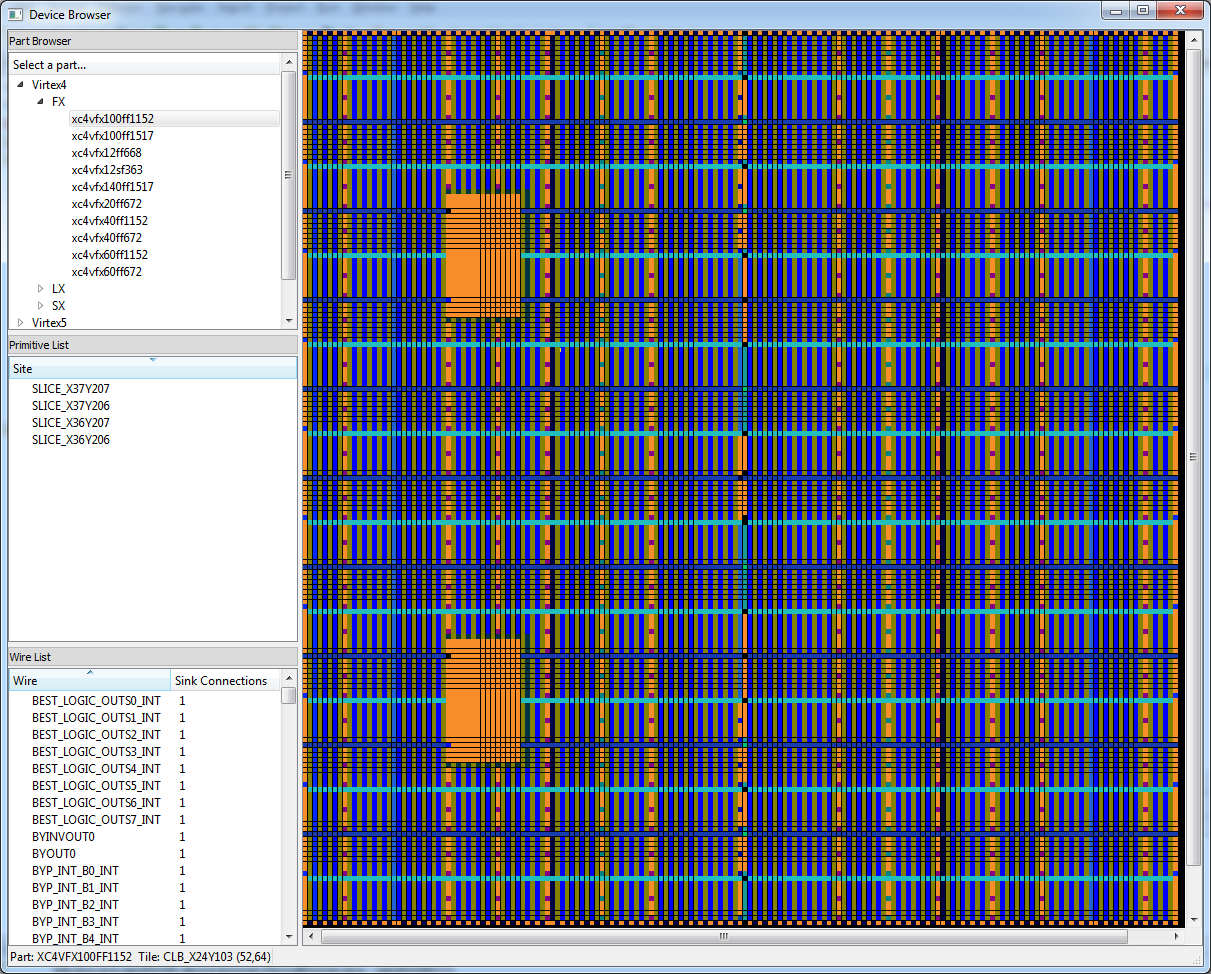


Figure – Device Browser Sample Display

### Steps for Installation For Eclipse Usage

For Eclipse usage, you follow essentially the same steps above. The major difference is that you perform each task via Eclipse commands:

1. Using File🡪Import from Eclipse you can clone the Github repo at byuccl/RapidSmith2 locally and then using the New Project Wizard create an Eclipse project for the cloned copy of RapidSmith2. IMPORTANT: do not specify a location inside your Eclipse workspace as the destination for the Import – Eclipse does not like that and will give vague errors if you do so. Instead, import into a location like /Users/smith/Documents/git/RapidSmith2. As above, the location to clone from is <https://github.com/xrtc/RapidSmith2>.
2. Add all the jar files in the jars folder (hessian-4.0.6.jar, qtjambi-4.6.3.jar, qtjambi-<your\_platform>-4.6.3.jar, jopt-simple-3.2.jar) to your project in Eclipse. This can be done in the project properties window under the “Java Build Path” entry.
3. Define an environment variable called RAPIDSMITH\_PATH and set its value to the path where you have the Java project located on your computer. This can be done globally on your computer or it can be defined in each Run Configuration you create.
4. Test your installation by creating a Run Configuration to execute the program:  
     
    edu.byu.ece.rapidSmith.device.browser.DeviceBrowser   
     
   and it should bring up the GUI above in Figure 2

### Device Files For Use With RS2

Device files for one part (the xc7a100tcsg324) are included in the repo so you can immediately start working with RS2 using this part (initially, it will be the only device available when you run the DeviceBrowser above). If you desire to work with additional parts, follow the instructions found in this documentation in **Section 7 Generating Device Files in RS2** on page 20.

### Additional Notes for Mac OS X Installation

* The instructions above require you to set environment variables. If using an IDE such as Eclipse, this can be accomplished inside Eclipse (see Run configurations, Environment tab). Adding environment variables can also be done directly in Mac OS X. To add/modify global environment variables in Mac OS X, one preferred way would be to edit the environment.plist file in the ~/.MacOSX directory. Here is an example of a proper setup for RapidSmith (assuming the location where the RS2 repo was cloned to):

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">  <plist version=*"1.0"*>  <dict>  <key>RAPIDSMITH\_PATH</key>  <string>/Users/[user name]/Documents/git/RapidSmith2</string>  <key>CLASSPATH</key>  <string>$CLASSPATH:/Users/[user name]/Documents/git/RapidSmith2:/Users/[user name]/Documents/git/RapidSmith2:/Users/[user name]/Documents/git/RapidSmith2/jars/hessian-4.0.6.jar:/Users/[user name]/Documents/git/RapidSmith2/jars/qtjambi-4.6.3.jar:/Users/[user name]/Documents/git/RapidSmith2/jars/qtjambi-macosx-gcc-4.6.3.jar</string>  </dict>  </plist> |

* Another difference is that when running programs that use Qt in Java under Mac OS X, the user will need to supply an extra JVM switch, “-XstartOnFirstThread”

# Examples

## CreateDesignExample Source Code

To get started programming with RapidSmith, here is an example of a very simple program. It is heavily documented and consists of creating a new design, adding some new cells and nets to it, and placing a couple of the cells. It then prints out the netlist by iterating over the data structure thus created.

|  |
| --- |
| **package** edu.byu.ece.rapidSmith.examples2;  **import** java.io.\*;  **import** edu.byu.ece.rapidSmith.RapidSmithEnv;  **import** edu.byu.ece.rapidSmith.design.\*;  **import** edu.byu.ece.rapidSmith.design.subsite.\*;  **import** edu.byu.ece.rapidSmith.device.\*;  **public** **class** CreateDesignExample {  /\*\*  \* A simple class to illustrate creating designs in RapidSmith2.  \* **@author** Brent Nelson  \*/  **public** **static** **void** main(String[] args) **throws** IOException{  // Load the cell library from the directory indicated by the part name  // (directory should be $RAPIDSMITH\_PATH/devices/artix7)  CellLibrary libCells = **new** CellLibrary(RapidSmithEnv.*getDefaultEnv*()  .getPartFolderPath("xc7a100tcsg324")  .resolve("cellLibrary.xml"));  System.***out***.println("Cell library loaded: cellLibrary.xml");  // Load the device file from the directory indicated by the part name  Device device = RapidSmithEnv.*getDefaultEnv*().getDevice("xc7a100tcsg324");  System.***out***.println("Device loaded: xc7a100tcsg324");    // Create a new empty CellDesign for the designated FPGA part  CellDesign design = **new** CellDesign("HelloWorld2", "xc7a100tcsg324");  // Create a new cell and add it to the current design. It is a LUT1 cell.  // Then, set the INIT property for the LUT cell (program the LUT)  Cell invcell = design.addCell(**new** Cell("lutcell", libCells.get("LUT1")));  invcell.updateProperty("INIT", PropertyType.***DESIGN***, "2'h1");  // Create a flip flop cell and set its properties  Cell ffcell = design.addCell(**new** Cell("ffcell", libCells.get("FDRE")));  ffcell.updateProperty("INIT", PropertyType.***DESIGN***, "INIT0");  ffcell.updateProperty("SR", PropertyType.***DESIGN***, "SRLOW");    // Create IOB's for the circuit's q output and for its clk input  Cell qbufcell = design.addCell(**new** Cell("qbuf", libCells.get("OBUF")));  Cell clkbufcell = design.addCell(**new** Cell("clkbuf", libCells.get("IBUF")));    // Create the Q wire and connect it up  CellNet qnet = design.addNet(**new** CellNet("qnet", NetType.***WIRE***));  qnet.connectToPin(ffcell.getPin("Q"));  qnet.connectToPin(invcell.getPin("I0"));  qnet.connectToPin(qbufcell.getPin("I"));  // Create and connect the wire between the LUT' output to the flip flop's D input  CellNet dnet = design.addNet(**new** CellNet("dnet", NetType.***WIRE***));  dnet.connectToPin(ffcell.getPin("D"));  dnet.connectToPin(invcell.getPin("O"));  // Create and connect the clock wire between the IOB output and the flip flop's input  CellNet clknet = design.addNet(**new** CellNet("clknet", NetType.***WIRE***));  clknet.connectToPin(clkbufcell.getPin("O"));  clknet.connectToPin(ffcell.getPin("C"));  // Let's place some of the cells into a slice  // Get the first SLICEL in the device  Site slice = device.getAllSitesOfType(SiteType.***SLICEL***)[0];    // Let's place the invcell on the A5LUT  design.placeCell(invcell, slice.getBel("A5LUT"));  // Let's place the ffcell on the AFF  design.placeCell(ffcell, slice.getBel("AFF"));  // Now, prettyprint what we have created  **for** (Cell c : design.getCells()) {  System.***out***.print("Cell: " + c.getName() + " " +  c.getLibCell().getName());  **if** (c.isPlaced())  System.***out***.println(" <<<Placed on: " + c.getAnchor() + ">>>");  **else** System.***out***.println();  **for** (CellPin cp : c.getPins()) {  System.***out***.println(" Pin: " + cp.getName() + " " +  cp.getDirection() + " " +  (cp.getNet()!=**null**?cp.getNet().getName():"<unconnected>"));  }  **for** (Property p : c.getProperties()) {  System.***out***.println(" Property: " + p.getStringKey() + " = " +  p.getStringValue());  }  }  }  } |

## ImportExportExample Source Code

The example code below demonstrates how to import a Tincr Checkpoint into RS2, add some attributes to a few of the cells, and export the resulting circuit to a new Tincr Checkpoint. It, obviously, depends on the existence of a Tincr Checkpoint, the creation of which is not covered here. See the Tincr documentation for details.

|  |
| --- |
| **package** edu.byu.ece.rapidSmith.examples2;  **import** java.io.\*;  **import** edu.byu.ece.rapidSmith.RapidSmithEnv;  **import** edu.byu.ece.rapidSmith.design.\*;  **import** edu.byu.ece.rapidSmith.design.subsite.\*;  **import** edu.byu.ece.rapidSmith.device.\*;  **public** **class** CreateDesignExample {  /\*\*  \* A simple class to illustrate creating designs in RapidSmith2.  \* **@author** Brent Nelson  \*/  **public** **static** **void** main(String[] args) **throws** IOException{  // Load the cell library from the directory indicated by the part name  // (directory should be $RAPIDSMITH\_PATH/devices/artix7)  CellLibrary libCells = **new** CellLibrary(RapidSmithEnv.*getDefaultEnv*()  .getPartFolderPath("xc7a100tcsg324")  .resolve("cellLibrary.xml"));  System.***out***.println("Cell library loaded: cellLibrary.xml");  // Load the device file from the directory indicated by the part name  Device device = RapidSmithEnv.*getDefaultEnv*().getDevice("xc7a100tcsg324");  System.***out***.println("Device loaded: xc7a100tcsg324");    // Create a new empty CellDesign for the designated FPGA part  CellDesign design = **new** CellDesign("HelloWorld2", "xc7a100tcsg324");  // Create a new cell and add it to the current design. It is a LUT1 cell.  // Then, set the INIT property for the LUT cell (program the LUT)  Cell invcell = design.addCell(**new** Cell("lutcell", libCells.get("LUT1")));  invcell.updateProperty("INIT", PropertyType.***DESIGN***, "2'h1");  // Create a flip flop cell and set its properties  Cell ffcell = design.addCell(**new** Cell("ffcell", libCells.get("FDRE")));  ffcell.updateProperty("INIT", PropertyType.***DESIGN***, "INIT0");  ffcell.updateProperty("SR", PropertyType.***DESIGN***, "SRLOW");    // Create IOB's for the circuit's q output and for its clk input  Cell qbufcell = design.addCell(**new** Cell("qbuf", libCells.get("OBUF")));  Cell clkbufcell = design.addCell(**new** Cell("clkbuf", libCells.get("IBUF")));    // Create the Q wire and connect it up  CellNet qnet = design.addNet(**new** CellNet("qnet", NetType.***WIRE***));  qnet.connectToPin(ffcell.getPin("Q"));  qnet.connectToPin(invcell.getPin("I0"));  qnet.connectToPin(qbufcell.getPin("I"));  // Create and connect the wire between the LUT' output to the flip flop's D input  CellNet dnet = design.addNet(**new** CellNet("dnet", NetType.***WIRE***));  dnet.connectToPin(ffcell.getPin("D"));  dnet.connectToPin(invcell.getPin("O"));  // Create and connect the clock wire between the IOB output and the flip flop's input  CellNet clknet = design.addNet(**new** CellNet("clknet", NetType.***WIRE***));  clknet.connectToPin(clkbufcell.getPin("O"));  clknet.connectToPin(ffcell.getPin("C"));  // Let's place some of the cells into a slice  // Get the first SLICEL in the device  Site slice = device.getAllSitesOfType(SiteType.***SLICEL***)[0];    // Let's place the invcell on the A5LUT  design.placeCell(invcell, slice.getBel("A5LUT"));  // Let's place the ffcell on the AFF  design.placeCell(ffcell, slice.getBel("AFF"));  // Now, prettyprint what we have created  **for** (Cell c : design.getCells()) {  System.***out***.print("Cell: " + c.getName() + " " +  c.getLibCell().getName());  **if** (c.isPlaced())  System.***out***.println(" <<<Placed on: " + c.getAnchor() + ">>>");  **else** System.***out***.println();  **for** (CellPin cp : c.getPins()) {  System.***out***.println(" Pin: " + cp.getName() + " " +  cp.getDirection() + " " +  (cp.getNet()!=**null**?cp.getNet().getName():"<unconnected>"));  }  **for** (Property p : c.getProperties()) {  System.***out***.println(" Property: " + p.getStringKey() + " = " +  p.getStringValue());  }  }  }  } |

## Device Browser

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

This GUI program is located in the 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).

See below for a screenshot.

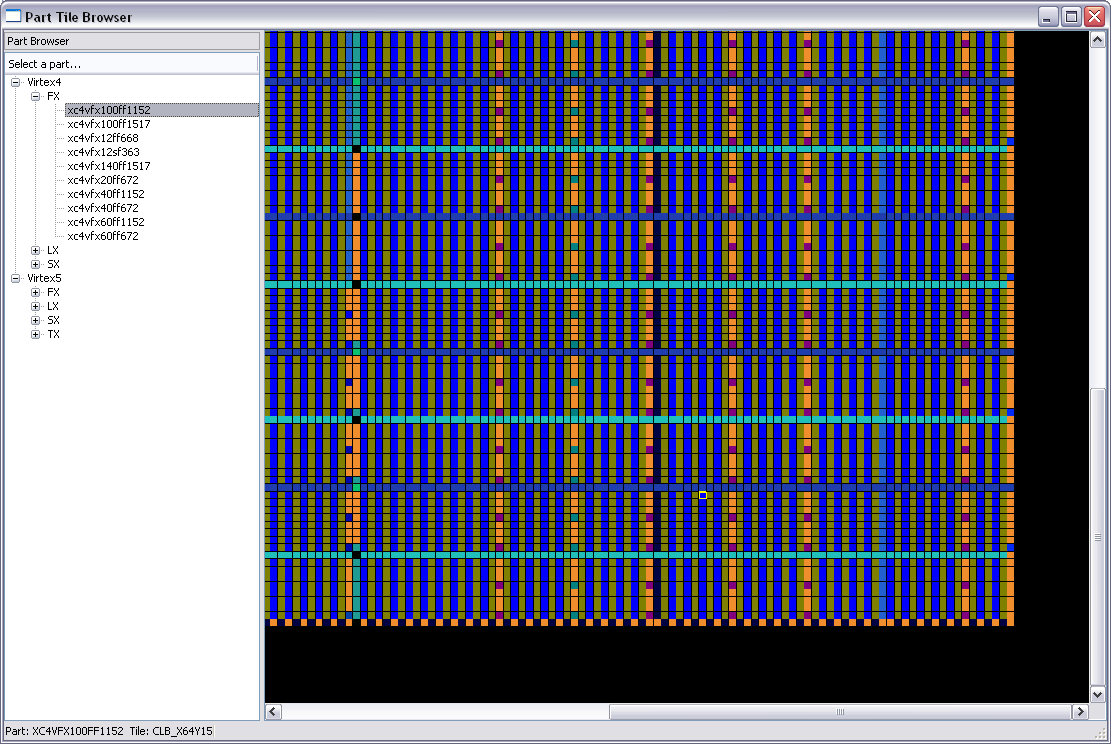


Figure 3 – Screen Shot of DeviceBrowser

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 below.

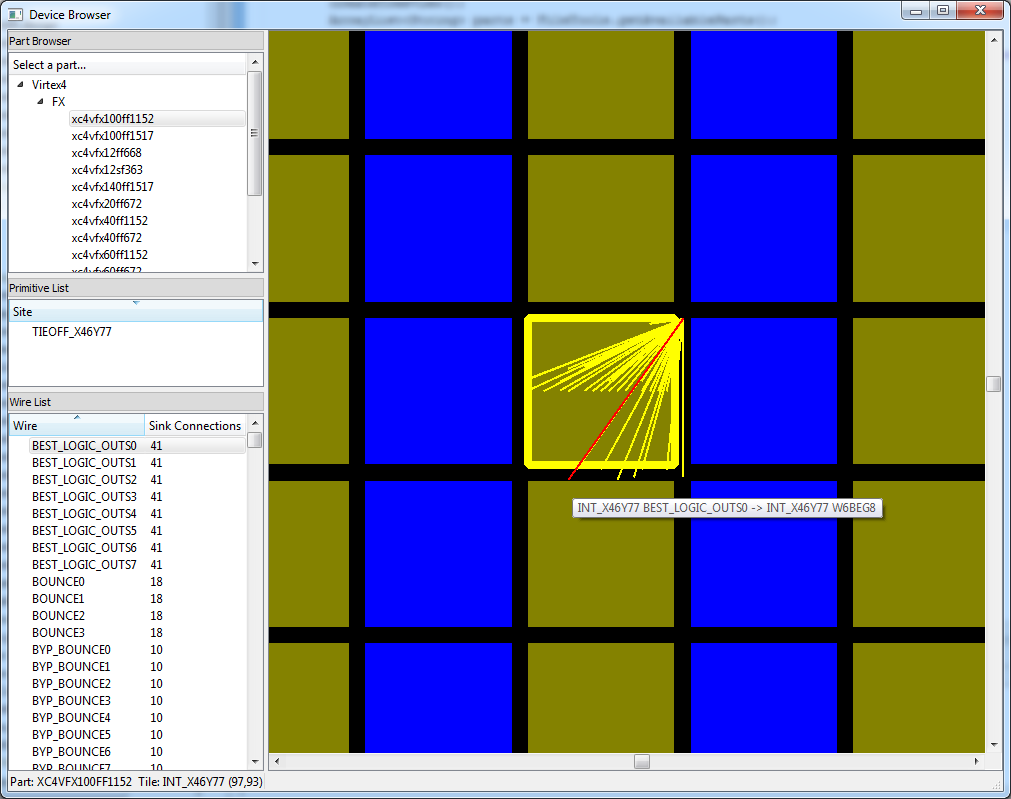


Figure 4 – DeviceBrowser Screen Shot Showing Wire Connections

# RS2 Structure

There are two main abstractions that developers need to be aware of; that of the *device* and that of the *design*.

## Devices in RS2

A device is defined in RS2 as a unique Xilinx FPGA part that includes package information but not speed grade (such as the xc7a100tcsg324 device included in the RS2 distribution). Each device contains specific information concerning its primitive sites, tiles, wires, IOBs, and PIPs that are available to realize designs. The device information is represented in RS2 in the **device** package. RS2 has significantly extended the original RapidSmith **device** class for its use as well as how device files are generated.

A *Device* object consists of a collection of *Tile*s, each of which contains one or more *Site*s. A *Site* contains one or more *Bel*s. *Site*s have *SitePin*s around their periphery as do *Bel* objects. The physical wires in the device are represented by objects of type *Wire*, *TileWire*, and *SiteWire*. However, the goal of RS2 is to largely hide the differences between these three object types and let the user simply deal with *Wire* objects.

## Designs in RS2

Designs in RS2 are essentially logical netlists. They are represented and stored in the data structures found in the **design.subsite** package. A *CellDesign* consists of a collection of *Cell* objects, interconnected by *CellNet*s. *CellNet*s connect to the *CellPin*s on *Cell*s. *Cell* objects have a name, properties, a link to the library cell they are an instantiation of, etc.

*Cell*s may be placed onto *Bel*s and the corresponding *CellPin*s mapped onto *BelPin*s. *CellNet*s, when physically routed, map onto one or more *RouteTree*s.

# Generating Device Files in RS2

RS2 is distributed with a single device file included (for an Artix7 device). The device files for this can be found in the ${RAPIDSMITH\_PATH}/devices/artix7directory and consist of a \*\_db.dat file and a \*\_info.dat file.

Additional device files for a family can be generated by a user. If these are additional device files for an already-supported device family, the process is relatively straightforward. The document: ${RAPIDSMITH\_PATH}/devices/doc/InstallingNewDevices.txt provides instructions on how to generate new device files for supported families. These should be adequate to generate new device files for families with existing support.

If you need device file support for a new family, the process is more difficult. Contact the authors for possible help in doing so.

# Placement in RS2

This chapter is intended to help users of RS2 understand how placement works in RS2 and in Vivado.

**Needs a total rewrite**

# Routing in RS2

This chapter is intended to help users and developers in understanding how routing resources are handled in RS2. It also illustrates how to build on the existing classes to create custom routers. RS2’s routing handling is significantly different from that found in RapidSmith.

**Needs a total rewrite, some can be lifted from the original RapidSmith documentation.**

# 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. This functionality came mainly from the documents distributed by Xilinx in the form of user guides, whitepapers and application notes. In other words, no proprietary information was used in the creation of the RapidSmith bitstream functionality.

The bitstream functionality from RapidSmith has been left intact in RS2 if users choose to use it with two caveats:

1. It has not been tested beyond Virtex 6.
2. It is not documented here – for documentation see the original RapidSmith User Guide distributed as a part of the RS2 distribution (in the doc directory).

# Legal and Dependencies

RS2 is a derivative of RapidSmith and is thus covered by the same license:

## RapidSmith Legal Text

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/>.

## 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://qt.nokia.com/downloads>

and more recent versions are available at:

[http:/qt.gitorious.org/qt-jambi](http://qt.gitorious.org/qt-jambi)

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.

# Appendix

Here is just a grouping of useful topics that may not fit in the rest of this document.

## Appendix C: Xilinx Family Names and Part Names

The part and family naming conventions used in RS2 largely follow those used in the Xilinx ISE tool partgen. RS2 includes an enum type called FamilyType for all known family architectures in the util package.

### Xilinx Part Names in RapidSmith

RS2 uses the part name pattern as produced by the Xilinx ISE partgen tool. These part names start with ‘X’ for Xilinx and are then often followed by a ‘C’ for commercial parts, ‘A’ for automotive parts, ‘Q’ for military grade parts, and ‘QR’ for space grade parts. The part names also include the package, however, because RS2 does not have any timing information, the speed grade is optional. Some examples are shown below:

|  |  |
| --- | --- |
| Examples of valid part names in RapidSmith | Examples of invalid part names in RapidSmith |
| * XC4VFX12FF668 * XC5LX110TFF1136-2 * XCV50BG256 | * Virtex 4 LX30 * XC5VSX35T-2FF665C * XC5VLX20T |

RS2 contains methods in util.RunXilinxTools to automatically run ISE’s partgen and parse its output to obtain part names of installed devices. The installer uses these methods in order to determine which parts are valid on the system. Other manipulation and conversion function for part names are found in the util.PartNameTools class.

### Xilinx Family Names in RS2

By using the FamilyType enum, it makes writing code easier when trying to figure out what family the current design is targeting. The util.PartNameTools has several methods to help identify a family type from a part name and also identify sub family names. With the most recent Xilinx tools (ISE 11.1 and above) do not support legacy devices (Spartan 2, Spartan 2E, Virtex, Virtex E, Virtex 2 and Virtex 2 Pro families) and must use ISE 10.1.03 or older to create XDLRC reports and import/export XDL design. PartNameTools contains a method to determine if a given family type is a legacy type:

/\*\*

\* This method determines which family types require the older version

\* of the Xilinx tools (10.1.03 or older).

\* **@param** familyType The family type to check.

\* **@return** True if this part requires older tools (10.1.03) or older, false

\* otherwise.

\*/

**public** **static** **boolean** isFamilyTypeLegacy(FamilyType familyType);

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. [↑](#footnote-ref-1)