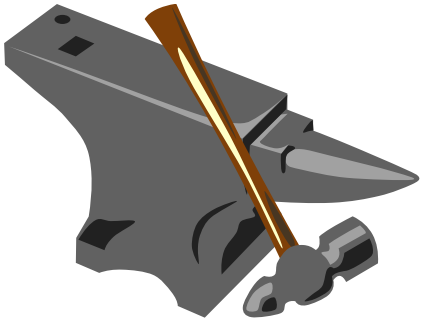

RAPIDSMITH

A Library for Low-level Manipulation of Partially Placed-and-Routed FPGA Designs

Technical Report and Documentation

Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan,
Brent Nelson, Brad Hutchings, and Michael Wirthlin

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

This work was supported by the I/UCRC program of the National Science Foundation under grant number 0801876.

TABLE OF CONTENTS

TABLE OF FIGURES

INTRODUCTION

What is RapidSmith?

The BYU RapidSmith project is a Java-based API that aims to provide academics with an *easy-to-use* platform to try out experimental ideas and algorithms on modern Xilinx FPGAs. RapidSmith is based on the Xilinx Design Language (XDL) which provides a human-readable file format equivalent to the Xilinx proprietary Netlist Circuit Description (NCD). With RapidSmith, researchers are able to import XDL/NCD, manipulate, place, route and export designs among a variety of possible design transformations possible. The RapidSmith project makes an excellent test bed to try out new ideas and algorithms for FPGA CAD research as code can quickly be written to take advantage of the APIs available.

RapidSmith also contains packages which can parse/export bitstreams (at the packet level) and represent the frames and configuration blocks in the provided data structures. It can parse, manipulate and export bitstreams according to Xilinx documented methods.

RapidSmith does not include any proprietary information about Xilinx FPGAs that is not publicly available.

Who Should Use RapidSmith?

RapidSmith is aimed at use by academics in all fields of FPGA CAD research. It is written in Java; therefore those using it will need to have a basic knowledge of programming and using Java. It also depends on some understanding of Xilinx FPGAs and XDL, however, this documentation hopes to bring people unfamiliar with these topics up to speed.

RapidSmith by no means is a Xilinx ISE replacement and **cannot** be used without a valid and current license to a Xilinx tools installation. RapidSmith should not be used for designs bound for commercial products and is offered mainly as a research tool.

Why RapidSmith?

The Xilinx ISE tools provide an `xdl` executable that allows conversion of NCD files to and from XDL which can then be parsed, manipulated and exported using RapidSmith. The `xdl` executable also creates special device files which are huge in size but contain useful detailed device data.

RapidSmith takes care of all of the parsing and detailed FPGA part information that can be cumbersome to use—alleviating the need to build such parsing tools by the researcher. RapidSmith creates special part files from these device files created by the ISE tools which can then be used by RapidSmith for design manipulation. This project provides researchers the ability to leverage all of the XDL work previously done and avoid duplicate work. This will enable researchers to have more time to focus on what matters most: their research of new ideas and algorithms.

Which Xilinx Parts does RapidSmith Support?

Virtex 4 and 5 families have been tested the most and are currently supported in all forms and applications. However, the XDLRC reports which can be extracted from the `xdl` executable are very regular and so RapidSmith can create device files for all modern Xilinx FPGA families. Therefore, RapidSmith supports (to a

lesser extent than Virtex 4 and Virtex 5) the following families: Spartan 2, Spartan 2E, Spartan 3, Spartan 3A, Spartan 3ADSP, Spartan 3E, Spartan 6, Virtex, Virtex E, Virtex 2, Virtex 2 Pro, and Virtex 6. To create the device files for the legacy devices, Xilinx ISE 10.1.03 or earlier will be needed. The table below provides a complete set of compatible features for each Xilinx FPGA family. See the Appendix for more family compatibilities (Spartan 6L, Virtex 6L, etc.).

Xilinx FPGA Family	Device Database, XDL Parsing, Manipulation & Export	Placement Capabilities	Router Capabilities	Bitstream Parsing, Manipulation & Export
Spartan 2	X	X		
Spartan 2E	X	X		
Spartan 3	X	X		
Spartan 3A	X	X		
Spartan 3ADSP	X	X		
Spartan 3E	X	X		
Spartan 6	X	X		
Virtex	X	X		
Virtex E	X	X		
Virtex 2	X	X		
Virtex 2 Pro	X	X		
Virtex 4	X	X	X	X
Virtex 5	X	X	X	X
Virtex 6	X	X		X

How is This Different than VPR?

VPR (Versatile Place and Route) 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 and VPR is that RapidSmith aims to provide the ability to target commercial Xilinx FPGAs. All features of these FPGAs which are accessible via XDL are available in RapidSmith. 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.

Some may argue that Java is a poor platform for FPGA CAD tool design as it has a reputation of being a memory hog and slow. We believe that these claims are overstated and that both speed and memory can be controlled to the point where this is not an issue.

LEGAL AND DEPENDENCIES

RapidSmith Legal Text

```
BYU RapidSmith Tools

Copyright (c) 2010-2011 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

RapidSmith 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

RapidSmith 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

RapidSmith 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

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

GETTING STARTED

Installation

Getting RapidSmith

You can download the latest release of RapidSmith from the sourceforge page here:

http://sourceforge.net/projects/rapidsmith/files/

You can also checkout the repository from SVN. We recommend using Eclipse, however, any IDE will work fine. To check out the RapidSmith project, the SVN URL is:

```
https://rapidsmith.svn.sourceforge.net/svnroot/rapidsmith/trunk rapidsmith
```

This repository contains all the files you need (including supporting JAR files). If you are using Eclipse as your IDE, it contains project files to get the project up and running with minimal effort.

Requirements for Installation

- 200 MB free disk space (for all Virtex 4 and Virtex 5 family devices)
  - NOTE: ~30-50 GB of free hard disk space needed during installation (XDLRC files are very large in size, the largest Virtex 6 parts take over 23GBs).
- Windows XP/Vista/7 or Linux (Mac OS X will work (see notes below), but Xilinx tools do not so we currently don't support it)
- Xilinx ISE 11.1 or higher (10.1.03 or earlier for legacy devices).
- JDK 1.6 (earlier versions may work, but have not been tested).
- Supporting JARs
  - INCLUDED: Caucho Hessian Implementation JAR v.4.0.6 (Used for compressing database device files)
  - INCLUDED: Qt Jambi (Qt for Java) for the Part Tile Browser example. Just adding the jars to the CLASSPATH variable is adequate.
  - INCLUDED: JOpt Simple for use by some examples in the bitstream tools packages.
  - OPTIONAL: JavaCC if the user wants to change the XDL design parser. There is also a good plugin for Eclipse for JavaCC which makes it easier to modify and compile .jj files.

Steps for Installation

1. Make sure the Xilinx tools and JDK are on your `PATH`.
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 RapidSmith 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 (this can be done automatically if the project is imported into an IDE such as Eclipse).
6. Generate the supporting device and enumeration files needed to run the various parts of RapidSmith. Please note that if you are generating both families of Virtex 4 and Virtex 5 parts, it will take **several**

hours and is best left to run **overnight** because of the time requirement. This only needs to be done once, however. To generate the part files, follow these steps:

a. Choose which parts you plan to use, or you can choose to do all parts in the Virtex 4 and Virtex 5 families (in the future, more parts will be compatible).
b. Run the installer for RapidSmith by executing the main method in the class `edu.byu.ece.rapidSmith.util.Installer`. This is accomplished by running the following at the command line:

```
Java -Xmx1600M edu.byu.ece.rapidSmith.util.Installer virtex4 virtex5
```

c. The previous command will take several hours. Some of the larger parts will also require a lot of heap memory to generate the part file (sometimes 1600M is too large for some computers, if it fails, try 1200M).
d. You can test if the file generation worked by looking in the appropriate folders (devices/virtex4 and devices/virtex5). You can also run the `BrowseDevice` class as a test to see if you are able to browse any of the parts that have just been created. You can run this with the following command:

```
Java edu.byu.ece.rapidSmith.util.BrowseDevice xc5vlx20tff323
```

Additional Notes for Mac OS X Installation

- The Xilinx tools do not run under Mac OS X and therefore, the installer would have to generate the files on Linux or Windows. However, once the files are created, they could be moved to a Mac OS X installation.
- 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:

```
<?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/workspace/rapidSmith</string>
	<key>CLASSPATH</key>
	<string>$CLASSPATH:/Users/[user name]/Documents/workspace/rapidSmith:/Users/[user
name]/Documents/workspace/rapidSmith:/Users/[user
name]/Documents/workspace/rapidSmith/jars/hessian-4.0.6.jar:/Users/[user
name]/Documents/workspace/rapidSmith/jars/qtjambi-4.6.3.jar:/Users/[user
name]/Documents/workspace/rapidSmith/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"

Overview

RapidSmith is organized into several packages (all packages are prefixed with "edu.byu.ece.rapidSmith"):

Package Name	Description
bitstreamTools.bitstream	Represents the packet view of a Xilinx bitstream. It contains classes to represent the header, packets, types, configuration registers

	and bitstream parsing and export facilities.
bitstreamTools.bitstream.test	Contains classes and scripts to test the bitstream package.
bitstreamTools.configuration	Provides an FPGA-level view of the configuration data in a bitstream using frames and contains an implementation of the frame address register.
bitstreamTools.configurationSpecification	This package contains specifications (column layouts) of all supported devices. It also defines different constructs such as block types, block sub types and part library functions.
bitstreamTools.examples	This package provides some examples on how to use the bitstream functionality in RapidSmith.
bitstreamTools.examples.support	Some support classes for the examples in the previous package.
design	Represents all of the constructs in XDL design files (Instances, Nets, PIPs, Modules, and Designs).
design.explorer	This is a GUI interactive explorer that allows the user to navigate through the various constructs in the design (Nets, Instances, Modules and Module Instances). It also has a tile map which allows the user to view the locations of various objects on the FPGA fabric. It also contains an experimental timing report parser to correlate timing information with a design.
design.parser	A JavaCC-based parser for XDL files which populate an instance of the Design class in the design package.
device	This package encompasses all the details of an FPGA device (part name, tiles, primitive sites, routing resources). All information about Xilinx parts is populated in device from the XDLRC files generated by the `xdl` executable.
device.browser	This program is an extension of the part tile browser found in the examples package. It allows the user to browse all of the installed parts and also navigate primitive sites as well as routing resources.
device.helper	Some classes to help in the creation of the device files.
examples	Some user examples of how to use RapidSmith.
gui	This package is used to help build graphical programs in Qt for RapidSmith. It contains

	useful and common widgets that can be put together easily using the Qt framework.
placer	This contains classes to place designs.
primitiveDefs	This is also populated from the XDLRC file. It is specific to a Xilinx family of parts (such as Virtex 4 or Virtex 5). It defines all primitives which are part of a Xilinx family of parts (SLICEL, SLICEM, RAMB16, …).
router	This contains classes to route designs and has a framework to help users of RapidSmith create new routers.
tests	This package contains test classes that will exercise various portions of RapidSmith.
timing	Currently, this is an experimental TWR parser that will parse timing reports output from Xilinx Trace (`trce`).
util	This contains miscellaneous support classes and utilities, including the installer.

`bitstreamTools.*` Package

Please see the chapter on Bitstreams in RapidSmith for details on the bitstream functionality in RapidSmith.

`design` Package

The design package has all the essential classes necessary to represent all kinds of XDL designs with classes to represent each type of XDL construct. Below in Figures 1, 2, and 3 are some basic illustrations of how the most common XDL constructs map into RapidSmith `design` classes:

Figure 1 - Design and Attribute classes

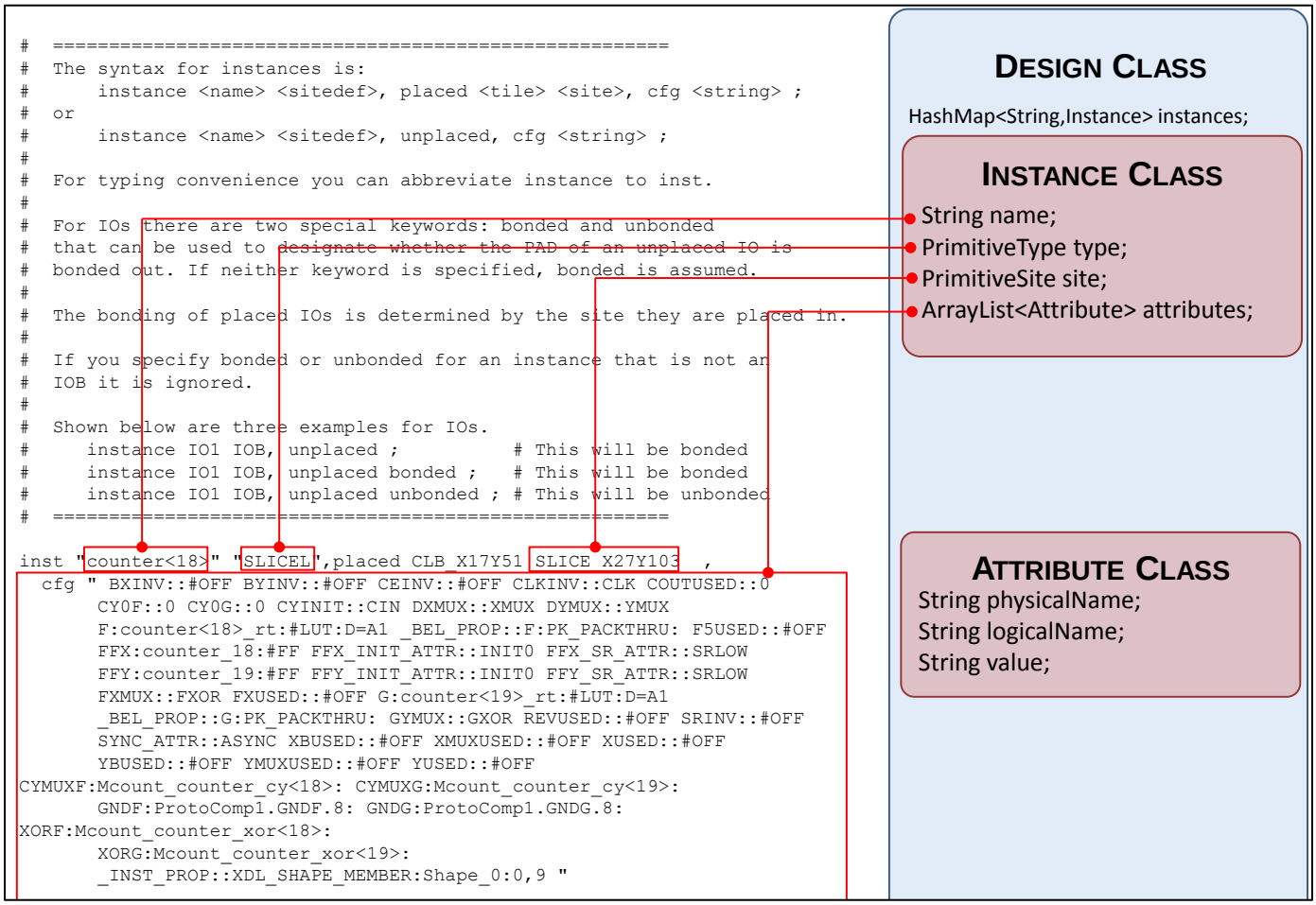

Figure 2 - Instance class

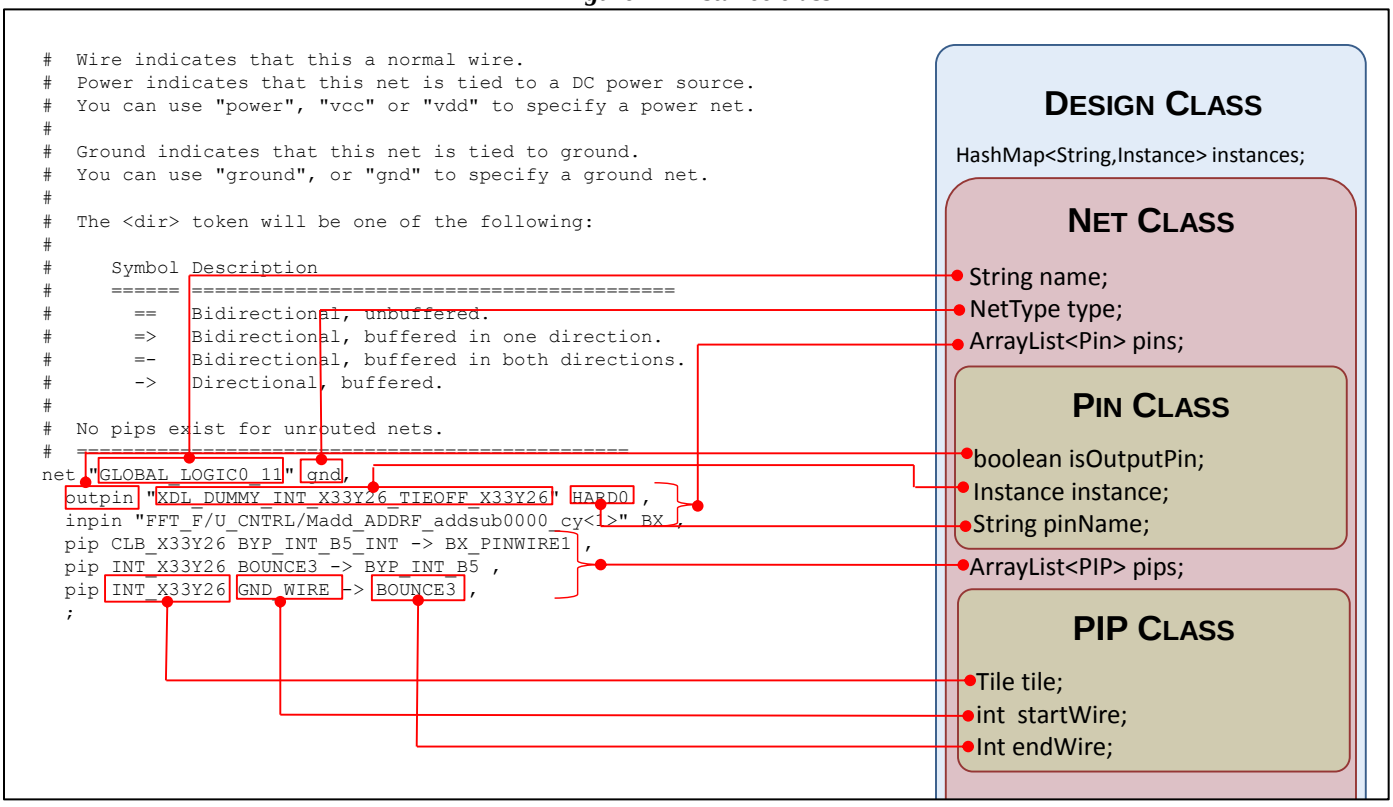

Figure 3 - Net, Pin and PIP Classes

There are other classes such as Module and ModuleInstance classes that abstract the macro-like property of XDL which will be explained later. There are also enumeration classes such as InstanceType which are an exhaustive list of all primitive types found in XDL and NetType which determines if a net is a WIRE, GND, or VCC.

`design.explorer` Package

The design explorer loads XDL design files and allows the user a GUI interface to the design members (Instances, Nets, Modules and Module Instances) with hyperlinks to the various other sites, tiles, modules and instances. Currently the design explorer is read only although there is potential for it to be able to modify designs. Below is a screenshot from the design explorer.

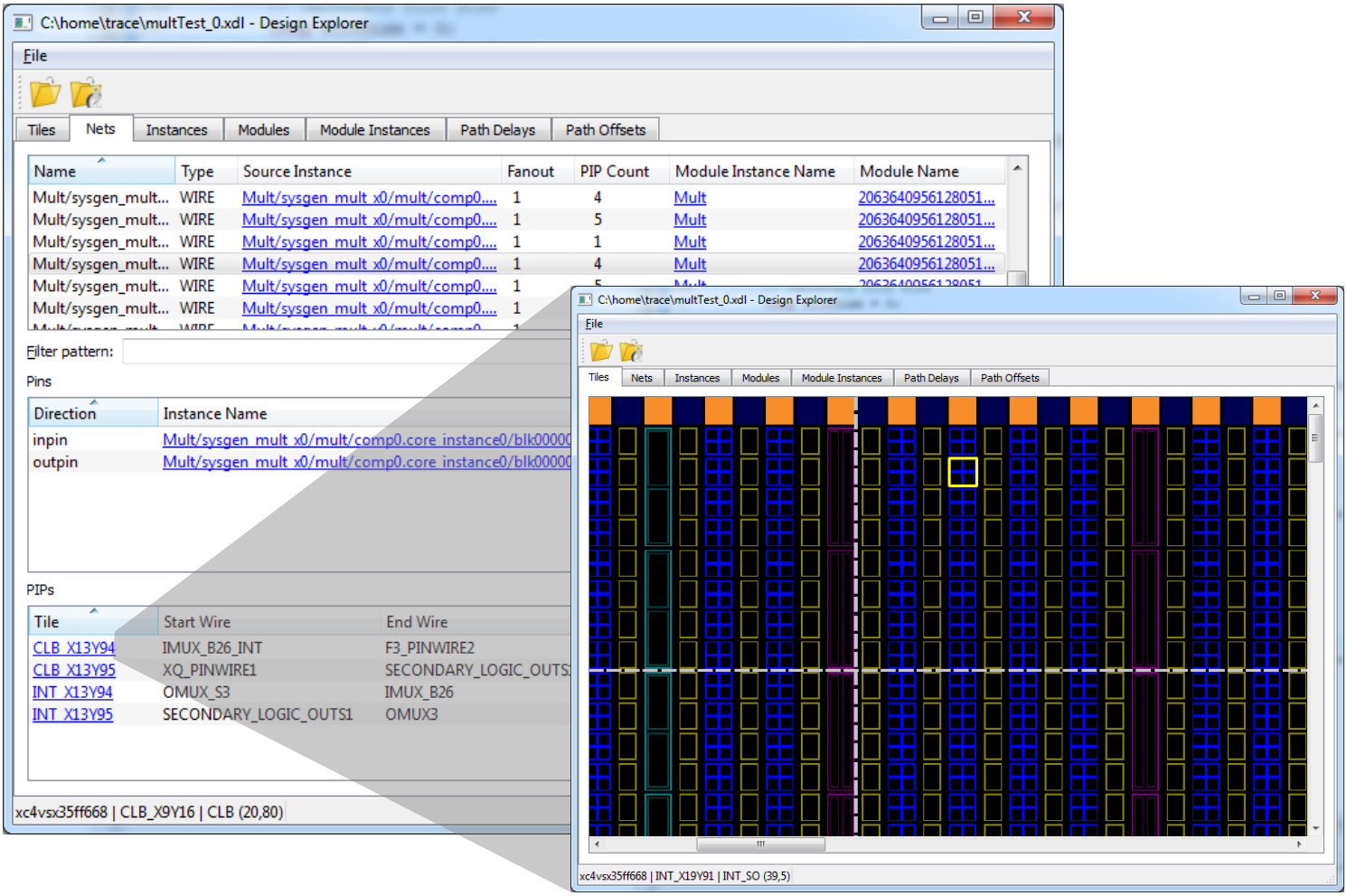

Figure 4 - Hyperlinks in Design Explorer

Recently, a timing report parser has been added which allows the timing information for a particular design to be loaded at the same time as a design. This can be done through the file menu or through the tool bar with a folder/clock icon. This timing report parser is still experimental and so mixed results may result depending on the type of timing report and the device used.

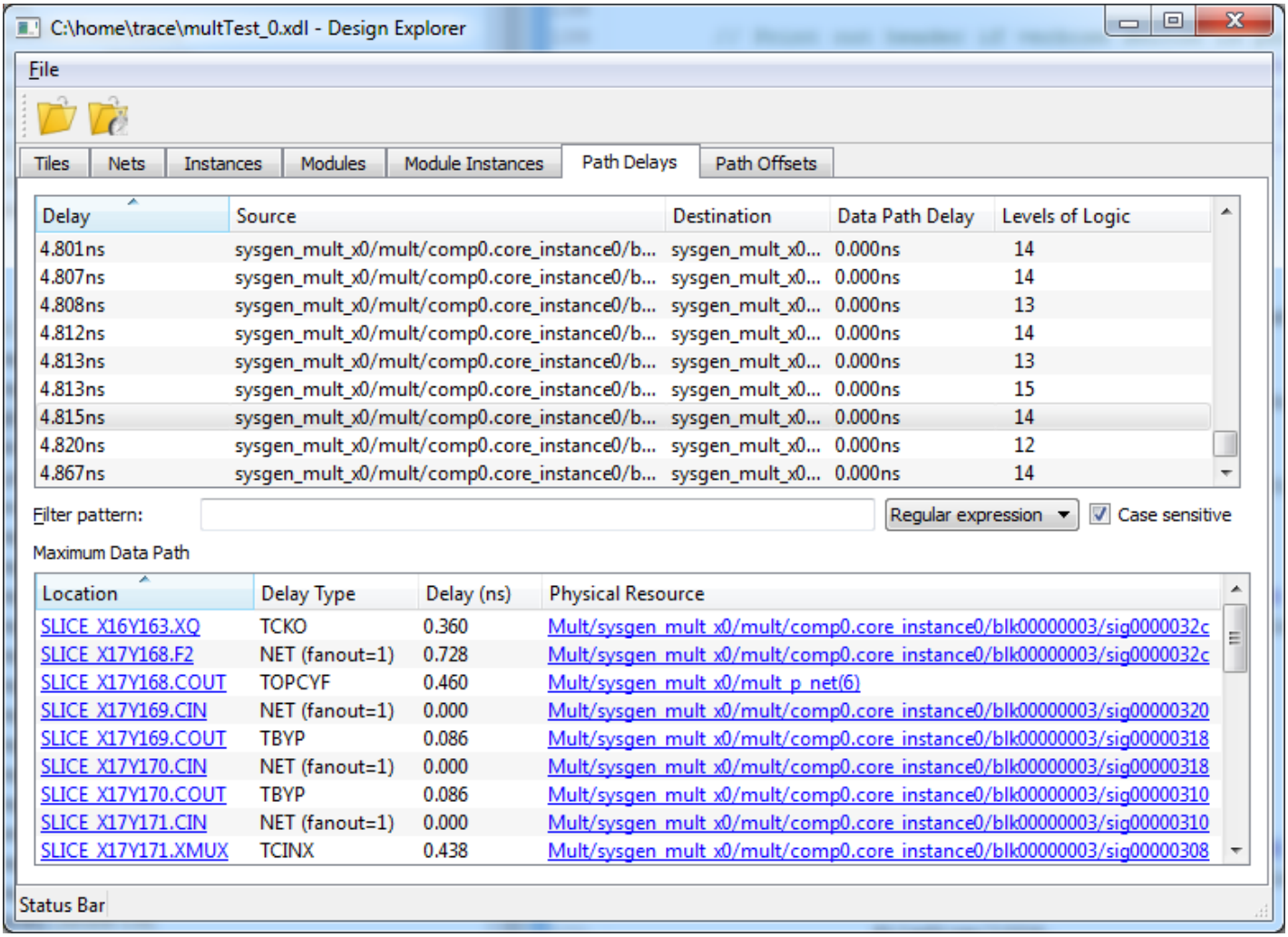

Figure 5 - Timing report loaded in Design Explorer

`design.parser` Package

The XDL parser parses xdl design or hard macro files and populates the Design class accordingly. The parser is a custom parser (previously a JavaCC parser was used but was ultimately abandoned) written exclusively for RapidSmith.

`device` Package

This package works closely with the design package in that the specific Device class is loaded when a design is loaded. The Xilinx XDLRC part descriptions partition the FPGA into a 2D grid of tiles. Each tile contains some mixture of primitive sites, wires and PIPs (Programmable Interconnect Points). Primitive sites are resource locations where XDL "inst" or instances of primitives are allowed to reside. Wires and PIPs provide wiring and routing resources information to connect the primitive instances together to form a complete design. With this information provided by Xilinx and leveraged by RapidSmith a number of different placement and routing algorithms can be constructed by leveraging the APIs in this package.

The device package also contains a class called WireEnumerator. All of the wires in a family are enumerated to an integer so they do not need to be stored as Strings. The WireEnumerator class helps translates wires from integers to Strings and vice versa. It also keeps track of important information about wires such as the type of wire (DOUBLE, HEX, PENT, …) and wire direction (NORTH, SOUTH, EAST, …) among other attributes.

`device.browser` Package

The device browser application allows the user to see a color coded tile array that allows them to browse any installed device. The primitive and wire lists are populated by double clicking a tile. The user can zoom in and out using the mouse wheel and can also pan by holding down the right mouse button while moving the mouse. See below for a screenshot of the application.

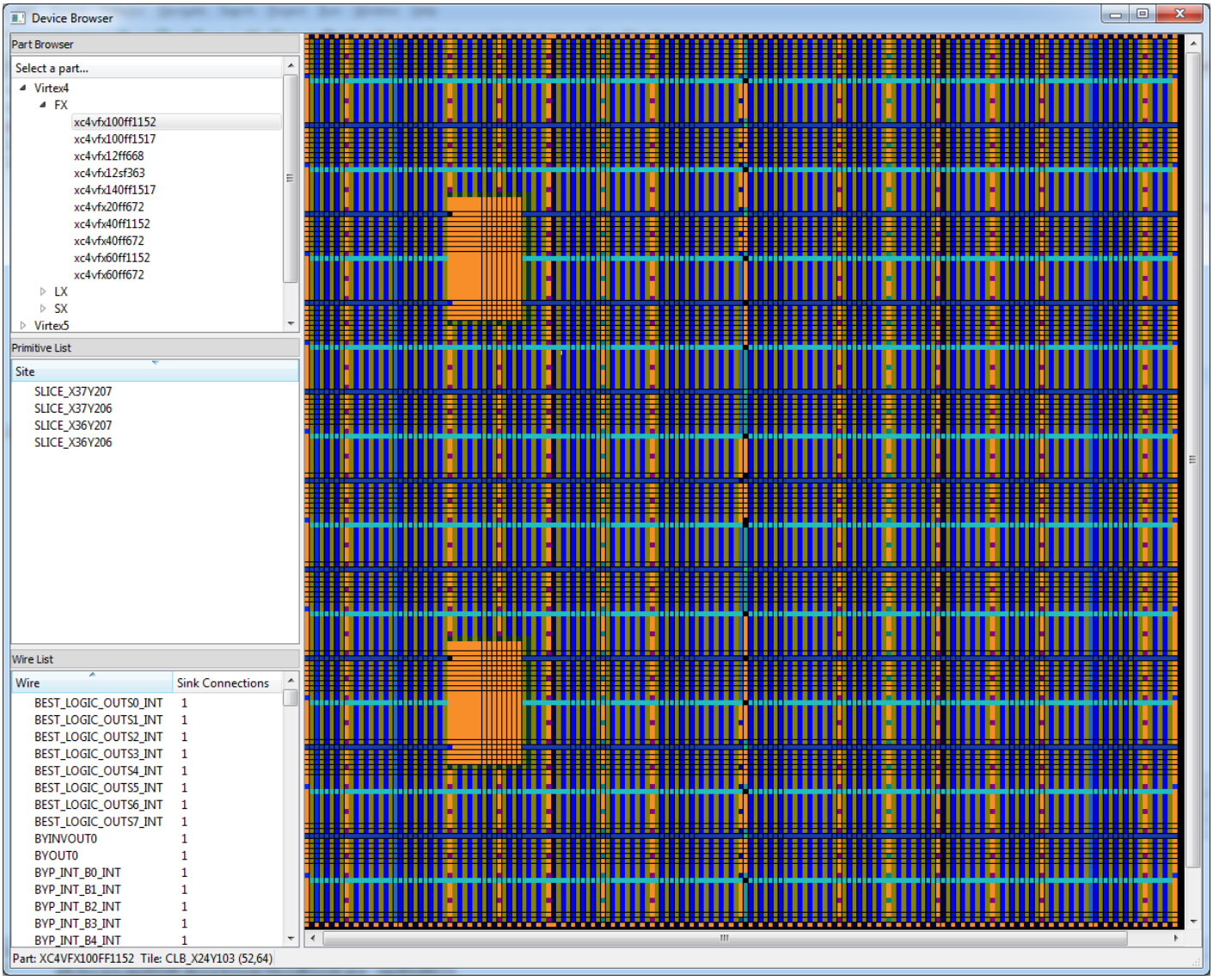

Figure 6 - Device Browser screenshot

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.

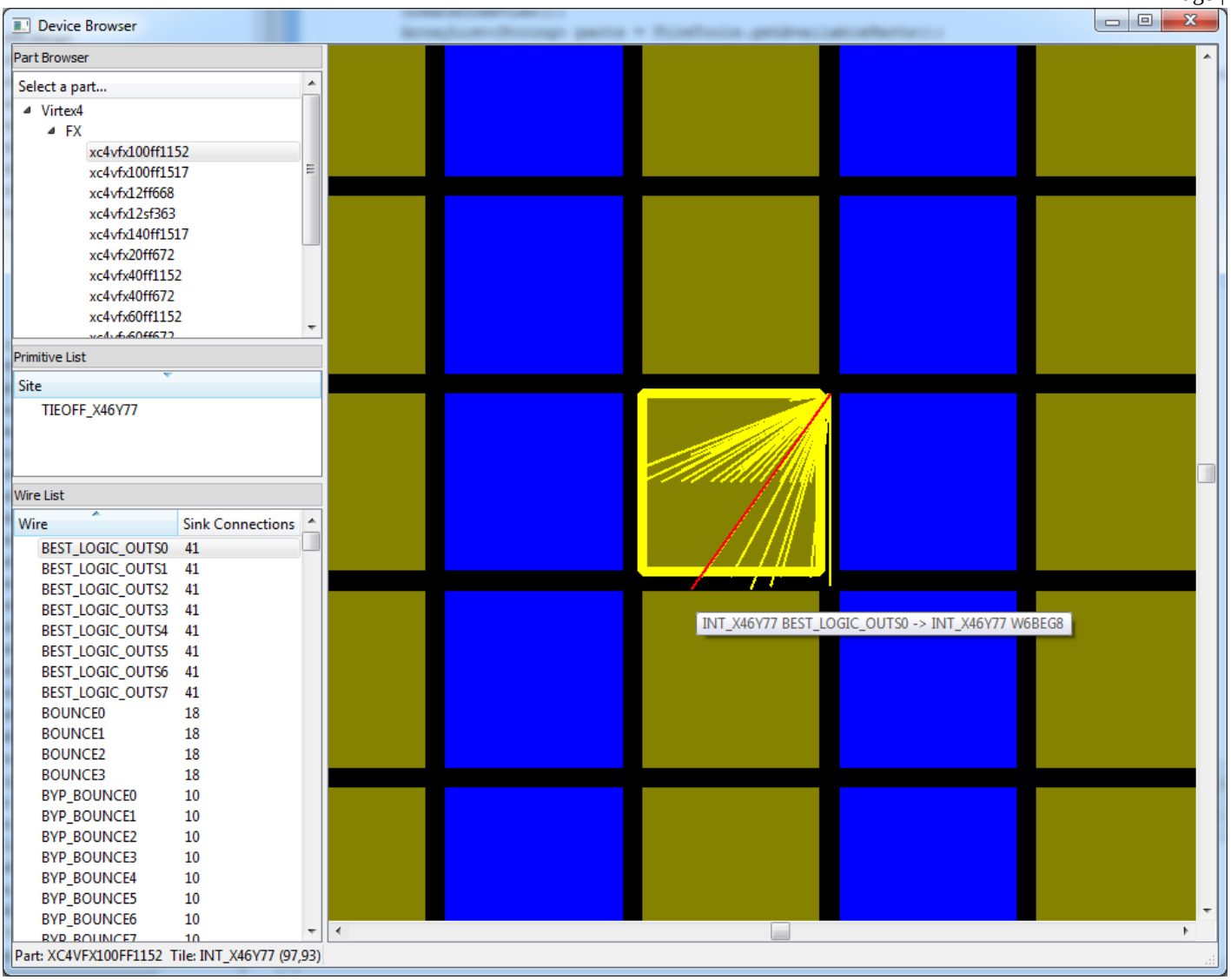

Figure 7 - Device browser screenshot showing wire connections

`device.helper` Package

This package contains special classes to help pack the device files smaller. They are generally used only during installation.

`examples` Package

This package contains some examples of how to get started with RapidSmith and some different ways of using the various APIs available.

`placer` Package

This package still has yet to be completed but will have an example of a placer.

`primitiveDefs` Package

In the XDLRC descriptions produced by the Xilinx 'xdl' executable, each copy has a section at the end called primitive_defs which has a list of primitive definitions for all types of primitives found in the part. The primitiveDefs packages makes that information available in a convenient data structure to access the attributes and various parameters the primitives can be configured with.

`router` Package

This package has an example of a basic router that routes Virtex 4 and Virtex 5 designs. It also contains an abstract class for which routers can be built upon.

`tests` Package

In order to help ensure correct functionality in RapidSmith as it grows, a tests package has been added to hold all of the different tests that can be performed to check for correct functionality with each new update. Currently, this package contains a class for testing the device, primitive defs and wire enumerator files (it is the class used to create the statistical information on RapidSmith files found in the Appendix).

`timing` Package

A new experimental package that contains a timing report parser (TWR files output from Xilinx Trace) has been added. This can parse a TWR file into a basic data structure contained in the timing package. This parser has been integrated into the design explorer application in RapidSmith.

`util` Package

This has miscellaneous classes used for support of all other packages. It is suggested to have the user browse the JavaDoc API descriptions to get a better feel for what is contained in the util package.

Examples

Hello World

To get started programming with RapidSmith, here is an example of a very simple program.

```
/*
 * Copyright (c) 2010 Brigham Young University
 *
 * This file is part of the BYU RapidSmith Tools.
 *
 * 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/>.
 *
 */
package edu.byu.ece.rapidSmith.examples;

import java.util.HashMap;
import edu.byu.ece.rapidSmith.design.*;
import edu.byu.ece.rapidSmith.device.*;

/**
 * A simple class to illustrate how to use some of the basic methods in RapidSmith.
 * @author Chris Lavin
 */
public class HelloWorld{
	public static void main(String[] args){
		// Create a new Design from scratch rather than load an existing design
```



```
Design design = new Design();

// Set its name
design.setName("helloWorld");

// When we set the part name, it loads the corresponding Device and
// WireEnumerator. Always include package and speed grade with the part name.
design.setPartName("xc4vfx12ff668-10");

// Create a new instance
Instance myInstance = new Instance();
myInstance.setName("Bob");
myInstance.setType(PrimitiveType.SLICEL);
// We need to add the instance to the design so it knows about it
design.addInstance(myInstance);
// Make the F LUT an Inverter Gate
myInstance.addAttribute(new Attribute("F","LUT_of_Bob","#LUT:D=~A1"));

// Add the instance to the design
design.addInstance(myInstance);

// This is how we can get the reference to the instance from the design,
// by name
Instance bob = design.getInstance("Bob");

// Let's find a primitive site for our instance Bob
HashMap<String, PrimitiveSite> primitiveSites =
                                design.getDevice().getPrimitiveSites();
for(PrimitiveSite site : primitiveSites.values()){
       // Some primitive sites can have more than one type reside at the site, such as
       // SLICEM sites which can also have SLICELs placed there.  Checking if the site
       // is compatible makes sure you get the best possible chance of finding a place
       // for bob to live.
       if(site.isCompatiblePrimitiveType(bob.getType())){
              // Let's also make sure we don't place bob on a site that is already used
              if(!design.isPrimitiveSiteUsed(site)){
                     bob.place(site);
                     System.out.println("We placed bob on tile: " + bob.getTile() +
                                   " and site: " + bob.getPrimitiveSiteName());
                     break;
              }
       }
}

// Another way to find valid primitive sites if we want to use an exclusive site type
PrimitiveSite[] allSitesOfTypeSLICEL =
                    design.getDevice().getAllSitesOfType(bob.getType());
for(PrimitiveSite site : allSitesOfTypeSLICEL){
       // Let's also make sure we don't place bob on a site that is already used
       if(!design.isPrimitiveSiteUsed(site)){
              bob.place(site);
              System.out.println("We placed bob on tile: " + bob.getTile() +
                            " and site: " + bob.getPrimitiveSiteName());
              break;
       }
}

// Let's create an IOB to drive our Inverter gate in Bob's LUT
Instance myIOB = new Instance();
myIOB.setName("input");
myIOB.setType(PrimitiveType.IOB);
design.addInstance(myIOB);
// These are typical attributes that need to be set to configure the IOB
// the way you like it
myIOB.addAttribute(new Attribute("INBUFUSED","","0"));
myIOB.addAttribute(new Attribute("IOATTRBOX","","LVCMOS25"));
// Another way to find a primitive site is by name, this is the pin name
```

```
            // that you might find in a UCF file
            myIOB.place(design.getDevice().getPrimitiveSite("C17"));

            // Let's also create a new net to connect the two pins
            Net fred = new Net();
            // Be sure to add fred to the design
            design.addNet(fred);
            fred.setName("fred");
            // All nets are normally of type WIRE, however, some are also GND and VCC
            fred.setType(NetType.WIRE);
            // Add the IOB pin as an output pin or the source of the net
            fred.addPin(new Pin(true,"I",myIOB));
            // Add Bob as the input pin or sink, which is the input to the inverter
            fred.addPin(new Pin(false, "F1", bob));

            // Now let's write out our new design
            // We'll print the standard XDL comments out
            String fileName = design.getName() +".xdl";
            design.saveXDLFile(fileName, true);

            // We can load XDL files the same way.
            Design inputFromFile = new Design();
            inputFromFile.loadXDLFile(fileName);

            // Hello World
            System.out.println(inputFromFile.getName());
        }
}
```

Hand Router

This is a command-line-based router than allows a user to route one net at a time and write out the design changes afterwards. Although not particularly useful as a router, it illustrates how RapidSmith could be used to build a router.

Part Tile Browser

This GUI will let you browse Virtex 4 and 5 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. See below for a screenshot.

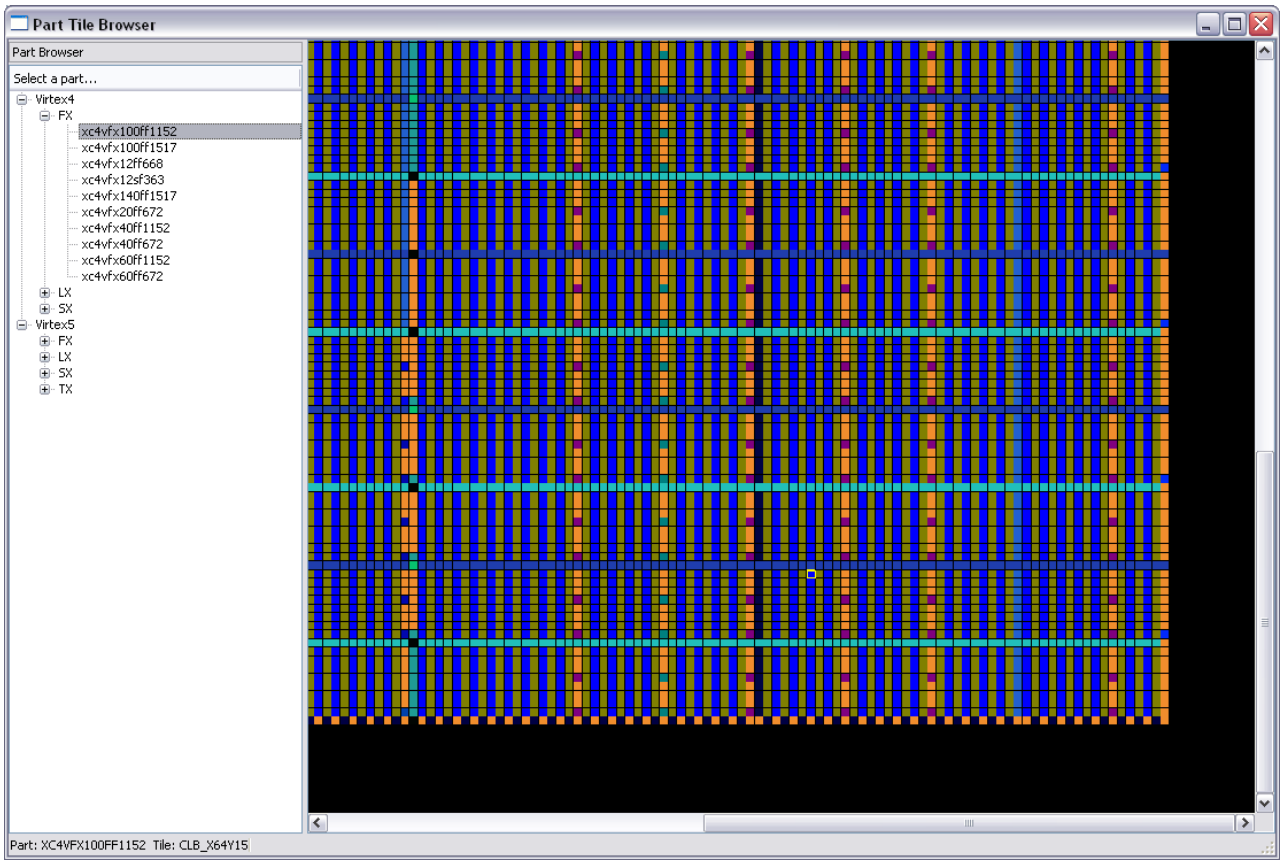

Figure 8: Screenshot of Part Tile Browser

UNDERSTANDING XDL

What is XDL?

XDL (or Xilinx Design Language, see ISE 6.1 documentation in `help/data/xdl` folder) is a human-readable ASCII format compatible with the more widely used Xilinx format NCD (or Netlist Circuit Description). XDL has most if not all the same capabilities of the NCD format and Xilinx provides an executable called `xdl` which can convert NCD designs to XDL and vice versa (run "`xdl -h`" for details). XDL and NCD are both native Xilinx netlist formats for describing and representing Xilinx FPGA designs. XDL is the interface used by RapidSmith to insert and extract design information at different points in the Xilinx design flow.

XDL can represent designs that are:

- Mapped (unplaced and unrouted)
- Partially placed and unrouted
- Partially placed and partially routed
- Fully placed and unrouted
- Fully placed and partially routed
- Fully placed and fully routed
- Contain hard macros and instances of hard macros
- A hard macro definition (equivalent to Xilinx NMC files)

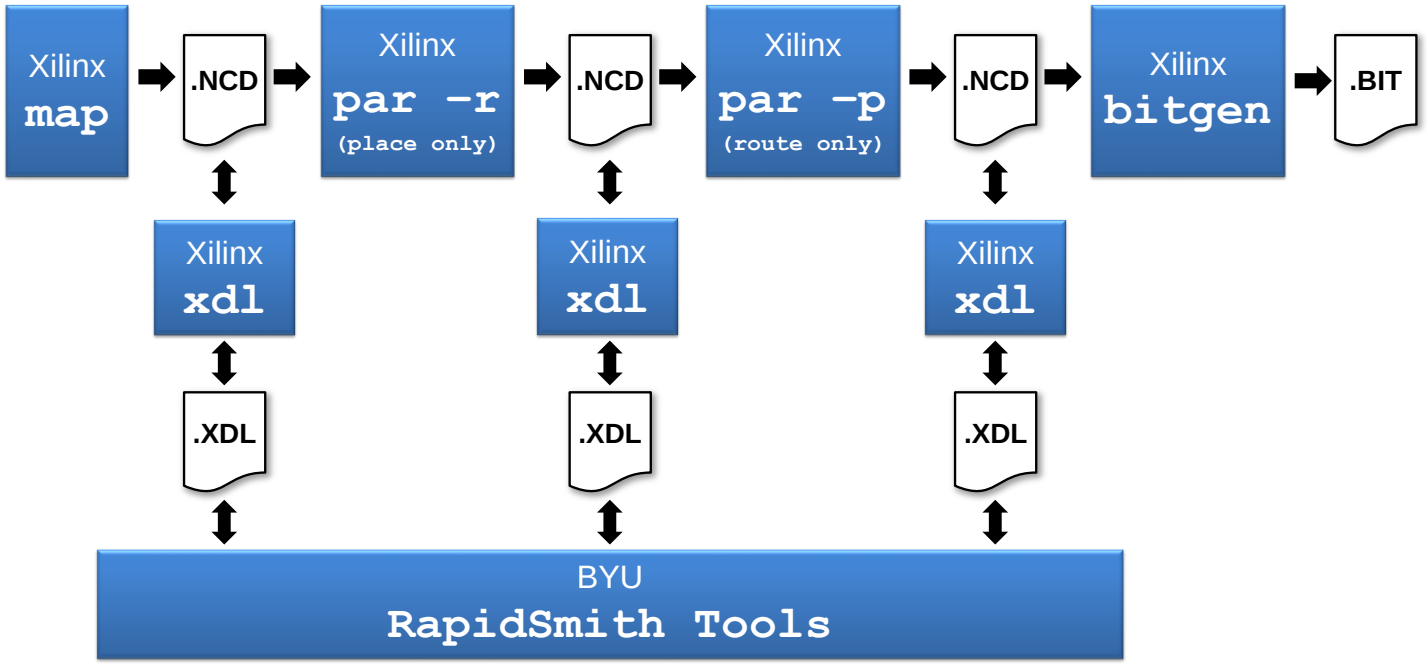

Figure 9: Block diagram of where XDL fits in CAD flow.

RapidSmith provides some Java methods that can perform the XDL/NCD conversion (by calling the `xdl` executable) from within Java in the `util.FileConverter` class. It also has methods for calling a number of Xilinx programs from within the RapidSmith environment.

The Xilinx `xdl` executable also contains options for generating report files (with extension .XDLRC) which contain descriptive information about a particular Xilinx part. XDLRC report files have a different format to that of XDL (as they describe an FPGA rather than a design) and depending on the options given can create enormous files (several gigabytes) of text but are quite complete in describing the primitive sites, routing resources and tile layout of a Xilinx FPGA. RapidSmith makes uses of these XDLRC files by generating them and parsing them into much smaller device files that can them be used with the rest of the RapidSmith API.

DISCLAIMER: The user must be aware that XDL is an externally **unsupported** format by Xilinx. All questions about XDL and any problems associated with XDL or this tool should **NOT** be addressed to Xilinx, but through the RapidSmith website and forum. The RapidSmith project is merely a tool to make use of the XDL software technical interface and cannot be used without a valid and current license for the Xilinx ISE tools. The RapidSmith project is at the mercy of Xilinx in the availability of XDL and will attempt to accommodate updates and changes to the interface as they arise.

Basic Syntax of XDL Files

XDL is a self-documenting file format in that each type of statement is generally preceded by comments that explain the syntax. Comments in XDL are denoted by using a '#' character at the beginning of a line. The '#' is also used in other constructs that are part of the language that do not fall on the beginning character of the line. In XDL there are four basic statements that make up the entire content of the file and description of the circuit.

Design Statement

The design statement (represented as the `design.Design` class) is included in every XDL file (even hard macros) and there is only one design statement in a file. It includes global information such as the design name and part name of the targeted FPGA. It can also contain a list of attributes in a 'cfg' string. Below is an example of a design statement.

```
# ======================================================
# The syntax for the design statement is:
# design <design_name> <part> <ncd version>;
# or
# design <design_name> <device> <package> <speed> <ncd_version>
# ======================================================
design "designName" xc4vfx12ff668-10 v3.2 ,
  cfg "
       _DESIGN_PROP::BUS_INFO:4:OUTPUT:gpio<3:0>
       _DESIGN_PROP::PIN_INFO:gpio<0>:/top/PACKED/top/gpio<0>/PAD:OUTPUT:3:gpio<3\:0>
       _DESIGN_PROP::PIN_INFO:gpio<1>:/top/PACKED/top/gpio<1>/PAD:OUTPUT:2:gpio<3\:0>
       _DESIGN_PROP::PIN_INFO:gpio<2>:/top/PACKED/top/gpio<2>/PAD:OUTPUT:1:gpio<3\:0>
       _DESIGN_PROP::PIN_INFO:gpio<3>:/top/PACKED/top/gpio<3>/PAD:OUTPUT:0:gpio<3\:0>
       _DESIGN_PROP::PK_NGMTIMESTAMP:1231972339";
```

Module Statement

Modules (represented as the `design.Module` class) are collections of instances and nets which can be described as hard macros if the instances are placed and nets are routed. A module will have a list of ports that determine the interface of the hard macro or module and each module will have its own list of instances and nets to describe the logic inside. An abbreviated module statement is shown below.


```
# =========================================================
# The syntax for modules is:
#     module <name> <inst_name> ;
#     port <name> <inst_name> <inst_pin> ;
#     .
#     instance ... ;
#     .
#     net ... ;
#     .
#     endmodule <name> ;
# =========================================================
module "moduleName" "anchorInstanceName" , cfg "_SYSTEM_MACRO::FALSE" ;
  port "portName1" "anchorInstanceName" "F2";
  port "portName2" "anotherInstanceInTheModule" "F4";
...
  inst "anchorInstanceName" "SLICEL", placed CLB_X14Y4 SLICE_X23Y8 , …
...
  net "aNetInsideTheModule" , …
...
endmodule "moduleName";
```

Instance Statement

The instance statement (represented as the `design.Instance` class), which begins with the keyword 'inst', is an instance of an FPGA primitive which can be placed or unplaced depending if a tile and primitive site location are specified. The instance also has a primitive type (such as SLICEL, SLICEM, DCM_ADV, …). Instance names should be unique in a design to avoid problems in RapidSmith. Instances are configured with a 'cfg' string which is a list of attributes that define LUT content, and other functionality. An example of an instance statement is shown below.

```
inst "instanceName" "SLICEL",placed CLB_X14Y4 SLICE_X23Y8 ,
  cfg " BXINV::#OFF BYINV::#OFF CEINV::#OFF CLKINV::#OFF COUTUSED::#OFF CY0F::#OFF
    CY0G::#OFF CYINIT::#OFF DXMUX::#OFF DYMUX::#OFF F::#OFF F5USED::#OFF FFX::#OFF
    FFX_INIT_ATTR::#OFF FFX_SR_ATTR::#OFF FFY::#OFF FFY_INIT_ATTR::#OFF
    FFY_SR_ATTR::#OFF FXMUX::#OFF FXUSED::#OFF
    G:DCM_AUTOCALIBRATION_DCM_clock/DCM_clock/md/RSTOUT1:#LUT:D=A1
    _BEL_PROP::G:LIT_NON_USER_LOGIC:DCM_STANDBY GYMUX::#OFF REVUSED::#OFF SRINV::#OFF
    SYNC_ATTR::#OFF XBUSED::#OFF XMUXUSED::#OFF XUSED::#OFF YBUSED::#OFF
    YMUXUSED::#OFF YUSED::0 "
   ;
```

Net Statement

The net statement (represented as the `design.Net` class) are the nets that describe inputs/outputs and routing of nets in a design. Nets can be of 3 different types: GND, VCC, or WIRE. The GND and VCC keyword must be present to mark a net as being sourced by ground or power, the keyword WIRE is not required as it is the default type. Nets also must have a unique name when compared with all other nets. Nets have two sub components to describe them: pins and PIPs. An example of a net statement is shown below.

Pins (represented as the `design.Pin` class) define the source and one or more sinks within the net. A pin is uniquely identified by the name of the instance where the pin resides as well as the internal name of the pin on this instance. It also has a direction of being an 'outpin' (source) or an 'inpin' (sink). A net can only have one source or 'outpin' in the net.


```
net "netName" ,
  outpin "instanceNameOfSourcePin" Y ,
  inpin "instanceNameOfSinkPin" RST ,
  pip CLB_X14Y4 Y_PINWIRE1 -> BEST_LOGIC_OUTS5_INT ,
  pip DCM_BOT_X15Y4 SR_B0_INT3 -> DCM_ADV_RST ,
  pip INT_X14Y4 BEST_LOGIC_OUTS5 -> OMUX8 ,
  pip INT_X15Y5 OMUX_EN8 -> N2BEG0 ,
  pip INT_X15Y7 N2END0 -> SR_B0 ,
  ;
```

PIPs (programmable interconnect points, represented by the `design.PIP` class) define routing resources used within the net to complete routing connections between the source and sinks. A PIP is uniquely described as existing in a tile (ex: INT_X2Y3) and two wires with a connection between them. Almost all PIPs are unidirectional (‘->’) in that they can only go in one direction. Long lines are the one exception to that rule as they are bidirectional and are denoted by using a ‘-=’ symbol, however RapidSmith uses the ‘->’ symbol for all PIPs as this does not cause the xdl converter any problems.

Basic Syntax of XDLRC Files

XDLRC files are report files generated by the Xilinx `xdl` executable. During installation, RapidSmith will create XDLRC files and parse them for their pertinent information and then pack it into small device files that can be used later with the tool. Each construct found in XDLRC files and the corresponding RapidSmith representation is described in the remainder of this subsection.

Tiles

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

Tiles (represented in the `device.Tile` class) are the building blocks of Xilinx FPGAs. Every FPGA is described as 2D array or grid of tiles laid out like a checker board (this can be seen also in the Part Tile Browser example). 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)
          )
```


Primitive sites (represented in the `device.PrimitiveSite` class) are declared in tiles. A primitive site is a location on the FPGA that allows for an instance of that primitive type (primitive types are enumerated in the `device.PrimitiveType` enum) to reside. For example, in the declaration of a SLICEL primitive site above, any SLICEL instance can be placed at that site. A primitive site has a unique name (`SLICE_X9Y127`) and type (`SLICEL`). However, in some cases, more than one primitive type is compatible with a given primitive site. One example of this is the primitive type SLICEM (Virtex 4 slices that contain RAM functionality in the LUT among other enhancements to the SLICEL type) is a superset of SLICEL functionality. Therefore, a SLICEL primitive instance can be placed in a SLICEM primitive site. RapidSmith allows the developer to determine if a given site is compatible in the `device.PrimtiveSite` class using the method `isCompatiblePrimitiveType(PrimitiveType otherType)`.

Primitive site declarations in XDLRC also 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`. RapidSmith provides mechanisms to translate between these two names in the `device.PrimitiveSite` class with the method `getExternalPinName(String internalName)`.

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)
          )
```

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` connected to 5 other neighboring tiles. These connections (denoted by 'conn') are described using the unique tile name and wire name of that tile to denote connectivity. These connections are not programmable, but hard wired into the FPGA. Inter-tile connections are not programmable, however, intra-tile connections (PIPs, see below) are. RapidSmith must represent the routing resources of Xilinx FPGAs very carefully as a significant fraction of the FPGA description is routing. Therefore, the wire names (such as `E2BEG0`, …) are enumerated into integers or Java primitive `int` data types using the `device.WireEnumerator` class. The `WireEnumerator` class keeps track of what integer value goes with each wire name and allows for significant compaction of the FPGA routing description.

The wire connections are described using a relative tile offset to reuse data structure elements. The class used to represent these wires and corresponding connections is in the `device.Wire` class.

PIP

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

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` if the PIP exists in a net's PIP list in a given design.

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." The connections are also represented in the `device.Wire` class as connections with a special flag denoting the connection as a PIP.

Primitive Definitions

At the end of every XDLRC file (regardless of verboseness) is a list of all primitive definitions for the Xilinx part. Primitive definitions are used mainly for reference and are reflected in the `primitiveDefs`.* package. In more recent Xilinx parts, some of the primitive definitions have been found to lack some information which may require special handling in RapidSmith. Currently, the primitive definitions are not widely used in RapidSmith.

RAPIDSMITH STRUCTURE

This section details much of the complexity and theory behind the structure of RapidSmith. There are two main abstractions that developers need to be aware of; that of the device and design. A hierarchy of classes within RapidSmith can be seen in Figure 6 below.

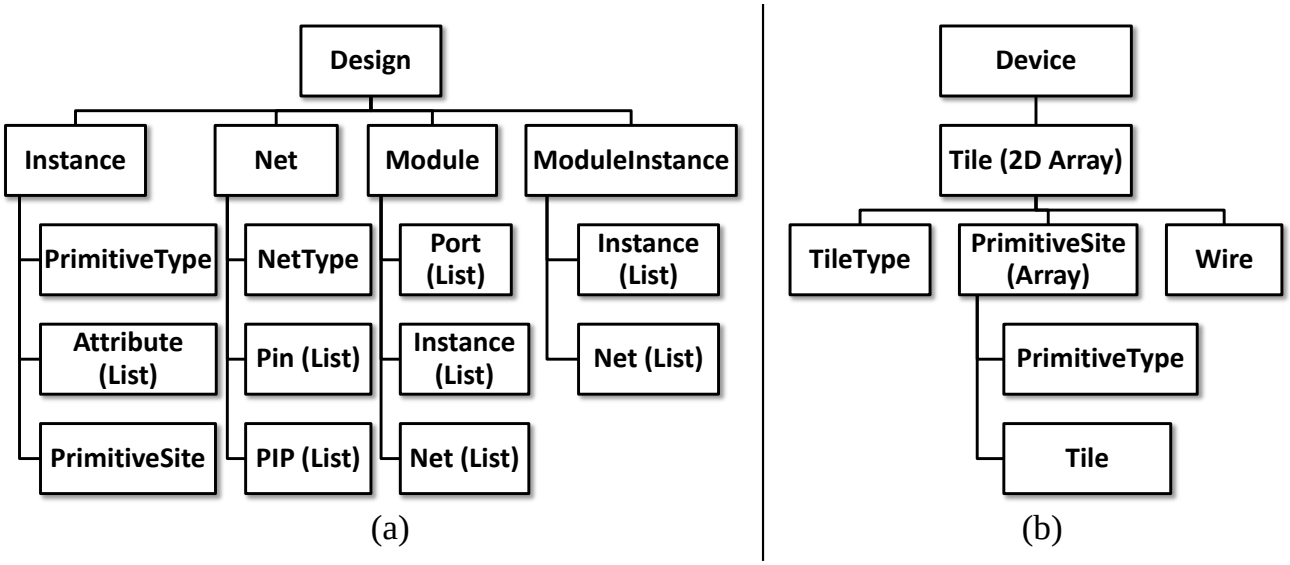

Figure 10: (a) The classes involved in defining a design in RapidSmith, (b) The major classes involved representing a device.

A RapidSmith Design

Designs in RapidSmith are represented and stored in the data structures found in the `design` package. The classes found in this package closely follow the constructs found in XDL design files to make the classes easier to follow and make the abstraction understandable to those who are familiar with XDL. For those who have less experience with XDL, see the previous section on understanding XDL.

Loading Designs

There is typically only one way to load a design with RapidSmith and that is to create a new design and call the method `loadXDLDesign(String fileName)` on that instance of the design. An NCD file can also be loaded indirectly by using methods in the `util.FileConverter` class which enables the conversion of an existing NCD file to XDL by calling the Xilinx `xdl` executable. Example code of how this could be done is shown below:

```
// Loading an XDL design
Design myDesign = new Design();
myDesign.loadXDLFile("myDesign.xdl");

// Loading an existing NCD file by converting it to XDL
String ncdFileName = "myOtherDesign.ncd";
String xdlFileName = FileConverter.convertNCD2XDL(ncdFileName);
if(xdlFileName == null){
      MessageGenerator.briefErrorAndExit("ERROR: Conversion of " +
                  ncdFileName + " to XDL failed.");
}
Design otherDesign = new Design();
otherDesign.loadXDLFile(xdlFileName);
```


When loading a design, one must be conscious of everything that gets loaded. In RapidSmith, a JavaCC-based XDL parser (found in the `design.parser` package) reads and parses the given XDL file and populates the instance of the Design class respectively. When the parser populates the design with the design part name, it causes the corresponding device file of the Xilinx part as well as the wire enumerator to be loaded and populates the design. This is done be default because the primitive sites and wires in the design reference those same resources in the `device` class.

Saving Designs

RapidSmith also has a method to save designs in the XDL format similar to the method for loading them. In a similar manner, the saved XDL file can be converted to NCD using the `FileConverter` class. Very little error checking is made when loading and saving XDL designs, but a good test would be the conversion to NCD as Xilinx runs several DRCs when the design is converted.

A RapidSmith Device

A device is defined in RapidSmith as a unique Xilinx FPGA part that includes package information but not speed grade (such as xc4vfx12ff668). Each device contains specific information concerning its primitive sites, tiles, wires, IOBs, and PIPs that are available to realize designs. This information is made available through the Xilinx executable `xdl` in `-report` mode. See the previous section on XDLRC for more details on these device resources.

Device

During the initial setup of RapidSmith, the Installer creates fully verbose XDLRC files (`$xdl -report -pips -all_conns <partName>`) for each device specified as command line parameters. After the creation of each XDLRC file, they are parsed, compacted by the Installer, and a device file is generated for later use. These device files are placed in `$(RAPIDSMITH_PATH)/devices/familyName/partName_db.dat` and then the corresponding XDLRC file is deleted as they can be several gigabytes in size. These device files make accessing device information about a specific FPGA part much more convenient than a gigantic text file. Most of the device files are just a few megabytes or less and can be loaded in a few seconds or less. RapidSmith uses a custom form of serialization as well as a compression library to make sure the device files are small and load quickly.

Wire Enumerator

In order to make the device files small, each uniquely named wire is assigned to an integer as enumeration. This avoids moving strings around in memory which would be costly in terms of space and comparison times. RapidSmith has a class called `WireEnumerator` which enumerates all uniquely named wires in an FPGA family and has methods to convert to and from the wire name and enumeration or enum for short. It also stores information about each wire such as a direction or type which can be useful in building a router. Note that wires with the same name can occur several times within a device and they are uniquely identified not only by their name, but also by the tile in which they are present.

In order to create the wire enumeration files, a subset of XDLRC files must be parsed so that a complete set of wires can be enumerated. This is automatically done by the installer and the files are placed in `$(RAPIDSMITH_PATH)/devices/familyName/wireEnumerator.dat`. Only one wire enumerator is needed per FPGA family.

Memory and Performance

Although Java typically has a reputation for being slow and a memory hog, RapidSmith has been able to create a very good device representation that is compact and fast loading, even for some of the largest parts offered by Xilinx. Performance and memory footprint figures are shown in the tables below for the device database file and data structure. These results were taken on Windows XP Professional SP3 (32-bit) with the Oracle (previously Sun) Java JVM build 1.6.0_21-b07. The workstation used was an HP Compaq dc7800 CMT with an Intel Core 2 Duo 3.0 GHz (E6850) with 4GB RAM and 500GB SATA hard drive. Note that the `Device` and `WireEnumerator` classes are both required before loading any design in RapidSmith. For a more extensive list of performance figures of all RapidSmith-compatible devices, see the Appendix.

Virtex 4 Device Performance and Memory Usage

Part Name	Heap Memory Footprint	File Size on Disk	Load Time from Disk
FX12	38.7 MB	599.0 KB	0.48 secs
FX20	63.4 MB	924.9 KB	0.79 secs
FX40	94.6 MB	1159.0 KB	1.02 secs
FX60	103.6 MB	1206.8 KB	1.42 secs
FX100	120.1 MB	1358.0 KB	1.51 secs
FX140	140.4 MB	1546.3 KB	1.63 secs
LX15	34.5 MB	231.8 KB	0.38 secs
LX25	37.3 MB	287.7 KB	0.46 secs
LX40	42.8 MB	348.5 KB	0.51 secs
LX60	64.0 MB	464.6 KB	0.80 secs
LX80	74.1 MB	596.7 KB	0.96 secs
LX100	74.2 MB	701.7 KB	0.97 secs
LX160	109.5 MB	875.6 KB	1.44 secs
LX200	115.5 MB	1010.6 KB	1.53 secs
SX25	62.8 MB	373.9 KB	0.76 secs
SX35	64.7 MB	441.0 KB	0.78 secs
SX55	74.8 MB	539.4 KB	0.93 secs

Virtex 5 Device Performance and Memory Usage

Part Name	Heap Memory Footprint	File Size on Disk	Load Time from Disk
FX30T	62.9 MB	781 KB	0.86 secs
FX70T	83.9 MB	898.7 KB	1.03 secs
FX100T	107.6 MB	1014.3 KB	1.46 secs
FX130T	111.8 MB	1058.2 KB	1.5 secs
FX200T	129.3 MB	1227.3 KB	1.62 secs
LX20T	39.1 MB	497.4 KB	0.52 secs
LX30	41.4 MB	381 KB	0.53 secs
LX30T	46.1 MB	558.8 KB	0.56 secs
LX50	42.8 MB	417.4 KB	0.55 secs
LX50T	47.7 MB	586.7 KB	0.79 secs

LX85	71.6 MB	534.8 KB	0.98 secs
LX85T	77.4 MB	706.6 KB	1.01 secs
LX110	74.4 MB	588.9 KB	1.01 secs
LX110T	81 MB	761.9 KB	1.03 secs
LX155	102.6 MB	716.3 KB	1.43 secs
LX155T	113.5 MB	893.8 KB	1.47 secs
LX220	138.3 MB	862.4 KB	1.64 secs
LX220T	143.5 MB	1038.7 KB	1.87 secs
LX330	147 MB	1068.4 KB	1.88 secs
LX330T	152.9 MB	1250 KB	1.91 secs
SX35T	59.4 MB	596.5 KB	0.84 secs
SX50T	61.2 MB	630.1 KB	0.84 secs
SX95T	85.3 MB	754.4 KB	1.07 secs
SX240T	127.2 MB	1135.9 KB	1.64 secs
TX150T	89.9 MB	871.2 KB	1.1 secs
TX240T	122.4 MB	1111.2 KB	1.57 secs

Virtex 6 Device Performance and Memory Usage

Part Name	Heap Memory Footprint	File Size on Disk	Load Time from Disk
CX75T	54 MB	583.9 KB	0.79 secs
CX130T	63.4 MB	714.8 KB	0.87 secs
CX195T	77.5 MB	853.2 KB	1.06 secs
CX240T	80.6 MB	937.3 KB	1.08 secs
HX250T	82.3 MB	973 KB	1.1 secs
HX255T	88.4 MB	1029.6 KB	1.12 secs
HX380T	97.8 MB	1287.3 KB	1.67 secs
HX565T	125.1 MB	1658.5 KB	1.81 secs
LX75T	54.1 MB	582.6 KB	0.79 secs
LX130T	63.3 MB	708.8 KB	0.87 secs
LX195T	77.3 MB	848.9 KB	1.07 secs
LX240T	80.6 MB	934.8 KB	1.08 secs
LX365T	107.2 MB	1186.4 KB	1.62 secs
LX550T	117.9 MB	1550.5 KB	1.8 secs
LX760	135.1 MB	1755.5 KB	2.48 secs
SX315T	106.5 MB	1157.1 KB	1.61 secs
SX475T	117.4 MB	1505.5 KB	1.79 secs

Spartan 6 Device Performance and Memory Usage

Part Name	Heap Memory Footprint	File Size on Disk	Load Time from Disk
LX100	28.7 MB	530.1 KB	0.43 secs
LX100T	29.7 MB	620.1 KB	0.43 secs

LX150	34.6 MB	623.8 KB	0.52 secs
LX150T	35.7 MB	713.0 KB	0.52 secs
LX16	11.6 MB	234.8 KB	0.15 secs
LX25	20.2 MB	307.7 KB	0.28 secs
LX25T	21.1 MB	362.1 KB	0.26 secs
LX45	20.6 MB	359.2 KB	0.26 secs
LX45T	20.8 MB	421.8 KB	0.27 secs
LX4	13.4 MB	289.0 KB	0.20 secs
LX75	25.2 MB	468.1 KB	0.39 secs
LX75T	26.3 MB	555.6 KB	0.40 secs
LX9	9.0 MB	215.1 KB	0.12 secs

Wire Enumerator Size and Performance

Family Name	Heap Memory Footprint	File Size on Disk	Load Time from Disk
Virtex 4	8.1 MB	234 KB	0.12 secs
Virtex 5	9.9 MB	264.4 KB	0.15 secs
Virtex 6	6.3 MB	171.3 KB	0.10 secs
Spartan 6	6.9 MB	168.0 KB	0.11 secs

PLACEMENT IN RAPIDSMITH

This chapter is intended to help users of RapidSmith understand how placement works in RapidSmith and in XDL.

Primitive Resources in RapidSmith

RapidSmith uses the XDLRC primitive definitions and sites declared to help create a map of useable places where objects may be placed. In order to understand placement in RapidSmith, let's review the following: primitive sites, primitive definitions and types, and primitive instances.

Primitive Site

A primitive site is an actual physical location in an FPGA. Let's take a look at a typical `primitive_site` declaration found in an XDLRC report for a Virtex 4:

```
(primitive_site SLICE_X1Y121 SLICEL internal 27
      (pinwire BX input BX_PINWIRE1)
      (pinwire BY input BY_PINWIRE1)
      (pinwire CE input CE_PINWIRE1)
      (pinwire CIN input CIN1)
      (pinwire CLK input CLK_PINWIRE1)
      (pinwire SR input SR_PINWIRE1)
      (pinwire F1 input F1_PINWIRE1)
      (pinwire F2 input F2_PINWIRE1)
      (pinwire F3 input F3_PINWIRE1)
      (pinwire F4 input F4_PINWIRE1)
      (pinwire G1 input G1_PINWIRE1)
      (pinwire G2 input G2_PINWIRE1)
      (pinwire G3 input G3_PINWIRE1)
      (pinwire G4 input G4_PINWIRE1)
      (pinwire FXINA input FXINA1)
      (pinwire FXINB input FXINB1)
      (pinwire F5 output F51)
      (pinwire FX output FX1)
      (pinwire X output X_PINWIRE1)
      (pinwire XB output XB_PINWIRE1)
      (pinwire XQ output XQ_PINWIRE1)
      (pinwire Y output Y_PINWIRE1)
      (pinwire YB output YB_PINWIRE1)
      (pinwire YQ output YQ_PINWIRE1)
      (pinwire COUT output COUT1)
      (pinwire YMUX output YMUX_PINWIRE1)
      (pinwire XMUX output XMUX_PINWIRE1)
)
```

This declaration is found inside of a tile declaration in XDLRC reports which denotes where the primitive site is located on the FPGA. Every primitive site belongs to a tile. If you open the Virtex 4 FX12 part in the Device Browser, you could find the primitive site SLICE_X1Y121 as shown in the screenshot below. You would find that it belongs to the tile CLB_X1Y60.

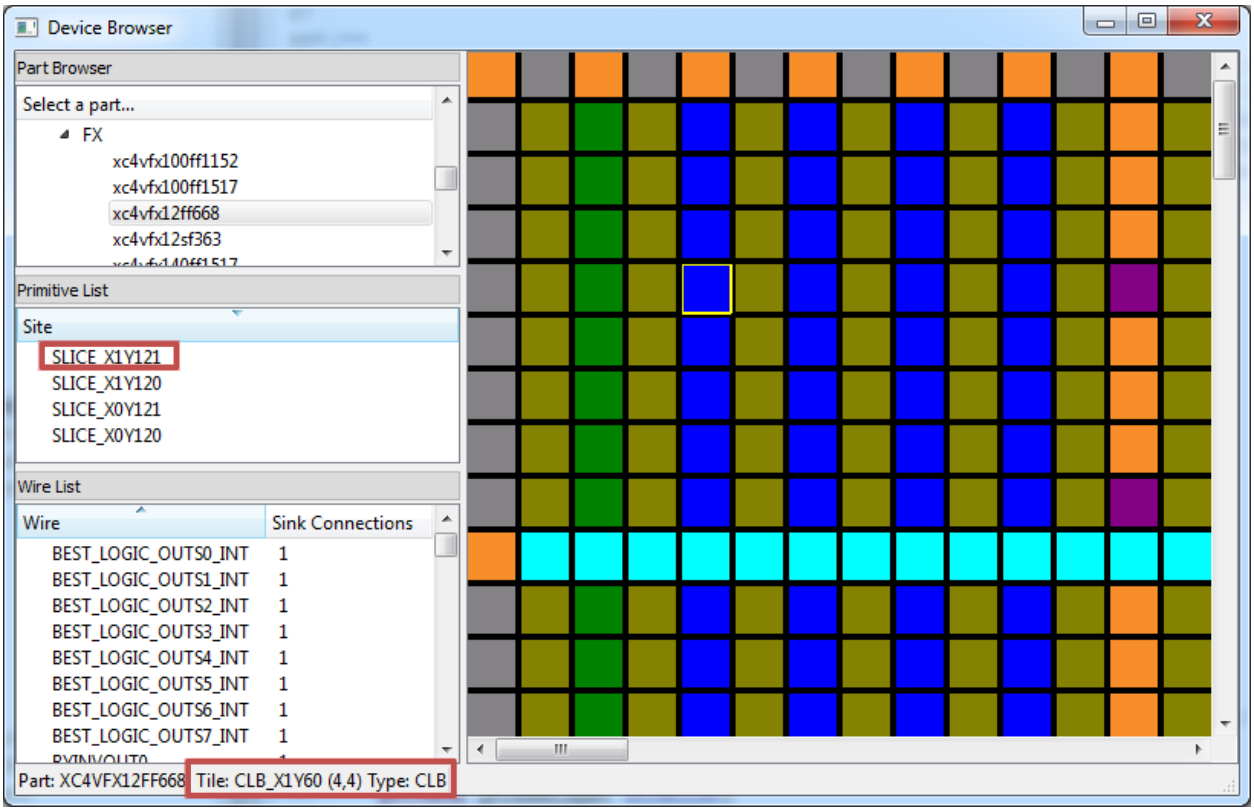

Figure 11 – Device Browser screenshot showing site SLICE_X1Y121 in tile CLB_X1Y60

Primitive sites all have a unique name (SLICE_X1Y121 in this example) to uniquely identify each one. Although the XY coordinates in the name help reference their location, they are not found in all primitive site names (IOBM/IOBS sites being a common example). Each site also has a type associated with it. RapidSmith enumerates all of the primitive types in the `device.PrimitiveType` enum.

Primitive Definitions and Types

Each Xilinx FPGA family has a list of primitive definitions which appear at the end of every XDLRC report in the `primitive_defs` declaration. A list of `primitive_def` declarations are contained in this declaration, each detailing the inputs, outputs, elements, and configurations of each primitive. Theses definitions are stored in the `primitiveDefs` package found in RapidSmith and are easily accessible using the data structures created. The names of each of the primitive definitions are called primitive types in RapidSmith. The PrimitiveType enum found in the device package contains all of the primitive types found in every Xilinx FPGA family supported in RapidSmith (see Appendix C for a complete list).

Primitive Instances

A primitive instance (or instance for short) is an actual instantiation of a primitive definition and is the same instance as declared in XDL "inst" constructs. These instances are represented in the design package using the `Instance` class. Each instance has a unique name and primitive type associated with it and can be placed or unplaced in an XDL design. When an instance is unplaced, it will have the keyword "unplaced" in its declaration. When an instance is placed, it will have the keyword placed followed by a tile name and a primitive site name (such as CLB_X1Y60 SLICE_X1Y121). An example of the first part of an instance declaration is shown below:

```
inst "instanceName" "SLICEL",placed CLB_X1Y60 SLICE_X1Y121 ,
…
```


Placement

Placement occurs by assigning an instance to a specific primitive site on the FPGA. Generally, an instance of type X can only be placed on a primitive site of type X. However, there are some exceptions where type X can actually be placed onto more than just the primitive site of the same type. A common example in modern devices is that a SLICEL instance can be placed on either a SLICEL primitive site or a SLICEM primitive site. There are actually several scenarios where this exception occurs. Sometimes, a primitive site of the same type never occurs on the FPGA fabric such as an IOB primitive type. IOB primitives must be placed on either an IOBM or IOBS primitive site.

In order to help avoid the special cases in placement with different primitive types, RapidSmith includes all of the legal placement types in the `PrimitiveSite` class and they can be accessed with the following methods:

```
/**
* This method will check if the PrimitiveType otherType can be placed
* at this primitive site.  Most often only if they are
* equal can this be true.  However there are a few special cases that require
* extra handling.  For example a SLICEL can reside in a SLICEM site but not
* vice versa.
* @param otherType The primitive type to try to place on this site.
* @return True if otherType can be placed at this primitive site, false otherwise.
*/
public boolean isCompatiblePrimitiveType(PrimitiveType otherType);

/**
 * This method gets the type of otherSite and calls the other method
 * public boolean isCompatiblePrimitiveType(PrimitiveType otherType);
 * See that method for more information.
 * @param otherSite The other site to see if its type is compatible with this site.
 * @return True if compatible, false otherwise.
 */
public boolean isCompatiblePrimitiveType(PrimitiveSite otherSite);
```

Placement Techniques

Currently RapidSmith only has a very limited random placer which is found in the placer package.

ROUTING IN RAPIDSMITH

This chapter is intended to help users and developers in understanding how routing resources are handled in RapidSmith. It also illustrates how to build on the existing classes to create custom routers.

Wire Resources in RapidSmith

RapidSmith has a unique way of representing wires and connections for Xilinx devices. This approach was developed mainly to minimize disk and memory usage while also maintaining some level of efficiency and speed.

Wire Representation

The wire enumerator class keeps a list of all uniquely XDLRC-named wires that exist in a given Xilinx FPGA family. Wires can span multiple tiles in the FPGA, however, the wire has a separate name for each tile in which it crosses. An example of this concept is illustrated in the DOUBLE lines found in several family architectures. A DOUBLE line is a wire that connects switch boxes either one or two hops away in a given direction. An example of this layout is given in Figure 7.

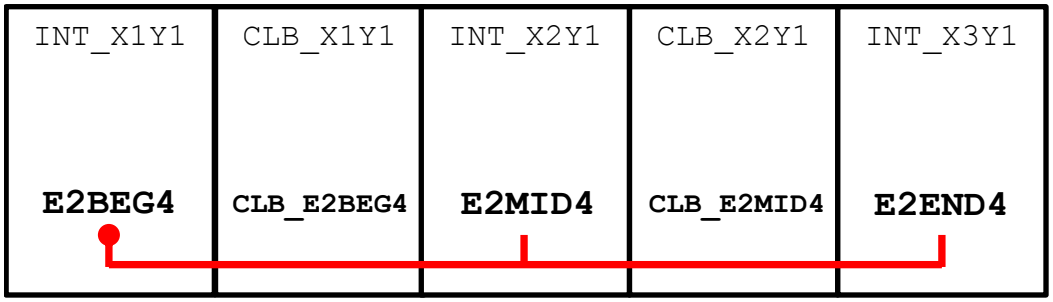

Figure 12 - A DOUBLE line in an FPGA illustrating how each part of the wire has a different name depending on the tile it is located in.

In this example, we see a wire that can be driven by one point, `E2BEG4`, and can drive either `E2MID4` in tile `INT_X2Y1` and/or `E2END4` in tile `INT_X3Y1`. However, the wire is assigned a name as it travels through the CLB tiles (`CLB_E2BEG4` and `CLB_E2MID4`). For the purposes of RapidSmith, these wires have been removed from device files as they do not contribute to the overall possible connections a wire can make and simply add overhead to the device data structures. This technique has actually dramatically reduced the size of the devices files and improved routing speed as dead-end connections do not need to be examined.

In RapidSmith, these uniquely-named wire segments are represented either as a `String` or as an `int` or `Integer`. Often it is represented as an integer to save space and increase comparison speed with other wires. To illustrate how this representation works, here is some example Java code that exposes the wire segments:

```
// Load the appropriate Device and WireEnumerator
// (this is done automatically when loading XDL designs)
String partName = "xc4vfx12ff668";
Device dev = FileTools.loadDevice(partName);
WireEnumerator we = FileTools.loadWireEnumerator(partName);
// Here we pick a wire name
String wireName = "E2BEG4";
// Here we get the integer enum value for that wire name
int wire = we.getWireEnum(wireName);
// The wire enumerator also keeps information about these wire segments
```



```
            // such as wire direction and type
            WireDirection direction = we.getWireDirection(wire);
            WireType type = we.getWireType(wire);
```

Now, there are actually several wires in an FPGA device with the same name. The wire `E2BEG4` exists in almost every switch box tile in the FPGA. To uniquely identify routing resources in a device, a tile and its name or wire enumeration is required (that is, `INT_X1Y1 E2BEG4` is its unique representation).

In an effort to save space and ultimately reuse much of the routing connections, the `Wire` class is used to represent internal and external tile connections. Each tile has a hash map where the key is the integer enum value of the wire and the value is an array of `Wire` objects. Each `Wire` object contains the following information to define a connection:

```
      /** The wire enumeration value of the wire to be connected to */
      private int wire;
      /** The tile row offset from the source wire's tile */
      private int rowOffset;
      /** The tile column offset from the source wire's tile */
      private int columnOffset;
      /** Does the source wire connected to this wire make a PIP? */
      private boolean isPIP;
```

The `Wire` objects can define the connecting wire by using the integer enumeration value of the wire name and a relative offset of the tile differences between the two wires (again, relative to save space and increase reuse of the object). The `Wire` object also defines if the connection made is a programmable connection (or PIP). When the row and column tile offsets are both 0, the connection exists within the same tile and is likely a PIP.

To query the connections that can be made from `INT_X1Y1 E2BEG4`, here is some sample Java code to illustrate how this is done:

```
// Load the appropriate Device and WireEnumerator
// (this is done automatically when loading XDL designs)
String partName = "xc4vfx12ff668";
Device dev = FileTools.loadDevice(partName);
WireEnumerator we = FileTools.loadWireEnumerator(partName);
// Here we pick a wire name
String wireName = "E2BEG4";
// Here we get the integer enum value for that wire name
int wire = we.getWireEnum(wireName);
String tileName = "INT_X1Y1";
Tile tile = dev.getTile(tileName);
Wire[] wireConnections = tile.getWireConnections(wire);
for(Wire w : wireConnections){
      System.out.println(tileName + " " +
                         wireName + " connects to " +
                         dev.getTile(tile.getRow()-w.getRowOffset(),
                                     tile.getColumn()-w.getColumnOffset()) + " " +
                         we.getWireName(w.getWire()) + " (is" +
                         (w.isPIP()? " " : " not ") +
                         "a PIP connection)");
}
```

Console Output:

```
INT_X1Y1 E2BEG4 connects to INT_X1Y1 BOUNCE1 (is a PIP connection)
INT_X1Y1 E2BEG4 connects to INT_X1Y1 BOUNCE2 (is a PIP connection)
```



```
INT_X1Y1 E2BEG4 connects to INT_X3Y1 E2END4 (is not a PIP connection)
INT_X1Y1 E2BEG4 connects to INT_X2Y1 E2MID4 (is not a PIP connection)
```

Routes in XDL are specified only with PIPs. Non-PIP connections (that is `E2BEG4` to `E2MID4`, etc.) are not declared in an XDL Net since the connection is implied. The two wire segments are part of the same piece of metal on the FPGA. Thus, when declaring the routing resources used in a Net (the list of PIPs), these connections are not explicitly listed. However, the PIP connections are, for example:

```
net "main_00/i_ila/i_dt0/1/data_dly1_20" ,
  outpin "main_00/i_ila/i_dt0/1/data_dly1_20" XQ ,
  inpin "main_00/i_ila/i_yes_d/u_ila/idata_70" BY ,
  pip CLB_X16Y48 XQ_PINWIRE2 -> SECONDARY_LOGIC_OUTS2_INT ,
  pip CLB_X18Y48 BYP_INT_B4_INT -> BY_PINWIRE0 ,
  pip INT_X16Y48 SECONDARY_LOGIC_OUTS2 -> OMUX7 ,
  pip INT_X17Y48 OMUX_E7 -> E2BEG4 ,
  pip INT_X18Y48 E2MID4 -> BYP_INT_B4 ,
  ;
```

The listing of PIPs in XDL is arbitrary, that is, they do not always follow from one connection to the next.

Basic Routing

RapidSmith has included an `AbstractRouter` class that allows for a common template so that routers can be constructed quite easily. However, the user should not feel restricted in using this template as it may not meet everyone's needs and/or requirements.

An example `BasicRouter` class has also been provided to illustrate how a router can be constructed easily. The `BasicRouter` class is ~400 lines of code. It is very simple and does not do any routing conflict resolution (it is a basic Maze router implementation) and it will commonly be unable to route certain connections in a design. Also, because the timing information for Xilinx parts is not publicly available, the router must use other means to optimize the router rather than delay. However, it does perform re-entrant routing, that is, it will attempt to route all nets that don't have any PIPs while keeping the original routed nets intact. If a net is impartially routed or improperly routed before given to the router, it does not resolve these problems. The behavior and mechanics of this router are described in the remainder of this section.

Router Structure

The basic router provided in RapidSmith is based on a simple maze router algorithm. It does not allow routing resources to be used more than once, and thus, routing resources come on a first-come-first-served basis. This makes for a very simple implementation but does not resolve routing conflicts when they arise. The router chooses a route by iterating through a growing set of nodes, represented by the Node class. A node is a unique tile and wire combination to uniquely identify any routing wire available in the FPGA. Nodes are given a cost based on their Manhattan distance from the sink of the current connection to be routed and then placed in a priority queue. Those nodes with the smallest cost propagate to the bottom of the queue.

The least cost node of the queue is iteratively removed. With each removal, the node is examined for its expanding connections and those new potential nodes are also placed on the queue. Each time a node is removed, it is tested to see if it is the sink, if it is, the method traverses the path it has found and returns, otherwise it continues to expand more connections of the current node.

The router uses the following basic algorithm:

1. The central routing method, `routeDesign()` prepares the nets in the design for routing.

2. For each net in the design, `routeDesign()` will call `routeNet()`.
    a. `routeNet()` prepares each inpin or sink in the net for routing.
        i. If this is the first inpin of the net, it will only supply the outpin or source of the net as a starting point to the router.
        ii. If this is the second or later inpin routed in the net, all intermediate points along those routes are added as starting points.
    b. For each inpin, `routeNet()` will call `routeConnection()`.
        i. `routeConnection()` initializes the priority queue of potential source nodes.
        ii. `routeConnection()` calls the main routing method `route()` for each connection to be routed.
            1. The `route()` method iterates over the nodes in the priority queue, expanding their connections and adding new ones to the queue and putting more connections on the queue. The process continues until the sink is found.
3. After a net has been routed, the routing resources used will be marked as used to avoid reusing the resources twice.

Routing Static Sources (VCC/GND)

One major preparation step in routing a full design is preparing where the static sources will be supplied from. The basic primitive in all Xilinx FPGAs to supply VCC and GND signals to a design is the TIEOFF. The TIEOFF accompanies every switch matrix and has several connections to all sink connections to its neighboring logic tile (CLB, BRAM, DSP, etc.). It has 3 pins, HARD0 or GND, KEEP1 (VCC) and HARD1 (VCC). By default, without any configuration, it seems that pins will default to KEEP1. Some pins, however, require a HARD1 when specified to be driven with VCC.

The `StaticSourceHandler` class takes care of partitioning the various nets and sinks into their respective tiles and instancing the TIEOFF automatically. It also will instance SLICEs when necessary. It also "reserves" certain routing resources for certain nets that could potentially introduce routing conflicts later. These reserved nodes are released just before the net is routed in the basic router.

Routing Clocks

When routing clocks, it is quite important that they get routed to the appropriate clock tree routing resources. The best current method to determine this is based on the `WireDirection` (the type CLK was placed in `WireDirection` because there are certain CLK wires that also fell into certain `WireType` categories). The cost function for determining node position in the priority queue take into account clock wires and significantly reduces their cost when routing clock nets.

Internal Pin Names and External Pin Names

In RapidSmith, there is the notion of each pin on an instance having an internal name and an external name. This can easily get confusing, especially where this can be a weak point for XDLRC report files which lack some of this information for some primitive types.

Internal pin names occur commonly in two places (although, they do occur in other places):

1. In XDL nets which contain "outpin" and "inpin" statements
2. In XDLRC primitive_def declarations in the primitive_defs section of an XDLRC report.

First, let's talk about pins found in nets in XDL designs. In XDL, pins in a net are declared first with either the keyword "outpin" (to designate the source) or "inpin" (to designate a sink). Following the keyword is the name of the instance the pin belongs to. To illustrate this, let's look at an example:


```
net "netName" ,
  outpin "fred" Y ,
  inpin "barney" RST ,
  pip CLB_X14Y4 Y_PINWIRE1 -> BEST_LOGIC_OUTS5_INT ,
  pip DCM_BOT_X15Y4 SR_B0_INT3 -> DCM_ADV_RST ,
  pip INT_X14Y4 BEST_LOGIC_OUTS5 -> OMUX8 ,
  pip INT_X15Y5 OMUX_EN8 -> N2BEG0 ,
  pip INT_X15Y7 N2END0 -> SR_B0 ,
  ;
```

In the example above, there are two pins, a source and a sink. The source is found on the instance "fred" and the sink is found on the instance "barney." The source pin on "fred" is pin "Y" and the sink pin is "RST" on "barney."

However, a problem arises when trying to use pin names in routing. For example if the pin name Y were used to specify routing to the instance it would be ambiguous because the Y pin belongs to a slice. Since PIPs declare routing resources at the tile level, the pin Y would have to be unique to the tile, however, there are actually multiple slices in a CLB tile making the reference "Y" ambiguous. To eliminate the ambiguity, Xilinx developed what we call an internal pin name and external pin name. The internal pin name (Y and RST in the example) is used when talking about a pin on an instance, however, to route to/from that pin the external name is used. In the PIP list of the example net above, the first PIP contains the external name "Y_PINWIRE1" of pin Y on "fred" and the second PIP contains that external name "DCM_ADV_RST" of the pin RST. In the Virtex 4 architecture, there are 4 slices in each CLB, so the Y pin on each slice is named Y_PINWIRE0, Y_PINWIRE1, Y_PINWIRE2 and Y_PINWIRE3 respectively. The mapping of an internal pin name to an external pin name is found in the primitive_site declaration in an XDLRC report.

Let's look at an example of an XDLRC primitive_site:

```
            (primitive_site SLICE_X1Y126 SLICEL internal 27
                  (pinwire BX input BX_PINWIRE1)
                  (pinwire BY input BY_PINWIRE1)
                  (pinwire CE input CE_PINWIRE1)
                  (pinwire CIN input CIN1)
                  (pinwire CLK input CLK_PINWIRE1)
                  (pinwire SR input SR_PINWIRE1)
                  (pinwire F1 input F1_PINWIRE1)
                  (pinwire F2 input F2_PINWIRE1)
                  (pinwire F3 input F3_PINWIRE1)
                  (pinwire F4 input F4_PINWIRE1)
                  (pinwire G1 input G1_PINWIRE1)
                  (pinwire G2 input G2_PINWIRE1)
                  (pinwire G3 input G3_PINWIRE1)
                  (pinwire G4 input G4_PINWIRE1)
                  (pinwire FXINA input FXINA1)
                  (pinwire FXINB input FXINB1)
                  (pinwire F5 output F51)
                  (pinwire FX output FX1)
                  (pinwire X output X_PINWIRE1)
                  (pinwire XB output XB_PINWIRE1)
                  (pinwire XQ output XQ_PINWIRE1)
                  (pinwire Y output Y_PINWIRE1)
                  (pinwire YB output YB_PINWIRE1)
                  (pinwire YQ output YQ_PINWIRE1)
                  (pinwire COUT output COUT1)
                  (pinwire YMUX output YMUX_PINWIRE1)
```



```
                (pinwire XMUX output XMUX_PINWIRE1)
        )
```

XDLRC report files show a mapping of internal pin name to external pin name on each line which starts with "(pinwire". The pattern is:

"(pinwire <internal pin name> <direction of pin> <external pin name>)"

This is very straight forward and is the second common location to find internal and external pin names in XDL/XDLRC. In RapidSmith, the mapping between internal and external pin names can be made using the following methods:

In the PrimitiveSite class:

```
/**
 * Gets the external wire enumeration of the name of the wire corresponding to the
 * internal wire name.
 * @param internalName The internal wire name in the primitive.
 * @return The corresponding external wire enum (Integer) name of the internal wire
 * name.
 */
public Integer getExternalPinName(String internalName);
```

In the Device class:

```
/**
 * Gets the external wire enumeration on the instance pin.
 * @param pin The pin to get the external name from.
 * @return The wire enumeration of the internal pin on the instance primitive of
 * pin.
 */
public Integer getPrimitiveExternalPin(Pin pin);
```

There is a problem, however, with some primitive types and getting mappings for their internal pin names to external pin names. Sometimes, a primitive type does not have a native primitive site in any device in Xilinx FPGA family. Therefore, the primitive must be placed on a compatible primitive site of a different type. For example, an IOB primitive instance does not have a native site on most families. However, it is fully compatible with IOBM or IOBS sites.

In certain instances, the internal pin names differ on the primitive with no native sites to the sites on which it can be placed. The biggest example of this is in the Virtex 5 which has 9 different primitive types which all use the same primitive site type (RAMBFIFO36). Because the RAMBFIFO36 site is declared several times in the Virtex 5 devices, all of the internal-to-external pin mappings are available. However, 8 other sets of mappings are not present. The lack of mappings makes routing designs which contain these primitives impossible. The solution to this problem is to apply a patch with the proper mappings. A complete patch will be included with RapidSmith in a future release.

BITSTREAMS IN RAPIDSMITH

In 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 provides only documented functionality when working with bitstreams. This functionality comes mainly from the documents distributed by Xilinx in the form of user guides, whitepapers and application notes. Any discussion relating to bitstreams in this document (unless explicitly clarified) refers to Virtex 4/5/6 bitstreams.

Bitstreams are stored in 32-bit words for ease of interpretation. The best reference for much of the details and specifics of the bitstream can be found in the Configuration User's Guide for each of the FPGA family architectures. Here are links to the references:

- Xilinx UG071 – Virtex 4 FPGA Configuration User Guide
- Xilinx UG191 – Virtex 5 FPGA Configuration User Guide
- Xilinx UG360 – Virtex 6 FPGA Configuration User Guide

Although each guide contains valuable information for each FPGA architecture family, the Virtex 5 guide is the most comprehensive and complete. It will provide the most insights and most of those insights can be applied to the other architecture families.

RapidSmith has a bitstream parser which is capable of importing bitstream files into the bitstream data structure. This data structure can also export bitstreams after modification to a bitstream file and/or an MCS PROM file. For more details on the bitstream import and export capabilities see the `bitstreamTools.bitstream.BitstreamParser` class and `bitstreamTools.bitstream.Bitstream` class.

Bitstream Composition

In order to effectively use the bitstream capabilities provided in RapidSmith a preliminary understanding of how Xilinx bitstream are organized is necessary. In RapidSmith, Xilinx bitstream files are composed of three main elements as encapsulated by the `bitstreamTools.bitstream.Bitstream` class:

1. A bitstream header (`bitstreamTools.bitstream.BitstreamHeader`)
2. Dummy and Synchronization data (`bitstreamTools.bitstream.DummySyncData`)
3. A list of bitstream packets (`bitstreamTools.bitstream.PacketList`)

Figure 13 below provides an illustrative diagram of how all of the components of a bitstream file correlate with the `bitstreamTools.bitstream` package in RapidSmith.

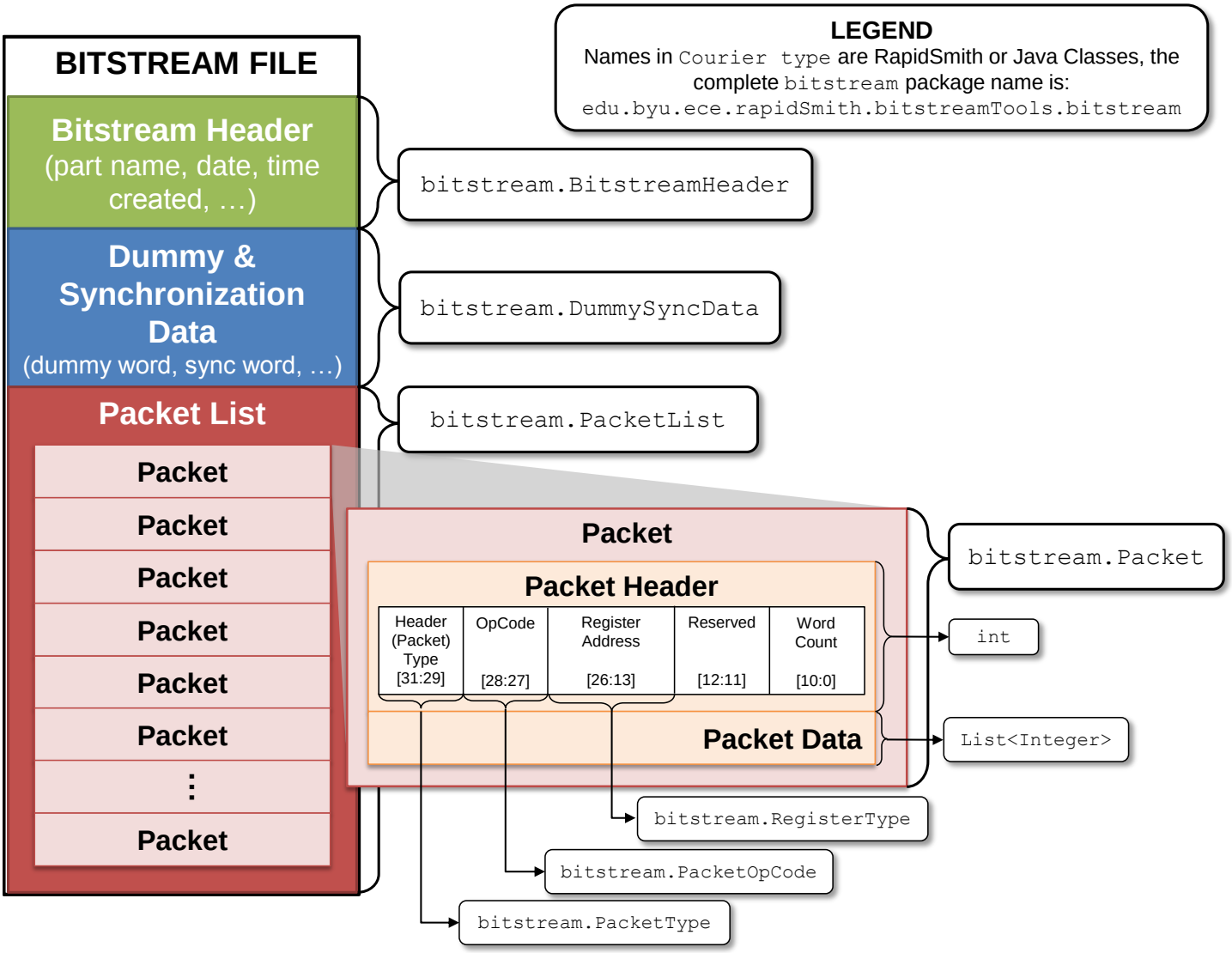

Figure 13 - This diagram represents how RapidSmith encapsulates bitstream files using the bitstream package.

The entire package `bitstreamTools.bitstream.*` is dedicated to the representation of bitstream files. A bitstream file contains everything the configuration logic needs to setup the FPGA in preparation for configuration as well as containing all of the configuration data itself.

Bitstream Header

The first set of data that appears in a bitstream file is the bitstream header. The bitstream header contains information about the bitstream such as:

- The name of the NCD file the bitstream from which it was created
- The FPGA part name the bitstream targets
- The date the bitstream was created
- The time the bitstream was created

All bitstream header information is actually ignored by the configuration logic and theoretically could be very large if a custom header was created. The `bitstreamTools.bitstream.BitstreamHeader` class is designed to parse headers and allow users to read and write the header and also export it to an XML format.

Dummy and Synchronization Data

In order for the configuration process to begin, the configuration controller must lock onto the sequence of words it is receiving from JTAG. It does this by waiting for a dummy word followed by a sync word (0xFFFFFFFF and 0xAA995566) in order to lock onto the incoming data. The `bitstreamTools.bitstream.DummySyncData` class is responsible for representing all dummy and synchronization data that is found immediately after the header but before the packet list as this data can be different from family to family.

Packet List

Packets are like configuration controller instructions that read or write a particular register. Once the configuration process has been synchronized, the configuration controller will begin to read the packets in the bitstream and execute their read or register write. Much of the details concerning packets can be found in the configuration user guides referenced at the beginning of this section, however, Figure 13 shows how a packet header is described in RapidSmith.

One of the reliability mechanisms in bitstreams are CRC checks. Bitstreams include special packets which contain a CRC checksum based on the previous data in the bitstream file. This CRC ensures that the bitstream arrives correctly when parsed by the configuration logic. RapidSmith has facilities to create and/or update CRC packets when data changes. See the `bitstreamTools.bitstream.CRC` and `bitstreamTools.bitstream.PacketListCRC` classes for more details.

There are two types of packets, Type I and Type II as indicated in the most significant bits of the packet header. Type I packets are used for small amounts of data being written to configuration registers. Type II packets always must follow a Type I packet and are able to carry a much bigger payload such as the entire configuration data for an FPGA. Generally, a normal bitstream will only have one Type II packet which is the packet carrying all of the configuration data for the FPGA. The configuration data can be sent in Type I packets, but a Type II packet is used for efficiency in programming and speed. This configuration data has its own addressing scheme and thus, the bitstream tools in RapidSmith accommodate this with a few other packages.

Bitstream Configuration Data

FPGA

The large Type II packet present in most bitstreams will likely contain all of the configuration data for the FPGA fabric. The bitstream tools in RapidSmith are able to emulate the configuring (at an abstract bitstream level) of the FPGA fabric in the `bitstreamTools.configuration.FPGA` class. In essence, the bitstream in the `bitstreamTools.bitstream.Bitstream` class "configures" or gets applied to the `FPGA` class.

Xilinx Configuration Specification

Because each FPGA is a little different and can vary among FPGA family architectures, RapidSmith has special classes that contain part specific information automatically generated from debug bitstreams to enable the `FPGA` class to emulate any device in the Virtex 4/5/6 families. In order to accommodate any device in the Virtex 4/5/6 families, the FPGA class requires what is called a XilinxConfigurationSpecification which defines certain bitstream attributes about a particular device. The XilinxConfigurationSpecification is a public Java interface declared in `bitstreamTools.configurationSpecification.XilinxConfigurationSpecification`. This interface is implemented by an abstract class

(`bitstreamTools.configurationSpecification.AbstractConfigurationSpecification`) and further extended by other classes, where each class contains information for each family. Specific parts are represented in PartLibrary classes which extend the family specific {V4,V5,V6}ConfigurationSpecifications.

Much of the headache of trying to understand the previous paragraph is avoided by simply doing the following to initialize an FPGA instance:

```
String partName = "xc4vfx12ff668-10";
XilinxConfigurationSpecification spec = DeviceLookup.lookupPartV4V5V6(partName);
FPGA virtex4fx12 = new FPGA(spec);
```

Frame Address Register

An organized mechanism is used to access various parts of the configuration data in an FPGA. This mechanism is called the frame address register or FAR for short, it is represented in RapidSmith as the `bitstreamTools.configuration.FrameAddressRegister` class. The FAR is one of the configuration registers found in the configuration logic. Like all configuration registers, the FAR is 32 bits and although the bits are laid out in a different order for different architectures, they all contain the same fields:

- **Top/Bottom**: A bit indicating if the referenced bits are in the top half rows or bottom half rows of the device.
- **Block Type**: The type of block (configuration block) that is being referenced by the register. The block types vary depending on the architecture—See configuration guides for more details.
- **Row Address**: References a row of frames. Row addresses start at 0 in the middle of the chip and increment towards the top or bottom.
- **Column Address**: Selects a major column (such as CLBs) to reference. Column addresses start at 0 at the left side of the chip and increment towards the right.
- **Minor Address**: Selects a frame inside of a configuration block. Different configuration blocks have different sizes of frames.

The bit field assignments for Virtex 4/5/6 are shown in the table below:

FAR Address Fields	Virtex 4	Virtex 5	Virtex 6
Block Type	[21:19]	[23:21]	[23:21]
Top / Bottom Bit	22	20	20
Row Address	[18:14]	[19:15]	[19:15]
Column Address	[13:6]	[14:7]	[14:7]
Minor Address	[5:0]	[6:0]	[6:0]

Frame

A frame is the smallest addressable configuration data in a Xilinx FPGA. In Virtex 4 and Virtex 5 architectures, a frame is 1312 bits or 41 32-bit words. In Virtex 6, a frame is 2592 bits or 81 32-bit words. The frame is addressed using the FAR and populating the appropriate fields. Each frame will have a unique address which corresponds to a different value in the FAR or what is called a *frame address*.

A frame in RapidSmith is represented by the `bitstreamTools.configuration.Frame` class. A frame has a unique frame address (represented by a 32-bit `int`) and contains 41/81 words of data which are stored in the `bitstreamTools.configuration.FrameData` class. Frame data is organized fairly similarly

among architectures. As an example of how frame data is organized, consider the figure below taken from the *Xilinx UG191(v3.9.1) - Virtex 5 Configuration Guide* (Figure 6-9, p. 130):

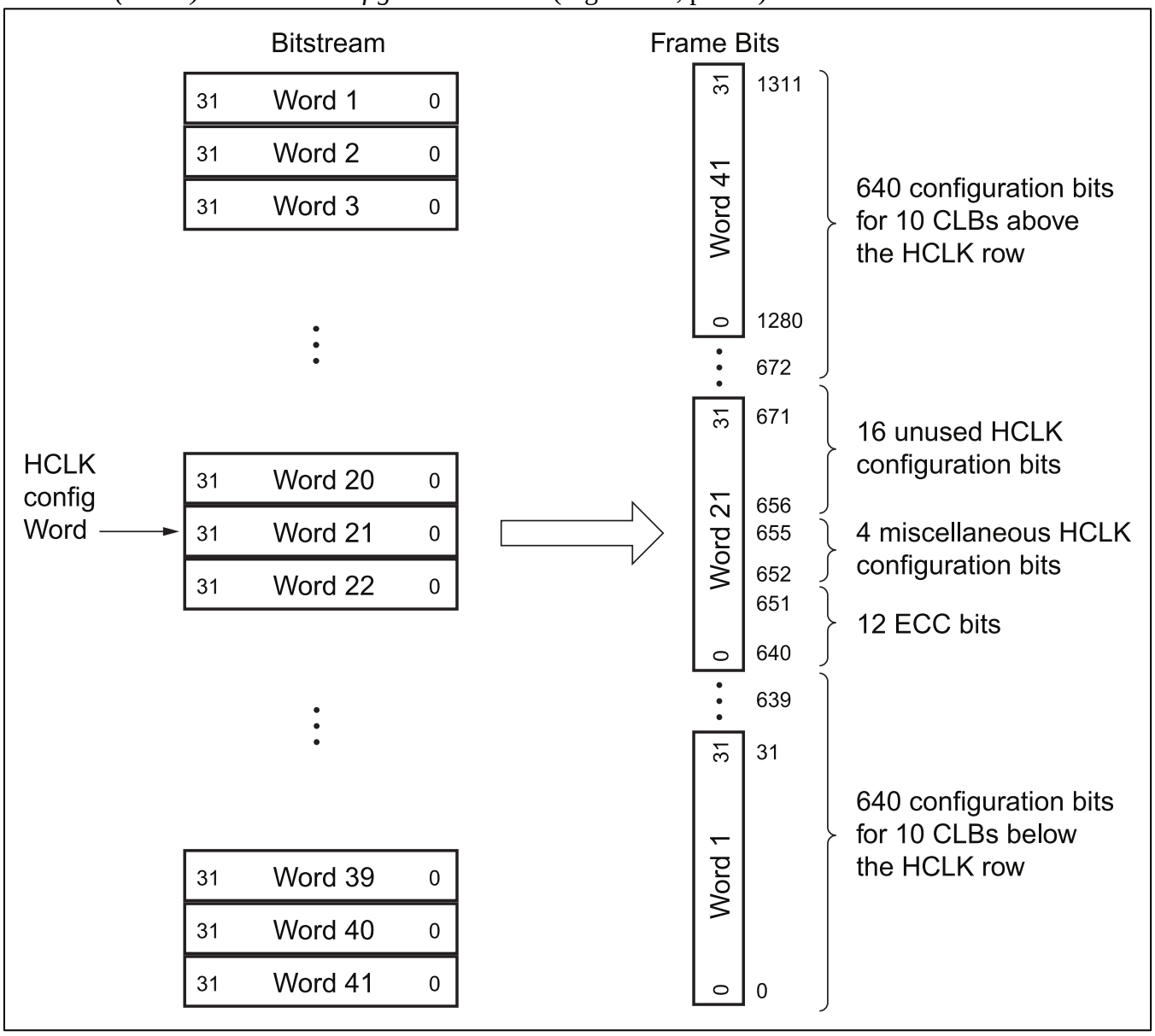

Figure 14 - Layout of the data in a Virtex 5 configuration frame.

As can be seen from Figure 14, the center configuration word contains configuration bits the control the horizontal clock wires and contains ECC bits for the frame data. The words before the center word configure the top half of a configuration block (or in the case of Virtex 5, the top 10 CLBs) and the words after the center word configure the bottom half of the configuration block.

Configuration Block

Frames are grouped into groups called configuration blocks. Although there is no explicit class to represent a configuration block in RapidSmith, it is heavily used concept. A configuration block can be defined as the grouping of frames that have the same top/bottom-bit, row address and column address in a frame's address (also as indicated in the frame address register). Configuration blocks come in different types, and each type has a subtype. Block types and subtypes are represented in the `bitstreamTools.configurationSpecification.BlockType` and `bitstreamTools.configurationSpecification.BlockSubType` classes. Block types and

subtypes are specified in the architecture's corresponding configuration specification. To give a quick overview of some block types and their frame sizes, consider the table below:

Configuration Block Type/SubType	Virtex 4	Virtex 5	Virtex 6
CLB Block	22 frames	36 frames	36 frames
IO Block	30 frames	54 frames	44 frames
DSP Block	21 frames	28 frames	28 frames
CLK Block	3 frames	4 frames	38 frames
MGT/GTX Block	20 frames	32 frames	30 frames
BRAM Interconnect Block	20 frames	30 frames	28 frames
BRAM Content Block	64 frames	128 frames	128 frames
Overhead	2 frames	2 frames	2 frames
BRAM Overhead	2 frames	2 frames	2 frames

Figure 15 - Frame counts for different block types and block subtypes in Virtex 4/5/6 architectures.

APPENDIX

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

Appendix A: Modifying LUT Content

LUTs (look up tables) found in Xilinx slices can be easily modified using RapidSmith. LUT content is stored in an attribute in an instance of a SLICEL or SLICEM or whatever type of SLICE the device has. Often the name of the LUT is a single letter such as F or G as in the Virtex 4 family. Virtex 5 FPGAs have 4 LUTs in a slice and are called "{A, B, C, D}{5, 6}LUT" which have the capability to act as a 5 input or 6 input LUT.

LUT Equation Syntax

Xilinx uses the following syntax for operators in a LUT equations string:

Operator	Operator Meaning
*	Logical AND
+	Logical OR
@	Logical XOR
~	Unary NOT

The parenthesis characters are also used to denote precedence in equation. Valid equation values are A1, A2, A3 and A4 for 4-input LUTs with an additional A5 for 5-input LUTs and A6 for 6-input LUTs. Some examples of LUT equations are:

LUT Equation	Equation Meaning
A1	Passes through the signal A1
A1*A2	A 2-input AND gate using A1 and A2
~A4	Inverts the signal on A4
(A4+(A1+(A2+A3)))	A multi-level OR of 4 inputs

XDL LUT Equation Syntax

An instance is configured by zero or more attributes in a list where an attribute has the pattern

```
<Physical Name>:<Optional Logical Name>:<Value>.
```

The physical name of a LUT has been mentioned at the beginning of this subsection as being F or G for Virtex 4 parts and "{A, B, C, D}{5, 6}LUT" for Virtex 5 parts. The optional logical name is generally chosen by the synthesis tools or whatever name propagated through to Xilinx NGDBuild and Map when the signal was converted. It is only for reference back to the user's original design and should correspond with the output of the LUT. The value has the following syntax:

```
#LUT:D=<equation>
```

This syntax can also be found in the XDLRC primitive_defs. If we put this altogether with our examples above we would get:

```
F:mySignal0:#LUT:D=A1
G:mySignal1:#LUT:D=A1*A2
```



```
F:mySignal2:#LUT:D=~A4
G:mySignal3:#LUT:D=(A4+(A1+(A2+A3)))
```

Notice that the value also contains the colon (‘:’). RapidSmith defines the separation of the three components of an attribute as the first and second colons, any colons found after the second colon is part of the attribute value.

It is also of some value to recognize that the router can re-arrange LUT inputs to make routing easier and this can change equation to some extent.

Appendix B: Hard Macros in XDL and RapidSmith

A hard macro is a collection of primitive instances and nets that have been placed and possibly routed. The XDL construct that represents a hard macro is the “module”. A hard macro file only has a single module and no other nets, instances or module instances present in the file. The design name is "__XILINX_NMC_MACRO" which is a static variable in the Design class:

```
public static final String hardMacroDesignName = "__XILINX_NMC_MACRO";
```

Xilinx NMC files

The NCD file format is not used for hard macros; instead, Xilinx uses the NMC format which is fairly similar. NMC files can be converted to and from XDL using the same `xdl` executable. For example:

```
xdl –ncd2ndl myHardMacro.nmc [optional: myHardMacro.xdl]
xdl –xdl2ncd myHardMacro.xdl myHardMacro.nmc
```

The user must specify the output file name for NMC files when converting from XDL to NMC, otherwise it will output a file with the same name but NCD extension and the Xilinx tools won’t read it properly. You may also need to apply the `–force` option if there is an error.

Xilinx Hard Macros

There are several hidden restrictions and conventions that must be adhered to in order to create a valid Xilinx hard macro. Here is a list of the known quirks to creating Xilinx-compatible hard macros:

- Any nets that are designated as GND or VCC are invalid (this will cause `xdl` to run in an infinite loop)
- The TIEOFF primitive found in INT tiles (switch matrix tiles) cannot be used in hard macros (this will cause a problem later on with `map` and/or `par`)
- Conventional “XDL_DUMMY” SLICEs which contain a “_NO_USER_LOGIC” attribute will not work with the Xilinx tools.
- Nets can only have 1 (one) hard macro port assigned to its pin list.

RapidSmith Hard Macro Generator

RapidSmith provides a generic hard macro generator that will take a placed and routed NCD design file (or XDL equivalent) and replace IOBs with module ports among other transformations to make it a valid and legal hard macro. The hard macro generator provided is the same as used in the published paper, “Using Hard Macros to Reduce FPGA Compilation Time”, C. Lavin, et al., FPL’2010. This class is found in the `util` package and can be run from the command line:

```
java edu.byu.ece.rapidSmith.util.HardMacroGenerator <input.xdl |
input.ncd> <output file type: xdl | nmc> [optional: original_toplevel.vhd]
```


The hard macro generator has to remove TIEOFFs (which supply GND and VCC) because they are not legal primitives in hard macros. Therefore, the hard macro generator creates input ports accordingly. It also renames ports according to primitive instance names of the IOBs and will attempt to name them as they were found in the original HDL. However, it only can guess, and for a more complete naming, the top level VHDL file can optionally be provided as a parameter to help the generator get the port names correctly. Unfortunately, the generator cannot parse Verilog.

Appendix C: Xilinx Family Names and Part Names

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

Xilinx Part Names in RapidSmith

RapidSmith uses the part name pattern as produced by the Xilinx 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 RapidSmith 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	• Virtex 4 LX30
• XC5LX110TFF1136-2	• XC5VSX35T-2FF665C
• XCV50BG256	• XC5VLX20T

RapidSmith contains methods in `util.RunXilinxTools` to automatically run `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. The user can also refer to the final table in the Appendix entry on Memory and Performance of RapidSmith for a long list of part names. Other manipulation and conversion function for part names are found in the `util.PartNameTools` class.

Xilinx Family Names in RapidSmith

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);
```


Appendix D: XDLRC Compatible Families

RapidSmith depends on Xilinx XDLRC files for device descriptions. Xilinx offers several different families which in actuality use the same XDLRC information. Therefore, there is a set of base (commercial grade) families that are compatible with other families as well. The follow is a table showing these compatibilities:

Base Families (Formal Name)	Base Families (`partgen` Name)	XDLRC Compatible Families
Spartan2	spartan2	-
Spartan2E	spartan2e	aspartan2e
Spartan3	spartan3	aspartan3
Spartan3A and Spartan3AN	spartan3a	aspartan3a
Spartan-3A DSP	spartan3adsp	aspartan3adsp
Spartan3E	spartan3e	aspartan3e
Spartan6	spartan6	spartan6l, aspartan6, qspartan6, qspartan6l
Virtex	virtex	qvirtex, qrvirtex
Virtex2	virtex2	qvirtex2, qrvirtex2
Virtex2P	virtex2p	qvirtex2p
Virtex4	virtex4	qrvirtex4, qvirtex4
Virtex5	virtex5	qvirtex5
Virtex6	virtex6	virtex6l
VirtexE	virtexe	qvirtexe

Only device files are generated for the base families. Therefore, when loading designs or RapidSmith files, it's always best to use the base family name. One way to ensure this is to use the following method:

```
/**
 * This method will take a familyType and return the base familyType
 * architecture.  For example, the XDLRC RapidSmith uses for Automotive
 * Spartan 6, Low Power Spartan 6 and Military Grade Spartan 6 all have
 * the same base architecture: Spartan 6.  This method determines the
 * base architecture based on the familyType.
 * @param type The given family type.
 * @return The base family type architecture.
 */
public static FamilyType getBaseTypeFromFamilyType(FamilyType type);
```

Appendix E: Memory and Performance of RapidSmith

This section contains a dump from the shell script generated in the `tests.TestFileLoading` class in RapidSmith. It determines the files size, Java heap usage in MB and load time from disk for the following 3 types of files. This script was run on a Windows 7 Enterprise 64-bit OS with an Intel Core i7 860 running at 2.8GHz. The test machine had 8 GB of DDR3 memory and was using version 0.3.0 of RapidSmith and the 32-bit Oracle JVM version 1.6.0_22. Running RapidSmith in 64-bit Java will work exactly the same as in 32-bit mode; however its performance will be slightly slower and use more memory.

Wire Enumerator Files

Family Name	File Size	Java Heap Usage	Load Time From Disk
SPARTAN2	17KB	0.6MB	0.030s

SPARTAN2E	19KB	0.6MB	0.030s
SPARTAN3	15KB	0.5MB	0.030s
SPARTAN3A	25KB	0.9MB	0.032s
SPARTAN3ADSP	33KB	1.2MB	0.035s
SPARTAN3E	26KB	0.9MB	0.033s
SPARTAN6	168KB	6.6MB	0.081s
VIRTEX	17KB	0.6MB	0.030s
VIRTEX2	21KB	0.7MB	0.033s
VIRTEX2P	90KB	3.1MB	0.055s
VIRTEX4	233KB	8.1MB	0.109s
VIRTEX5	264KB	9.9MB	0.128s
VIRTEX6	171KB	6.3MB	0.098s

Primitive Definitions Files

Family Name	File Size	Java Heap Usage	Load Time From Disk
SPARTAN2	8KB	0.1MB	0.085s
SPARTAN2E	9KB	0.1MB	0.084s
SPARTAN3	17KB	0.1MB	0.082s
SPARTAN3A	23KB	0.1MB	0.097s
SPARTAN3ADSP	63KB	0.1MB	0.110s
SPARTAN3E	21KB	0.1MB	0.085s
SPARTAN6	117KB	0.1MB	0.114s
VIRTEX	8KB	0.1MB	0.075s
VIRTEX2	18KB	0.1MB	0.079s
VIRTEX2P	88KB	0.1MB	0.108s
VIRTEX4	179KB	0.1MB	0.144s
VIRTEX5	322KB	0.1MB	0.191s
VIRTEX6	191KB	0.1MB	0.149s

Device Files

Family Name	Part Name	File Size	Java Heap Usage	Load Time From Disk
SPARTAN2	xc2s100fg256	134KB	11MB	0.111s
SPARTAN2	xc2s100fg456	134KB	11MB	0.107s
SPARTAN2	xc2s100pq208	134KB	11MB	0.108s
SPARTAN2	xc2s100tq144	134KB	11MB	0.111s
SPARTAN2	xc2s150fg256	150KB	16MB	0.159s
SPARTAN2	xc2s150fg456	150KB	16MB	0.157s
SPARTAN2	xc2s150pq208	150KB	16MB	0.158s
SPARTAN2	xc2s15cs144	84KB	2MB	0.052s
SPARTAN2	xc2s15tq144	84KB	2MB	0.052s
SPARTAN2	xc2s15vq100	84KB	2MB	0.052s
SPARTAN2	xc2s200fg256	179KB	21MB	0.196s

SPARTAN2	xc2s200fg456	179KB	21MB	0.195s
SPARTAN2	xc2s200pq208	179KB	21MB	0.200s
SPARTAN2	xc2s30cs144	103KB	4MB	0.067s
SPARTAN2	xc2s30pq208	103KB	4MB	0.067s
SPARTAN2	xc2s30tq144	103KB	4MB	0.067s
SPARTAN2	xc2s30vq100	103KB	4MB	0.068s
SPARTAN2	xc2s50fg256	117KB	7MB	0.088s
SPARTAN2	xc2s50pq208	117KB	7MB	0.088s
SPARTAN2	xc2s50tq144	118KB	7MB	0.088s
SPARTAN2E	xc2s100efg456	137KB	11MB	0.109s
SPARTAN2E	xc2s100eft256	137KB	11MB	0.109s
SPARTAN2E	xc2s100epq208	137KB	11MB	0.108s
SPARTAN2E	xc2s100etq144	137KB	11MB	0.108s
SPARTAN2E	xc2s150efg456	154KB	16MB	0.158s
SPARTAN2E	xc2s150eft256	154KB	16MB	0.156s
SPARTAN2E	xc2s150epq208	154KB	16MB	0.157s
SPARTAN2E	xc2s200efg456	189KB	21MB	0.199s
SPARTAN2E	xc2s200eft256	189KB	21MB	0.197s
SPARTAN2E	xc2s200epq208	189KB	21MB	0.198s
SPARTAN2E	xc2s300efg456	233KB	27MB	0.281s
SPARTAN2E	xc2s300eft256	233KB	27MB	0.285s
SPARTAN2E	xc2s300epq208	233KB	27MB	0.283s
SPARTAN2E	xc2s400efg456	326KB	42MB	0.373s
SPARTAN2E	xc2s400efg676	326KB	42MB	0.372s
SPARTAN2E	xc2s400eft256	326KB	42MB	0.374s
SPARTAN2E	xc2s50eft256	121KB	7MB	0.088s
SPARTAN2E	xc2s50epq208	121KB	7MB	0.087s
SPARTAN2E	xc2s50etq144	121KB	7MB	0.088s
SPARTAN2E	xc2s600efg456	390KB	60MB	0.610s
SPARTAN2E	xc2s600efg676	389KB	60MB	0.612s
SPARTAN3	xc3s1000fg320	357KB	48MB	0.377s
SPARTAN3	xc3s1000fg456	357KB	48MB	0.376s
SPARTAN3	xc3s1000fg676	357KB	48MB	0.373s
SPARTAN3	xc3s1000ft256	357KB	48MB	0.376s
SPARTAN3	xc3s1000lfg320	357KB	48MB	0.375s
SPARTAN3	xc3s1000lfg456	357KB	48MB	0.375s
SPARTAN3	xc3s1000lft256	357KB	48MB	0.376s
SPARTAN3	xc3s1500fg320	538KB	81MB	0.719s
SPARTAN3	xc3s1500fg456	538KB	81MB	0.712s
SPARTAN3	xc3s1500fg676	538KB	81MB	0.715s
SPARTAN3	xc3s1500lfg320	538KB	81MB	0.713s
SPARTAN3	xc3s1500lfg456	538KB	81MB	0.716s
SPARTAN3	xc3s1500lfg676	538KB	81MB	0.716s
SPARTAN3	xc3s2000fg456	739KB	122MB	1.088s

SPARTAN3	xc3s2000fg676	739KB	122MB	1.076s
SPARTAN3	xc3s2000fg900	739KB	122MB	1.091s
SPARTAN3	xc3s200ft256	106KB	13MB	0.144s
SPARTAN3	xc3s200pq208	106KB	13MB	0.143s
SPARTAN3	xc3s200tq144	106KB	13MB	0.144s
SPARTAN3	xc3s200vq100	106KB	13MB	0.145s
SPARTAN3	xc3s4000fg1156	912KB	168MB	1.377s
SPARTAN3	xc3s4000fg676	912KB	168MB	1.383s
SPARTAN3	xc3s4000fg900	912KB	168MB	1.371s
SPARTAN3	xc3s4000lfg900	912KB	168MB	1.381s
SPARTAN3	xc3s400fg320	203KB	23MB	0.198s
SPARTAN3	xc3s400fg456	203KB	23MB	0.198s
SPARTAN3	xc3s400ft256	203KB	23MB	0.197s
SPARTAN3	xc3s400pq208	203KB	23MB	0.199s
SPARTAN3	xc3s400tq144	203KB	23MB	0.198s
SPARTAN3	xc3s5000fg1156	1184KB	201MB	1.467s
SPARTAN3	xc3s5000fg676	1184KB	201MB	1.454s
SPARTAN3	xc3s5000fg900	1184KB	201MB	1.459s
SPARTAN3	xc3s50cp132	78KB	5MB	0.069s
SPARTAN3	xc3s50pq208	78KB	5MB	0.069s
SPARTAN3	xc3s50tq144	78KB	5MB	0.069s
SPARTAN3	xc3s50vq100	78KB	5MB	0.069s
SPARTAN3A	xc3s1400afg484	630KB	71MB	0.627s
SPARTAN3A	xc3s1400afg676	630KB	71MB	0.630s
SPARTAN3A	xc3s1400aft256	630KB	71MB	0.627s
SPARTAN3A	xc3s1400anfgg676	630KB	71MB	0.631s
SPARTAN3A	xc3s200afg320	194KB	14MB	0.157s
SPARTAN3A	xc3s200aft256	194KB	14MB	0.159s
SPARTAN3A	xc3s200anftg256	194KB	14MB	0.158s
SPARTAN3A	xc3s200avq100	194KB	14MB	0.158s
SPARTAN3A	xc3s400afg320	309KB	25MB	0.208s
SPARTAN3A	xc3s400afg400	309KB	25MB	0.209s
SPARTAN3A	xc3s400aft256	309KB	25MB	0.210s
SPARTAN3A	xc3s400anfgg400	309KB	25MB	0.211s
SPARTAN3A	xc3s50aft256	114KB	6MB	0.082s
SPARTAN3A	xc3s50antqg144	114KB	6MB	0.083s
SPARTAN3A	xc3s50atq144	114KB	6MB	0.082s
SPARTAN3A	xc3s50avq100	114KB	6MB	0.083s
SPARTAN3A	xc3s700afg400	456KB	38MB	0.363s
SPARTAN3A	xc3s700afg484	456KB	38MB	0.364s
SPARTAN3A	xc3s700aft256	456KB	38MB	0.361s
SPARTAN3A	xc3s700anfgg484	456KB	38MB	0.363s
SPARTAN3ADSP	xc3sd1800acs484	1073KB	114MB	1.101s
SPARTAN3ADSP	xc3sd1800afg676	1073KB	114MB	1.096s

SPARTAN3ADSP	xc3sd3400acs484	1443KB	163MB	1.410s
SPARTAN3ADSP	xc3sd3400afg676	1443KB	163MB	1.408s
SPARTAN3E	xc3s100ecp132	152KB	8MB	0.092s
SPARTAN3E	xc3s100etq144	152KB	8MB	0.092s
SPARTAN3E	xc3s100evq100	152KB	8MB	0.093s
SPARTAN3E	xc3s1200efg320	588KB	56MB	0.581s
SPARTAN3E	xc3s1200efg400	588KB	56MB	0.572s
SPARTAN3E	xc3s1200eft256	588KB	56MB	0.583s
SPARTAN3E	xc3s1600efg320	794KB	92MB	0.764s
SPARTAN3E	xc3s1600efg400	794KB	92MB	0.762s
SPARTAN3E	xc3s1600efg484	794KB	92MB	0.757s
SPARTAN3E	xc3s250ecp132	234KB	18MB	0.168s
SPARTAN3E	xc3s250eft256	234KB	18MB	0.168s
SPARTAN3E	xc3s250epq208	234KB	18MB	0.165s
SPARTAN3E	xc3s250etq144	234KB	18MB	0.168s
SPARTAN3E	xc3s250evq100	234KB	18MB	0.166s
SPARTAN3E	xc3s500ecp132	376KB	31MB	0.297s
SPARTAN3E	xc3s500efg320	376KB	31MB	0.295s
SPARTAN3E	xc3s500eft256	376KB	31MB	0.296s
SPARTAN3E	xc3s500epq208	376KB	31MB	0.295s
SPARTAN3E	xc3s500evq100	376KB	31MB	0.295s
SPARTAN6	xc6slx100csg484	530KB	28MB	0.337s
SPARTAN6	xc6slx100fgg484	530KB	28MB	0.336s
SPARTAN6	xc6slx100fgg676	530KB	28MB	0.334s
SPARTAN6	xc6slx100tcsg484	620KB	29MB	0.335s
SPARTAN6	xc6slx100tfgg484	620KB	29MB	0.335s
SPARTAN6	xc6slx100tfgg676	620KB	29MB	0.334s
SPARTAN6	xc6slx100tfgg900	620KB	29MB	0.335s
SPARTAN6	xc6slx150csg484	624KB	34MB	0.406s
SPARTAN6	xc6slx150fgg484	624KB	34MB	0.403s
SPARTAN6	xc6slx150fgg676	624KB	34MB	0.404s
SPARTAN6	xc6slx150fgg900	624KB	34MB	0.402s
SPARTAN6	xc6slx150tcsg484	713KB	35MB	0.403s
SPARTAN6	xc6slx150tfgg484	713KB	35MB	0.401s
SPARTAN6	xc6slx150tfgg676	713KB	35MB	0.405s
SPARTAN6	xc6slx150tfgg900	713KB	35MB	0.404s
SPARTAN6	xc6slx16cpg196	234KB	11MB	0.117s
SPARTAN6	xc6slx16csg225	234KB	11MB	0.115s
SPARTAN6	xc6slx16csg324	234KB	11MB	0.115s
SPARTAN6	xc6slx16ftg256	234KB	11MB	0.116s
SPARTAN6	xc6slx25csg324	307KB	20MB	0.213s
SPARTAN6	xc6slx25fgg484	307KB	20MB	0.211s
SPARTAN6	xc6slx25ftg256	307KB	20MB	0.211s
SPARTAN6	xc6slx25tcsg324	362KB	21MB	0.214s

SPARTAN6	xc6slx25tfgg484	362KB	21MB	0.214s
SPARTAN6	xc6slx45csg324	359KB	20MB	0.217s
SPARTAN6	xc6slx45csg484	359KB	20MB	0.218s
SPARTAN6	xc6slx45fgg484	359KB	20MB	0.214s
SPARTAN6	xc6slx45fgg676	359KB	20MB	0.218s
SPARTAN6	xc6slx45tcsg324	421KB	21MB	0.219s
SPARTAN6	xc6slx45tcsg484	421KB	21MB	0.223s
SPARTAN6	xc6slx45tfgg484	421KB	21MB	0.222s
SPARTAN6	xc6slx4cpg196	150KB	7MB	0.097s
SPARTAN6	xc6slx4csg225	150KB	7MB	0.095s
SPARTAN6	xc6slx4tqg144	150KB	7MB	0.095s
SPARTAN6	xc6slx75csg484	468KB	25MB	0.312s
SPARTAN6	xc6slx75fgg484	468KB	25MB	0.311s
SPARTAN6	xc6slx75fgg676	468KB	25MB	0.311s
SPARTAN6	xc6slx75tcsg484	555KB	26MB	0.315s
SPARTAN6	xc6slx75tfgg484	555KB	26MB	0.316s
SPARTAN6	xc6slx75tfgg676	555KB	26MB	0.318s
SPARTAN6	xc6slx9cpg196	215KB	9MB	0.104s
SPARTAN6	xc6slx9csg225	215KB	9MB	0.104s
SPARTAN6	xc6slx9csg324	215KB	9MB	0.103s
SPARTAN6	xc6slx9ftg256	215KB	9MB	0.103s
SPARTAN6	xc6slx9tqg144	215KB	9MB	0.102s
VIRTEX	xcv1000bg560	465KB	103MB	0.813s
VIRTEX	xcv1000fg680	465KB	103MB	0.815s
VIRTEX	xcv100bg256	132KB	11MB	0.108s
VIRTEX	xcv100cs144	132KB	11MB	0.107s
VIRTEX	xcv100fg256	132KB	11MB	0.108s
VIRTEX	xcv100pq240	132KB	11MB	0.108s
VIRTEX	xcv100tq144	132KB	11MB	0.108s
VIRTEX	xcv150bg256	148KB	16MB	0.157s
VIRTEX	xcv150bg352	148KB	16MB	0.156s
VIRTEX	xcv150fg256	148KB	16MB	0.157s
VIRTEX	xcv150fg456	148KB	16MB	0.158s
VIRTEX	xcv150pq240	148KB	16MB	0.167s
VIRTEX	xcv200bg256	176KB	21MB	0.197s
VIRTEX	xcv200bg352	177KB	21MB	0.196s
VIRTEX	xcv200fg256	176KB	21MB	0.195s
VIRTEX	xcv200fg456	177KB	21MB	0.196s
VIRTEX	xcv200pq240	176KB	21MB	0.196s
VIRTEX	xcv300bg352	216KB	27MB	0.282s
VIRTEX	xcv300bg432	216KB	27MB	0.284s
VIRTEX	xcv300fg456	216KB	27MB	0.282s
VIRTEX	xcv300pq240	216KB	27MB	0.282s
VIRTEX	xcv400bg432	274KB	41MB	0.365s

VIRTEX	xcv400bg560	274KB	41MB	0.370s
VIRTEX	xcv400fg676	274KB	41MB	0.369s
VIRTEX	xcv400hq240	274KB	41MB	0.367s
VIRTEX	xcv50bg256	116KB	7MB	0.086s
VIRTEX	xcv50cs144	116KB	7MB	0.086s
VIRTEX	xcv50fg256	116KB	7MB	0.087s
VIRTEX	xcv50pq240	116KB	7MB	0.087s
VIRTEX	xcv50tq144	116KB	7MB	0.087s
VIRTEX	xcv600bg432	330KB	60MB	0.611s
VIRTEX	xcv600bg560	330KB	60MB	0.606s
VIRTEX	xcv600fg676	330KB	60MB	0.605s
VIRTEX	xcv600fg680	330KB	60MB	0.603s
VIRTEX	xcv600hq240	330KB	60MB	0.610s
VIRTEX	xcv800bg432	399KB	79MB	0.652s
VIRTEX	xcv800bg560	399KB	79MB	0.658s
VIRTEX	xcv800fg676	399KB	79MB	0.651s
VIRTEX	xcv800fg680	399KB	79MB	0.657s
VIRTEX	xcv800hq240	399KB	79MB	0.652s
VIRTEX2	xc2v1000bg575	378KB	41MB	0.365s
VIRTEX2	xc2v1000ff896	378KB	41MB	0.362s
VIRTEX2	xc2v1000fg256	378KB	41MB	0.367s
VIRTEX2	xc2v1000fg456	378KB	41MB	0.365s
VIRTEX2	xc2v1500bg575	496KB	59MB	0.588s
VIRTEX2	xc2v1500ff896	496KB	59MB	0.585s
VIRTEX2	xc2v1500fg676	496KB	59MB	0.589s
VIRTEX2	xc2v2000bf957	599KB	81MB	0.735s
VIRTEX2	xc2v2000bg575	599KB	81MB	0.735s
VIRTEX2	xc2v2000ff896	599KB	81MB	0.735s
VIRTEX2	xc2v2000fg676	599KB	81MB	0.740s
VIRTEX2	xc2v250cs144	171KB	15MB	0.156s
VIRTEX2	xc2v250fg256	171KB	15MB	0.155s
VIRTEX2	xc2v250fg456	171KB	15MB	0.154s
VIRTEX2	xc2v3000bf957	759KB	109MB	1.102s
VIRTEX2	xc2v3000bg728	759KB	109MB	1.079s
VIRTEX2	xc2v3000ff1152	759KB	109MB	1.081s
VIRTEX2	xc2v3000fg676	759KB	109MB	1.075s
VIRTEX2	xc2v4000bf957	1083KB	169MB	1.413s
VIRTEX2	xc2v4000ff1152	1083KB	169MB	1.423s
VIRTEX2	xc2v4000ff1517	1083KB	169MB	1.420s
VIRTEX2	xc2v40cs144	96KB	3MB	0.067s
VIRTEX2	xc2v40fg256	96KB	3MB	0.065s
VIRTEX2	xc2v500fg256	290KB	26MB	0.281s
VIRTEX2	xc2v500fg456	290KB	26MB	0.280s
VIRTEX2	xc2v6000bf957	1381KB	242MB	2.153s

VIRTEX2	xc2v6000ff1152	1381KB	242MB	2.158s
VIRTEX2	xc2v6000ff1517	1381KB	242MB	2.149s
VIRTEX2	xc2v8000ff1152	2205KB	329MB	2.758s
VIRTEX2	xc2v8000ff1517	2205KB	329MB	2.755s
VIRTEX2	xc2v80cs144	113KB	6MB	0.081s
VIRTEX2	xc2v80fg256	113KB	6MB	0.081s
VIRTEX2P	xc2vp100ff1696	2520KB	348MB	2.850s
VIRTEX2P	xc2vp100ff1704	2515KB	348MB	2.858s
VIRTEX2P	xc2vp20ff1152	869KB	80MB	0.660s
VIRTEX2P	xc2vp20ff896	869KB	80MB	0.654s
VIRTEX2P	xc2vp20fg676	868KB	80MB	0.653s
VIRTEX2P	xc2vp2ff672	214KB	14MB	0.154s
VIRTEX2P	xc2vp2fg256	214KB	14MB	0.151s
VIRTEX2P	xc2vp2fg456	214KB	14MB	0.152s
VIRTEX2P	xc2vp30ff1152	1105KB	114MB	1.104s
VIRTEX2P	xc2vp30ff896	1105KB	114MB	1.113s
VIRTEX2P	xc2vp30fg676	1105KB	114MB	1.110s
VIRTEX2P	xc2vp40ff1148	1407KB	158MB	1.237s
VIRTEX2P	xc2vp40ff1152	1407KB	158MB	1.256s
VIRTEX2P	xc2vp40fg676	1407KB	158MB	1.229s
VIRTEX2P	xc2vp4ff672	450KB	28MB	0.289s
VIRTEX2P	xc2vp4fg256	450KB	28MB	0.290s
VIRTEX2P	xc2vp4fg456	450KB	28MB	0.291s
VIRTEX2P	xc2vp50ff1148	1601KB	190MB	1.504s
VIRTEX2P	xc2vp50ff1152	1596KB	190MB	1.500s
VIRTEX2P	xc2vp50ff1517	1595KB	190MB	1.500s
VIRTEX2P	xc2vp70ff1517	2052KB	263MB	2.243s
VIRTEX2P	xc2vp70ff1704	2052KB	263MB	2.239s
VIRTEX2P	xc2vp7ff672	610KB	44MB	0.383s
VIRTEX2P	xc2vp7ff896	610KB	44MB	0.381s
VIRTEX2P	xc2vp7fg456	610KB	44MB	0.381s
VIRTEX2P	xc2vpx20ff896	892KB	83MB	0.749s
VIRTEX2P	xc2vpx70ff1704	2117KB	263MB	2.261s
VIRTEX4	xc4vfx100ff1152	1357KB	120MB	1.166s
VIRTEX4	xc4vfx100ff1517	1357KB	120MB	1.166s
VIRTEX4	xc4vfx12ff668	598KB	38MB	0.378s
VIRTEX4	xc4vfx12sf363	598KB	38MB	0.374s
VIRTEX4	xc4vfx140ff1517	1546KB	140MB	1.421s
VIRTEX4	xc4vfx20ff672	924KB	63MB	0.608s
VIRTEX4	xc4vfx40ff1152	1159KB	94MB	0.800s
VIRTEX4	xc4vfx40ff672	1158KB	94MB	0.794s
VIRTEX4	xc4vfx60ff1152	1206KB	103MB	1.088s
VIRTEX4	xc4vfx60ff672	1206KB	103MB	1.083s
VIRTEX4	xc4vlx100ff1148	701KB	74MB	0.754s

VIRTEX4	xc4vlx100ff1513	701KB	74MB	0.753s
VIRTEX4	xc4vlx15ff668	231KB	34MB	0.307s
VIRTEX4	xc4vlx15ff676	231KB	34MB	0.307s
VIRTEX4	xc4vlx15sf363	231KB	34MB	0.312s
VIRTEX4	xc4vlx160ff1148	875KB	113MB	1.114s
VIRTEX4	xc4vlx160ff1513	875KB	113MB	1.113s
VIRTEX4	xc4vlx200ff1513	1010KB	115MB	1.152s
VIRTEX4	xc4vlx25ff668	287KB	37MB	0.371s
VIRTEX4	xc4vlx25ff676	287KB	37MB	0.372s
VIRTEX4	xc4vlx25sf363	287KB	37MB	0.369s
VIRTEX4	xc4vlx40ff1148	348KB	42MB	0.415s
VIRTEX4	xc4vlx40ff668	349KB	42MB	0.401s
VIRTEX4	xc4vlx60ff1148	464KB	64MB	0.629s
VIRTEX4	xc4vlx60ff668	464KB	64MB	0.625s
VIRTEX4	xc4vlx80ff1148	596KB	74MB	0.746s
VIRTEX4	xc4vsx25ff668	373KB	62MB	0.612s
VIRTEX4	xc4vsx35ff668	441KB	64MB	0.617s
VIRTEX4	xc4vsx55ff1148	539KB	74MB	0.734s
VIRTEX5	xc5vfx100tff1136	1014KB	107MB	1.115s
VIRTEX5	xc5vfx100tff1738	1014KB	107MB	1.121s
VIRTEX5	xc5vfx130tff1738	1058KB	115MB	1.136s
VIRTEX5	xc5vfx200tff1738	1227KB	129MB	1.232s
VIRTEX5	xc5vfx30tff665	781KB	63MB	0.659s
VIRTEX5	xc5vfx70tff1136	899KB	84MB	0.796s
VIRTEX5	xc5vfx70tff665	898KB	84MB	0.798s
VIRTEX5	xc5vlx110ff1153	589KB	74MB	0.779s
VIRTEX5	xc5vlx110ff1760	589KB	74MB	0.769s
VIRTEX5	xc5vlx110ff676	588KB	74MB	0.774s
VIRTEX5	xc5vlx110tff1136	761KB	80MB	0.782s
VIRTEX5	xc5vlx110tff1738	761KB	80MB	0.784s
VIRTEX5	xc5vlx155ff1153	716KB	102MB	1.101s
VIRTEX5	xc5vlx155ff1760	716KB	102MB	1.106s
VIRTEX5	xc5vlx155tff1136	893KB	112MB	1.124s
VIRTEX5	xc5vlx155tff1738	893KB	112MB	1.124s
VIRTEX5	xc5vlx20tff323	497KB	39MB	0.408s
VIRTEX5	xc5vlx220ff1760	862KB	138MB	1.244s
VIRTEX5	xc5vlx220tff1738	1039KB	143MB	1.420s
VIRTEX5	xc5vlx30ff324	380KB	41MB	0.419s
VIRTEX5	xc5vlx30ff676	381KB	41MB	0.415s
VIRTEX5	xc5vlx30tff323	558KB	46MB	0.440s
VIRTEX5	xc5vlx30tff665	558KB	46MB	0.440s
VIRTEX5	xc5vlx330ff1760	1069KB	147MB	1.470s
VIRTEX5	xc5vlx330tff1738	1250KB	153MB	1.435s
VIRTEX5	xc5vlx50ff1153	417KB	42MB	0.434s

VIRTEX5	xc5vlx50ff324	417KB	42MB	0.426s
VIRTEX5	xc5vlx50ff676	417KB	42MB	0.433s
VIRTEX5	xc5vlx50tff1136	587KB	47MB	0.452s
VIRTEX5	xc5vlx50tff665	586KB	47MB	0.457s
VIRTEX5	xc5vlx85ff1153	535KB	71MB	0.674s
VIRTEX5	xc5vlx85ff676	534KB	71MB	0.680s
VIRTEX5	xc5vlx85tff1136	706KB	77MB	0.779s
VIRTEX5	xc5vsx240tff1738	1135KB	128MB	1.243s
VIRTEX5	xc5vsx35tff665	596KB	59MB	0.642s
VIRTEX5	xc5vsx50tff1136	630KB	61MB	0.654s
VIRTEX5	xc5vsx50tff665	630KB	61MB	0.652s
VIRTEX5	xc5vsx95tff1136	754KB	85MB	0.804s
VIRTEX5	xc5vtx150tff1156	870KB	90MB	0.833s
VIRTEX5	xc5vtx150tff1759	871KB	90MB	0.832s
VIRTEX5	xc5vtx240tff1759	1111KB	123MB	1.182s
VIRTEX6	xc6vcx130tff1156	715KB	63MB	0.658s
VIRTEX6	xc6vcx130tff484	715KB	63MB	0.652s
VIRTEX6	xc6vcx130tff784	715KB	63MB	0.660s
VIRTEX6	xc6vcx195tff1156	853KB	77MB	0.800s
VIRTEX6	xc6vcx195tff784	853KB	77MB	0.807s
VIRTEX6	xc6vcx240tff1156	937KB	81MB	0.829s
VIRTEX6	xc6vcx240tff784	937KB	81MB	0.819s
VIRTEX6	xc6vcx75tff484	583KB	54MB	0.615s
VIRTEX6	xc6vcx75tff784	584KB	54MB	0.615s
VIRTEX6	xc6vhx250tff1154	973KB	82MB	0.821s
VIRTEX6	xc6vhx255tff1155	1029KB	88MB	0.849s
VIRTEX6	xc6vhx255tff1923	1029KB	88MB	0.870s
VIRTEX6	xc6vhx380tff1154	1285KB	98MB	1.210s
VIRTEX6	xc6vhx380tff1155	1286KB	98MB	1.264s
VIRTEX6	xc6vhx380tff1923	1288KB	98MB	1.318s
VIRTEX6	xc6vhx380tff1924	1287KB	98MB	1.200s
VIRTEX6	xc6vhx565tff1923	1661KB	126MB	1.315s
VIRTEX6	xc6vhx565tff1924	1660KB	126MB	1.313s
VIRTEX6	xc6vlx130tff1156	709KB	63MB	0.660s
VIRTEX6	xc6vlx130tff484	709KB	63MB	0.668s
VIRTEX6	xc6vlx130tff784	709KB	63MB	0.669s
VIRTEX6	xc6vlx195tff1156	849KB	77MB	0.829s
VIRTEX6	xc6vlx195tff784	848KB	77MB	0.896s
VIRTEX6	xc6vlx240tff1156	934KB	81MB	0.831s
VIRTEX6	xc6vlx240tff1759	935KB	81MB	0.830s
VIRTEX6	xc6vlx240tff784	934KB	81MB	0.820s
VIRTEX6	xc6vlx365tff1156	1186KB	107MB	1.201s
VIRTEX6	xc6vlx365tff1759	1187KB	107MB	1.207s
VIRTEX6	xc6vlx550tff1759	1550KB	122MB	1.299s

VIRTEX6	xc6vlx550tff1760	1551KB	122MB	1.302s
VIRTEX6	xc6vlx75tff484	582KB	54MB	0.689s
VIRTEX6	xc6vlx75tff784	582KB	54MB	0.620s
VIRTEX6	xc6vlx760ff1760	1758KB	136MB	1.606s
VIRTEX6	xc6vsx315tff1156	1157KB	107MB	1.190s
VIRTEX6	xc6vsx315tff1759	1157KB	107MB	1.192s
VIRTEX6	xc6vsx475tff1156	1505KB	121MB	1.298s
VIRTEX6	xc6vsx475tff1759	1506KB	121MB	1.280s