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.

CONTENTS

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.

The RapidSmith project does not manipulate Xilinx bitstreams and does not contain any information of the organization of such files. It also does not contain or provide methods of including FPGA timing information. 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?

Currently the only tested families of Xilinx FPGAs are the Virtex 4 and Virtex 5. However, the code is organized and written to support almost all FPGA families supported by ISE 11.1 to 12.2, these being: Virtex 4, Virtex 5, Virtex 6, Spartan 3A, Spartan 3ADSP, Spartan 3E and Spartan 6. There is currently no plans to support the Spartan 3 family, however, the code could be taken independently and changed to make it to work with the older architecture.

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 fast 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.

GETTING STARTED

Installation

Requirements for Installation

- 200 MB free disk space (for all Virtex 4 and Virtex 5 family devices)
  - NOTE: ~30 GB of free hard disk space needed during installation (XDLRC files are very large in size)
- Windows XP/Vista/7 or Linux
- Xilinx ISE 11.1 or higher
- JDK 1.6 (earlier versions may work, but have not been tested).
- Caucho Hessian Implementation JAR v.4.0.6 (Used for compressing database device files)
- 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.
- OPTIONAL: Qt Jambi (Qt for Java) for the Part Tile Browser example. Just adding the jars to the CLASSPATH variable is adequate.

Steps for Installation

1. Make sure the Xilinx tools and JDK are on your `PATH`.
2. Add the hessian-4.0.6.jar file 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 to where you have the Java project installed.
5. Compile all of the Java classes (this can be done automatically if the project is imported into an IDE such as Eclipse).
6. Be sure that the "device" directory found in the distribution ZIP file is copied to the base location of the project (where `RAPIDSMITH_PATH` is pointing).
7. 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 running the main method in the class `edu.byu.ece.rapidSmith.util.Installer` 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.
   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
   ```

Overview

RapidSmith is organized into several packages:

Package Name	Description
design	Represents all of the constructs in XDL files (Instances, Nets, PIPs, Modules, Designs).
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.helper	Some classes to help in the creation of the device files.
examples	Some user examples of how to use RapidSmith.
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, …).
placer	This contains classes to place designs.
router	This contains classes to route designs.
util	This contains miscellaneous support classes and utilities.

`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:

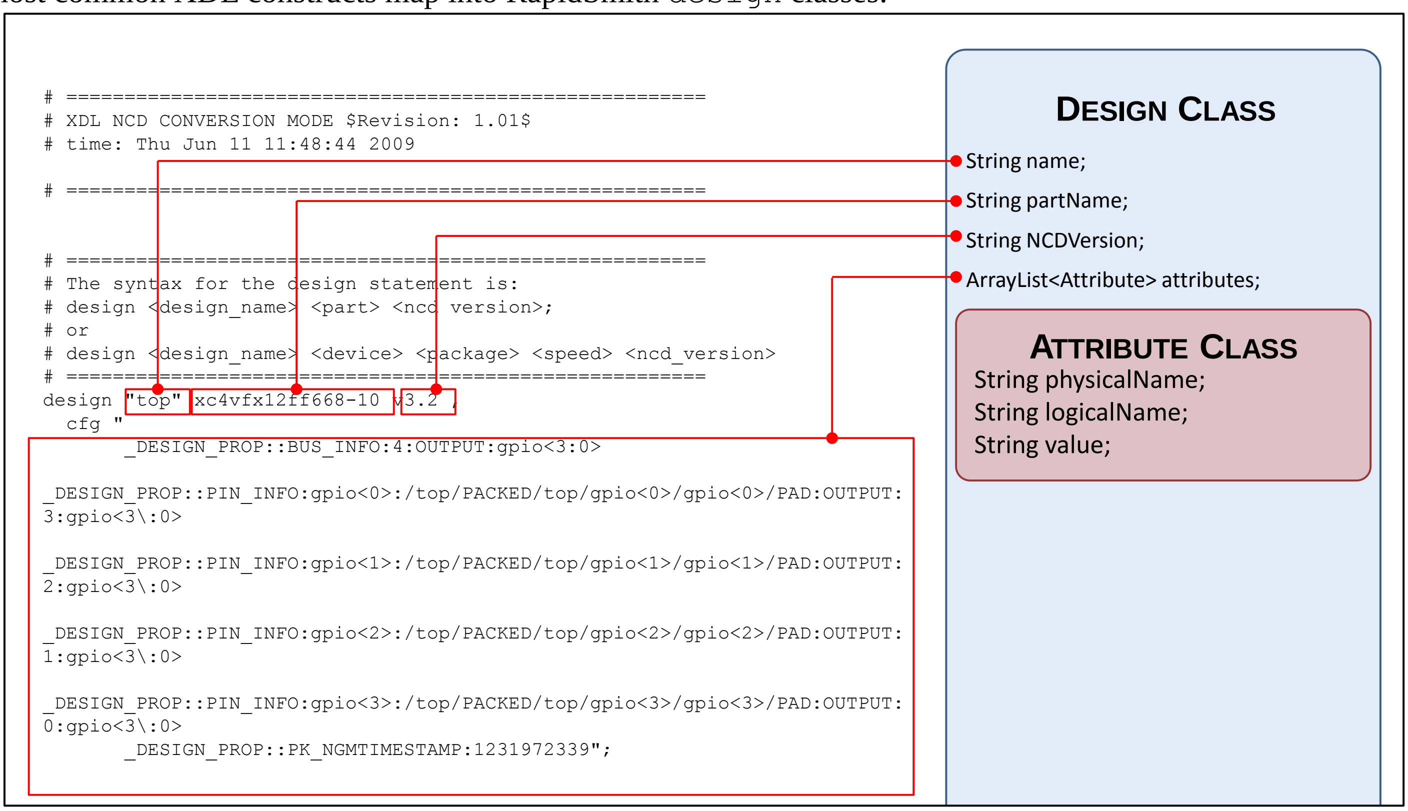

Figure 1 - Design and Attribute Classes

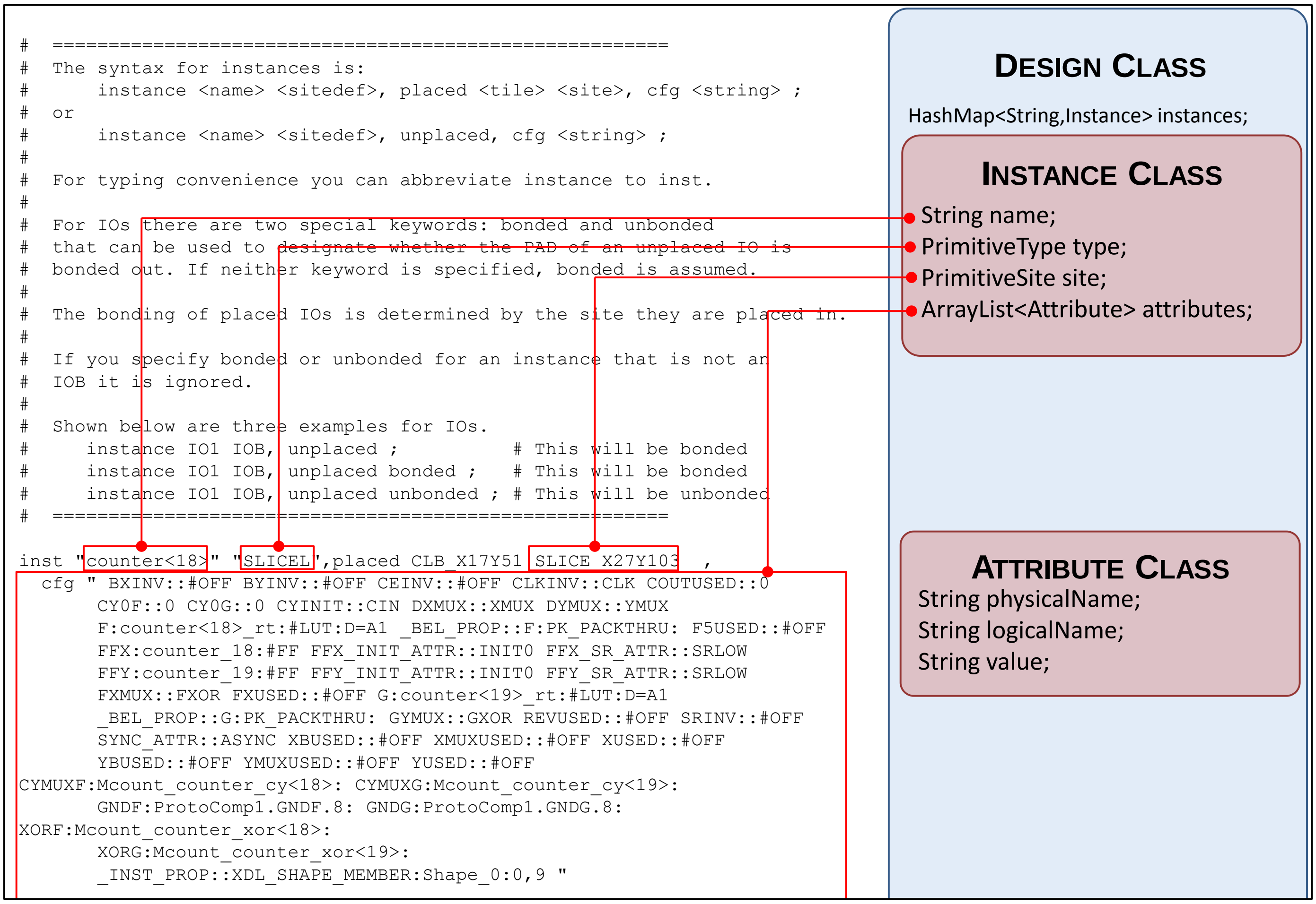

Figure 2 - Instance Class

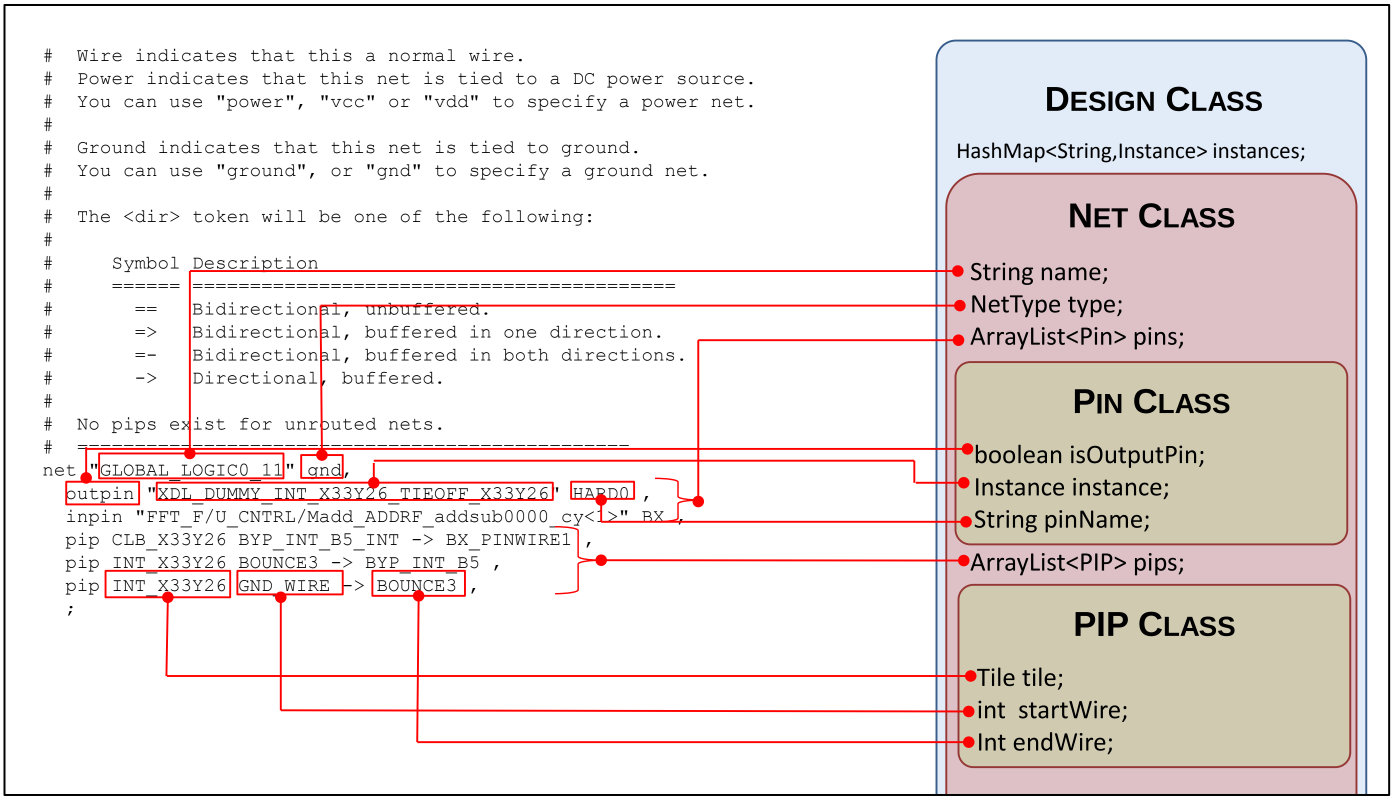

Figure 3 - Net, Pin and PIP Classes

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.parser` Package

The XDL design parser is written in JavaCC which compiles a `.jj` file into multiple .java files which implement a full-fledged parser. It will populate an instance of the Design class with the XDL design found in the `.xdl` file.

`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, …) wire direction (NORTH, SOUTH, EAST, …) among other attributes.

`examples` Package

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

`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.

`placer` Package

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

`router` Package

This package still has yet to be completed but will have an example of a router that routes Virtex 4 and Virtex 5 designs.

`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 example requires the Qt Jambi jar files as mentioned above in Requirements for Installation.

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.

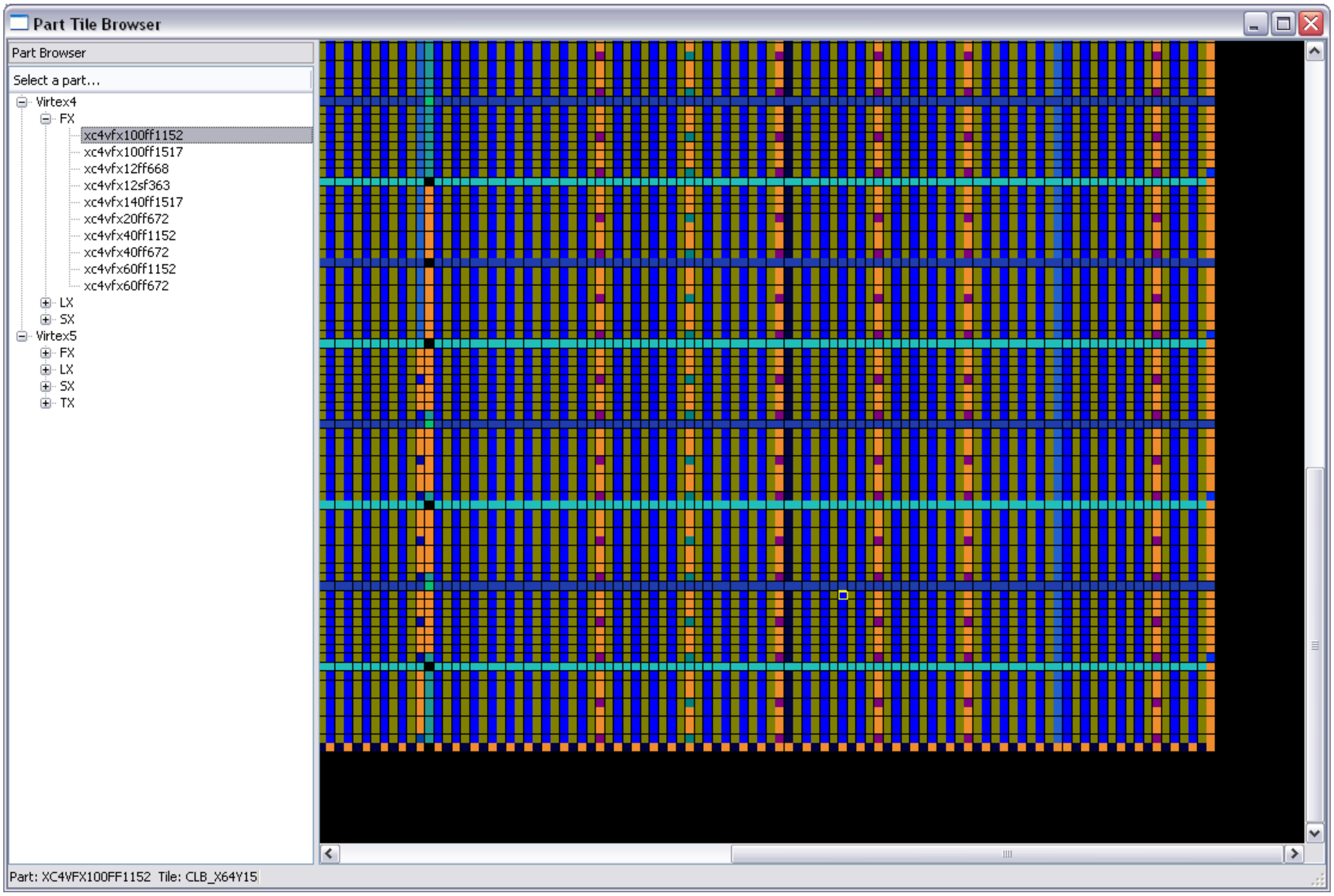

Figure 4: Screenshot of Part Tile Browser

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)

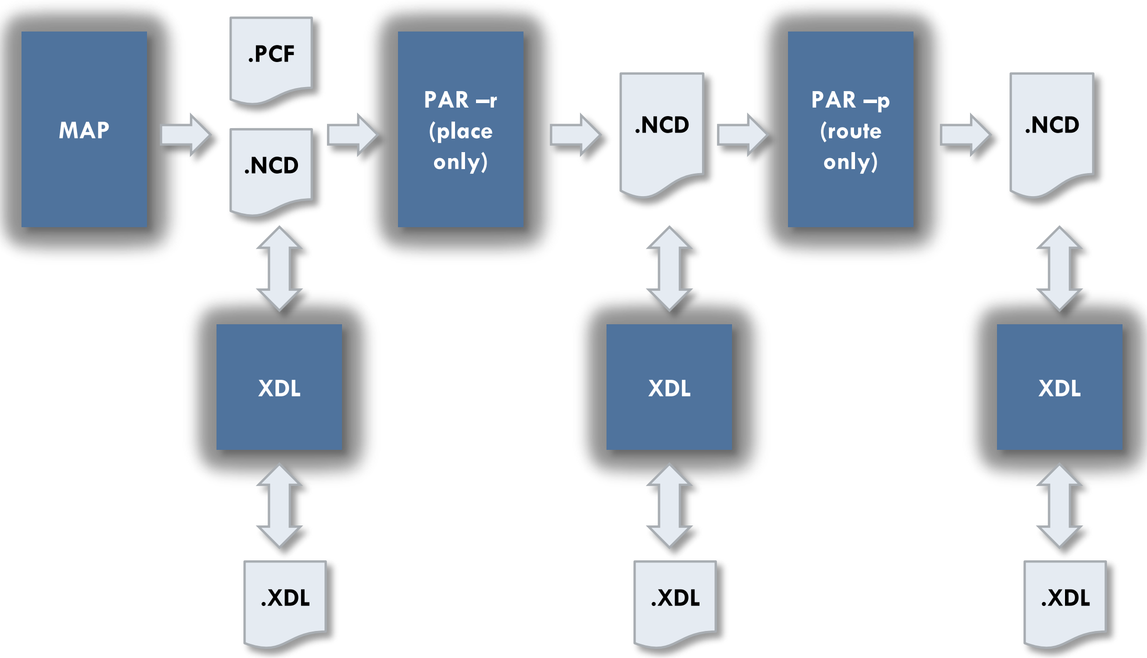

Figure 5: Block diagram of where XDL fits in CAD flow.

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 method 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. RapidSmith, during installation will create XDLRC files and parse them for their pertinent information and then packed 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 give 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 also 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) contains 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.

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, 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 devices 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.

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 also 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 loaded in with the design reference those same resources in the `device` class.

Saving Designs

RapidSmith has a method also 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.

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.

