A Toolkit for Rapid FPGA System Deployment

Umang K. Parekh

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Peter Athanas, Chair
Patrick Schaumont
Paul Plassmann

November 12, 2010
Blacksburg, Virginia

Keywords: FPGA, Router, Virtex-4, Toolkit, Autonomous

A Toolkit for Rapid FPGA System Deployment

Umang K. Parekh

(ABSTRACT)

FPGA implementation tools have not kept pace with growing FPGA density. It is common for non-trivial designs to take multiple hours to go through the entire FPGA toolflow (synthesis, mapping, placement, routing, bitstream generation). FPGA implementation tool runtime is a major hindrance to FPGA Productivity.

In modern FPGA designs, designers often change logic and/or connections in an already existing design. If small modifications are made to a particular module in a design, then almost the entire design will go through most of the FPGA toolflow again. This can be time consuming for complex designs and hinder productivity of FPGA designers. The main goal of this thesis is to improve FPGA productivity by reducing FPGA design implementation time for modifications made to an already existing design for rapid system deployment.

In this thesis, a toolkit is presented, which is capable of making design modifications at a lower level of abstraction for already existing designs on Xilinx FPGAs. The toolkit is a part of the open-source RapidSmith framework and includes the EDIF parser, mapper, placer, and router. It can be used to change logic and/or modify connections. Modules can be placed, unplaced, relocated, and/or duplicated with ease using this toolkit. Significant time-savings were seen by making use of the toolkit along-with the standard Xilinx FPGA toolflow, for making design modifications to already existing designs.

Acknowledgements

I would like to express my gratitude for my family who has been very supportive of my studies and graduate education. I could not have completed this work without the love and faith of my parents - Nita Parekh and Kumar Parekh and my brother - Kaushal. I am what I am today just because of your support and encouragement.

I would like to sincerely thank my advisor Dr. Peter Athanas for his continued guidance and insightful suggestions throughout the duration of my research. It has been a privilege, and a tremendously rewarding experience.

I would also like to thank Dr. Patrick Schaumont and Dr. Paul Plassmann for serving as members of my committee.

I would also like to thank Prof. Tom Walker for providing teaching assistantship to me. I am truly indebted by your gesture. I really learnt a lot from you and my TA experience with 1104.

I am thankful to my friends in the CCM Lab for enriching my research experience - Rohit Asthana, Wenwei Zha, Mrudula Karve, Abhay Tavaragiri, Karl Pareira, Sushrutha Vigraham, Sureshwar Rajagopalan, Ali Sohanghpurwala, Prabhaav Bharadwaj, Tony Frangieh, Jacob Couch and Adolfo Recio. It has been a real pleasure working with so many of you, and I will remember this place with many fond memories.

I would also like to thank all my friends, especially Urmila, for always being there for me and making me a better person.

Finally, I am thankful to God for His countless blessings.

Contents

List of Figures

Chapter 1

Introduction

Modern Field-Programmable Gate Arrays (FPGAs) have multi-millions of gates and future generations of FPGAs will only have more gates. However, the FPGA tools have not kept pace with growing FPGA density. It is common for modern designs to take multiple hours to synthesize, map, place, route, and generate bitstream. FPGA productivity is the major hindrance that the FPGA community is facing. One of the ways to improve FPGA productivity is by reducing the FPGA tool runtime. In this thesis, the aim is to improve FPGA productivity by reducing the FPGA design implementation time for modifications made to an already existing design for rapid system deployment. The toolkit presented in this thesis includes important modules of FPGA toolflow such as the Electronic Design Interchange Format [4] (EDIF) parser [9], mapper, placer, and router, in an open-source framework, RapidSmith [1].

For example, if a design has 100 modules, and a small design modification is made to Module 1, Module 1 will go through all the stages of FPGA toolflow, while the other 99 blocks might have to go through placement and routing stages again. This is redundant and would be time consuming for complex designs.

The Xilinx [3] tools have tried to address this issue with additional design features such as Partitions [10] and SmartGuide [12]. Xilinx's FPGA Editor [11] can be used to make design modifications as well. The use of hard macros in design is very common to avoid excessive tool runtime. The modules, for which the design phase is complete and the performance is acceptable, are saved as hard macros, which can then be instantiated into the design.

Several designs were tried with Partitions and/or SmartGuide features turned on; however, they failed at the mapping stage in ISE 12.1 [14]. Even if the features were working, the results would not be very flexible. For example, there might be design requirements where a particular amount of delay is needed between two points, or the longest route between two endpoints. The Xilinx tools do not handle such situations very well. Designers can use FPGA Editor in such situations and hand-route the net.

For experts, it is easy to make hand modifications in a design using FPGA Editor, but it is time consuming. This process can also be automated using Tcl scripts. However, implementing functionality such as module relocation would be difficult with FPGA Editor, even with the use of scripting. A hard macro based approach has been tried before [19]. This approach involves developing a tool to generate hard macros. A 3x reductions in FPGA implementation time was reported using this approach.

VPR (Versatile Place and Route) [8], is another open-source FPGA CAD tool which has been used by the FPGA research community for research-based FPGAs; however, VPR currently works only for FPGAs that can be defined by its architecture description format. Describing commercially available Xilinx FPGAs in terms of VPR's architecture description format was not considered a viable option.

The toolkit presented in this thesis contains tools such as the EDIF parser, mapper, placer, and router, which are necessary to make modifications in a design. There have been many

discrete efforts to improve the FPGA toolflow. Those discrete efforts were combined into a single toolkit so that it is easier for the end user to be utilized. The open-source EDIF parser was used from Brigham Young University (BYU). The mapper, placer, and router were developed from the Autonomous Adaptive Systems (AAS) project [13]. The router was ported over from C++ to Java to make it compatible with the RapidSmith framework.

Being open-source, the toolkit is easy to modify for customized design requirements and can be easily extended. The toolkit has strong Application Programming Interfaces (APIs) for addition/deletion/modification of placed/unplaced and/or routed/unrouted modules. The toolkit also contains a hand-placer and hand-router for customized design requirements. The entire framework is intuitive and easy to use such that it makes design modifications easier for the user. The toolkit, when used along with the Xilinx tools intelligently, decreases the tool runtime, and thus increases the overall productivity of FPGA designers. The toolkit can be used as an auxiliary tool to the original Xilinx toolflow to improve productivity as shown in Figure 1.1.

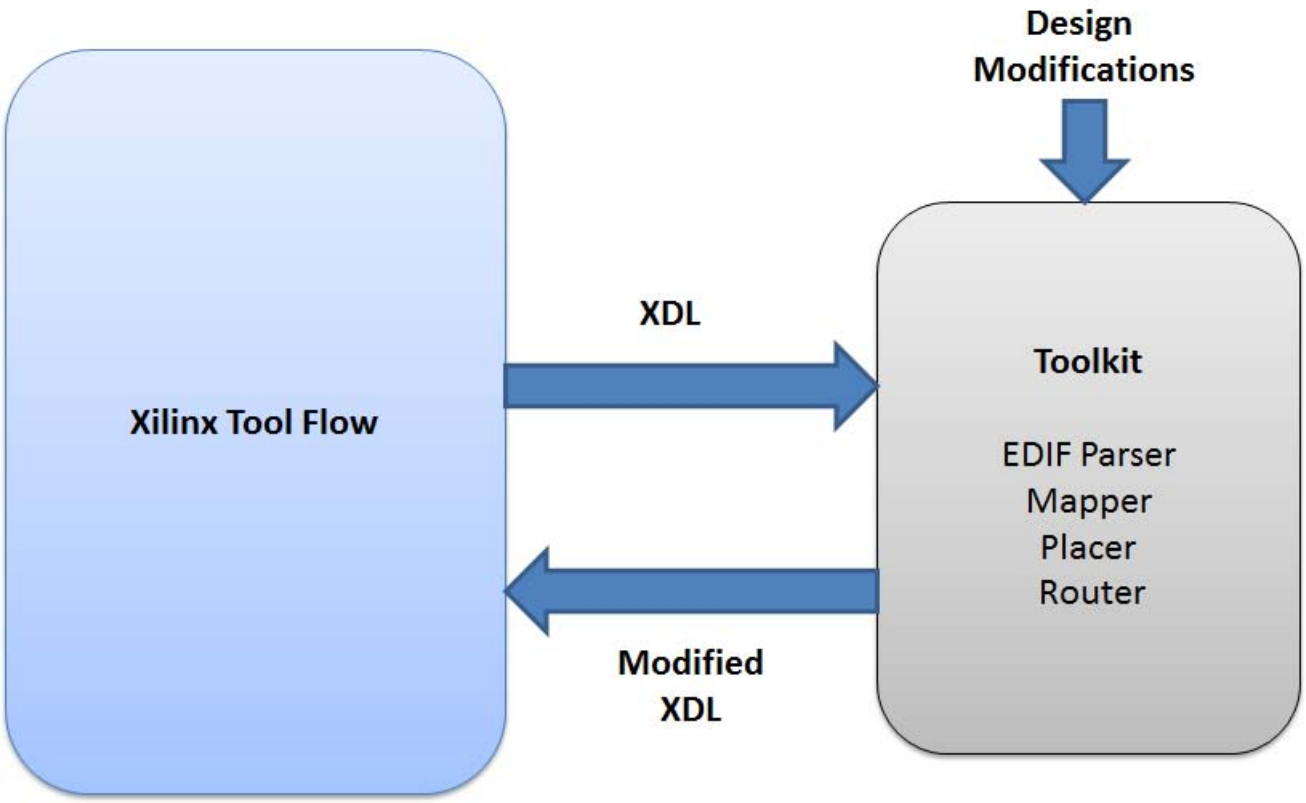


Figure 1.1 Design modification flow using the toolkit.

Chapter 2 gives the background information necessary to understand this thesis and some information about previous work along similar lines. Chapter 3 gives the system overview of the entire toolflow and describes the interface of all the tools in detail. Chapter 4 shows the results obtained by using the proposed toolkit over standard FPGA design flow. Chapter 5 discusses future work that has to be done on the framework to make it even more powerful. Finally, chapter 6 concludes the thesis.

Chapter 2

Background

In this thesis, auxiliary Electronic Design Automation (EDA) tools are created for Xilinx FPGAs, which when used along with Xilinx tools will result in improved productivity. Hence, it is important for the readers to have a good understanding about the underlying Xilinx architecture. It is also essential for the reader to have an in-depth knowledge about what each step in the toolflow does for Xilinx FPGAs. RapidSmith [1], the framework in which the tools have been built, deals only in Xilinx Descriptive Language. Thus, detailed information about Xilinx Descriptive Language is essential to understand this work. Tools such as the mapper, placer, and router have been influenced a lot from previous works: Adaptive Autonomous Systems [13] and Adaptive Computing Systems [2]. Many features of the tools were dictated by these previous works, so it is essential to know about them - to understand why a particular design decision was made for the mapper, placer, and router.

This chapter covers background topics, related work, and previous work that pertain to this thesis. It begins with a discussion of FPGAs, its logic resources, and routing architecture. Standard FPGA design flow is introduced in the second section. Xilinx Descriptive Language is introduced in the third section. XDLRC file (with extension *.xdlrc*), which gives detailed

internal information about Xilinx architecture is also described in this section. The next section describes the RapidSmith framework in detail. The last section in this chapter presents some of the previous work and an evaluation of already existing tools which were considered for this work.

2.1 Field Programmable Gate Arrays (FPGAs)

FPGAs are programmable semiconductor devices which are designed to be configured by the customer or designer after manufacturing, hence, "field-programmable". Xilinx FPGAs contain configurable logic blocks (CLBs) that are connected through programmable interconnect points (PIPs). CLBs can be configured to perform complex combinational functions, or merely simple logic gates such as AND and XOR. The ability to update the functionality after shipping, partial reconfiguration of the portion of the design and the low non-recurring engineering costs make FPGAs indispensible from being used for prototyping, testing, verification and/or in final products.

All Xilinx FPGAs contain the same basic resources: the slice, IOBs, programmable interconnects and other resources such as memory, multipliers, and clock buffers as shown in Figure 2.1 [17].

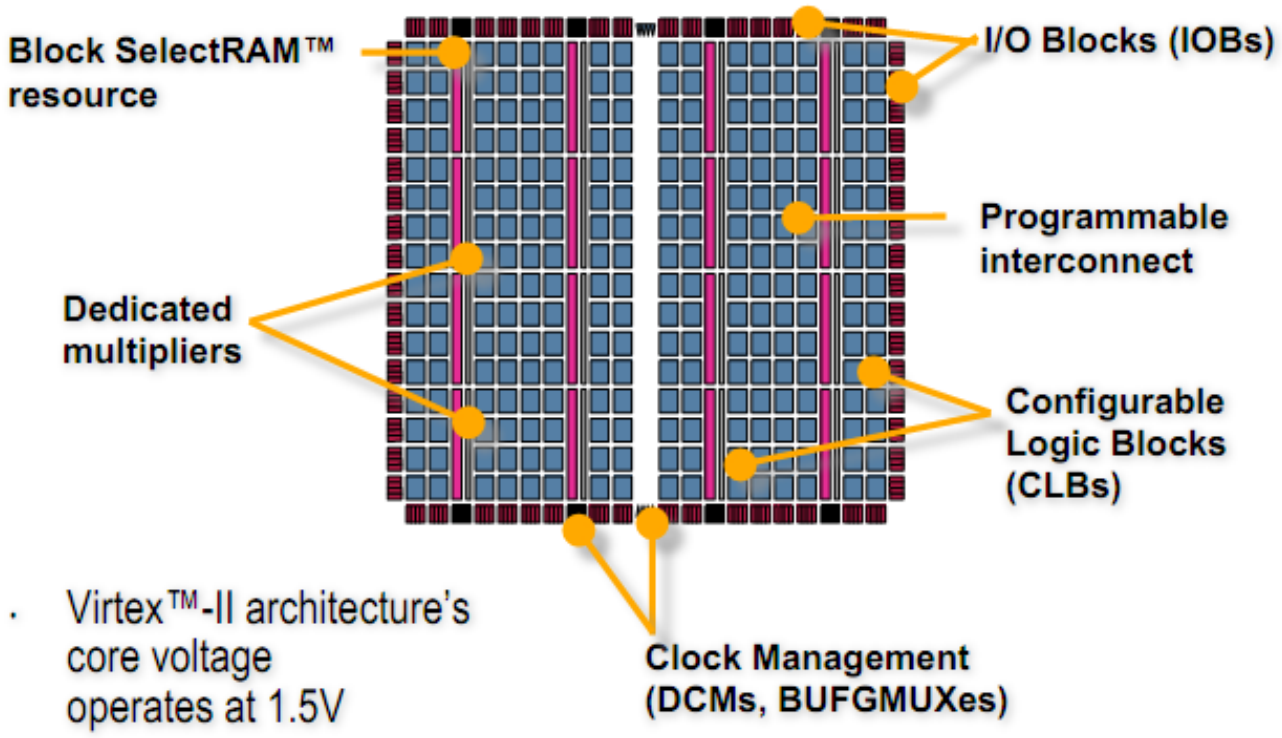


Figure 2.1 Virtex II FPGA (figure from [17]).

2.1.1 Configurable Logic Blocks (CLBs)

CLBs, as shown in Figure 2.2 [17], are comprised of slices. The number of slices within a CLB depends upon the architecture. Virtex-4 architecture contains four slices within each CLB. Local routing provides connections between the slices and the neighboring CLBs. The switch matrix provides access to the general routing resources such as double wire, hex wire, and long wires.

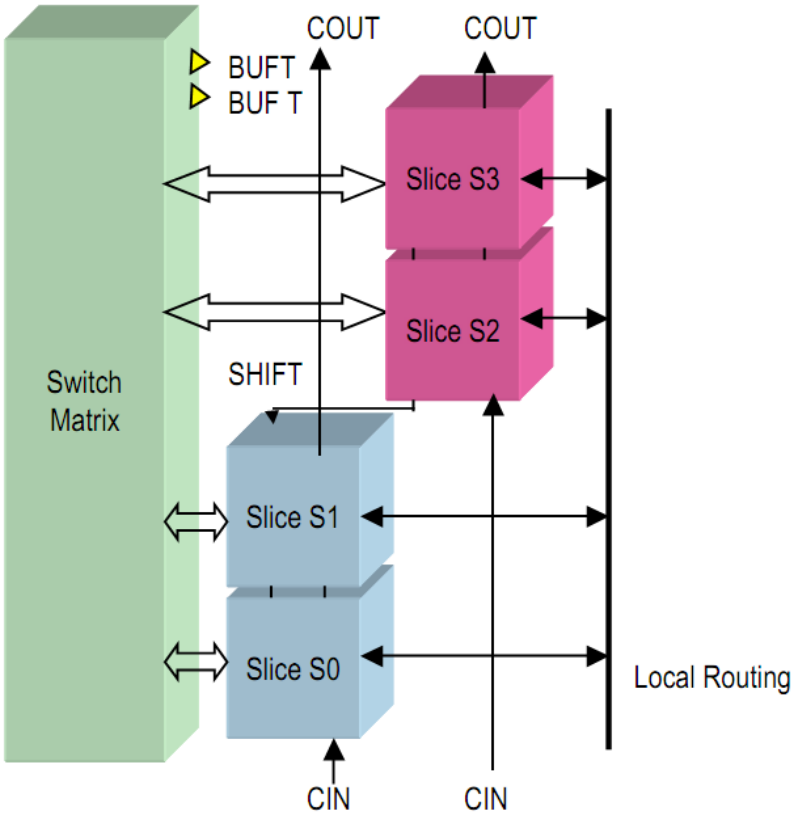


Figure 2.2 Virtex II CLB (figure from [17]).

2.1.2 Slice

A Slice, as shown in Figure 2.3 [17], is comprised of two or more LUTs (discussed in the next subsection). The number of LUTs within a slice depends upon the architecture. Slice has four outputs: two registered outputs and two non-registered outputs. The Virtex-4 architecture contains two LUTs within each slice. There are two kinds of slices: SLICEL and SLICEM. SLICEL can be used for logic only, while SLICEM can be used to implement logic or distributed RAM or shift register.

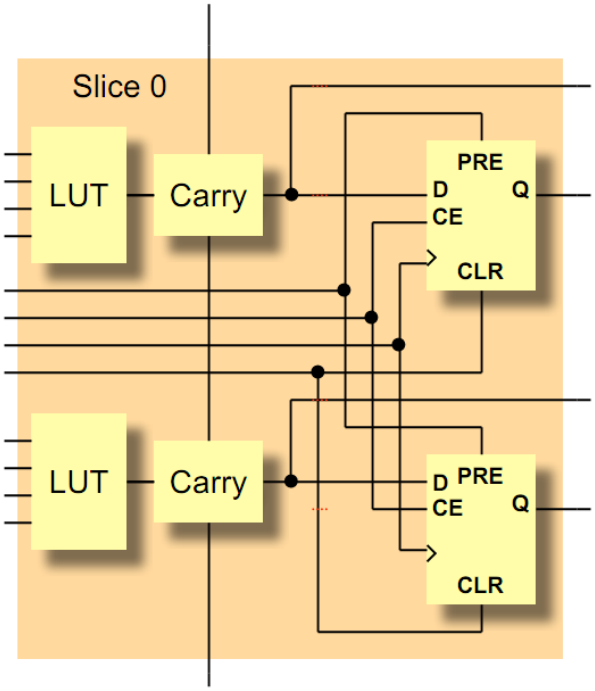
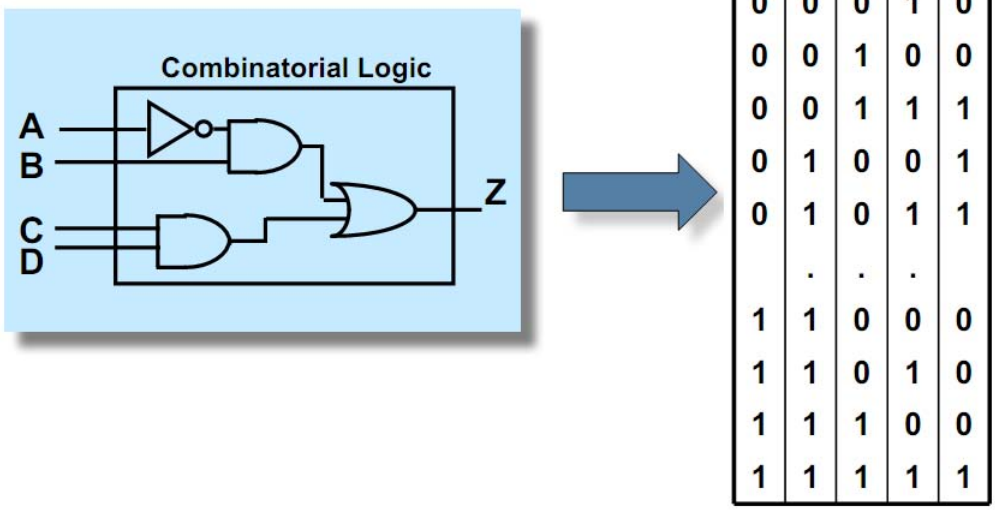


Figure 2.3 Xilinx Slice Structure (figure from [17]).

2.1.3 Lookup Tables (LUTs)

Lookup Tables, as shown in Figure 2.4 [17], or LUTs are also called function generators. If a LUT has n inputs, the truth table for all the 2^n combinations are stored, and depending upon the input combination, the output value is selected. Thus, the delay through the LUT is constant. Any function with k inputs can be implemented using a single LUT or by combining multiple LUTs. The mapper, described in this thesis, maps a given function only into LUTs and flip-flops. Until Virtex-4, all Xilinx architectures had 4-input LUTs. Virtex-5 and Virtex-6 both have 6-input LUTs.



A	B	C	D	Z
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
	.	.	.	
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figure 2.4 A LUT (figure from [17]).

2.1.4 Routing Architecture

The toolkit is compatible with the Virtex-4 architecture. This section will explain the Xilinx Virtex-4 routing architecture in detail. The Virtex-4 architecture has three kinds of routing resources: local, general purpose, and global [5].

Local routing resources: These provide direct connections between adjacent CLBs and feedback to the inputs of different LUTs. These direct connections bypass the routing matrix and provide high-speed connections to adjacent CLBs, as seen in Figure 2.5 [5].

General-purpose routing resources: These include long lines, hex lines, and double lines. Each CLB connects to a General Routing Matrix (GRM). Connections can be made from one GRM to other GRM in the vertical and/or horizontal direction. From each GRM, there are double length lines (or doubles) in each of the four directions. Hex length lines (or

hexes) are available in each of the four directions that connect to a GRM six blocks away. The long lines run horizontally or vertically for the length of the chip. Access to the long lines can be made every six blocks.

Global routing resources: The global routing resources can distribute high-fanout signals with minimal skew. These include four dedicated global nets with dedicated pins to distribute high-fanout clock signals.

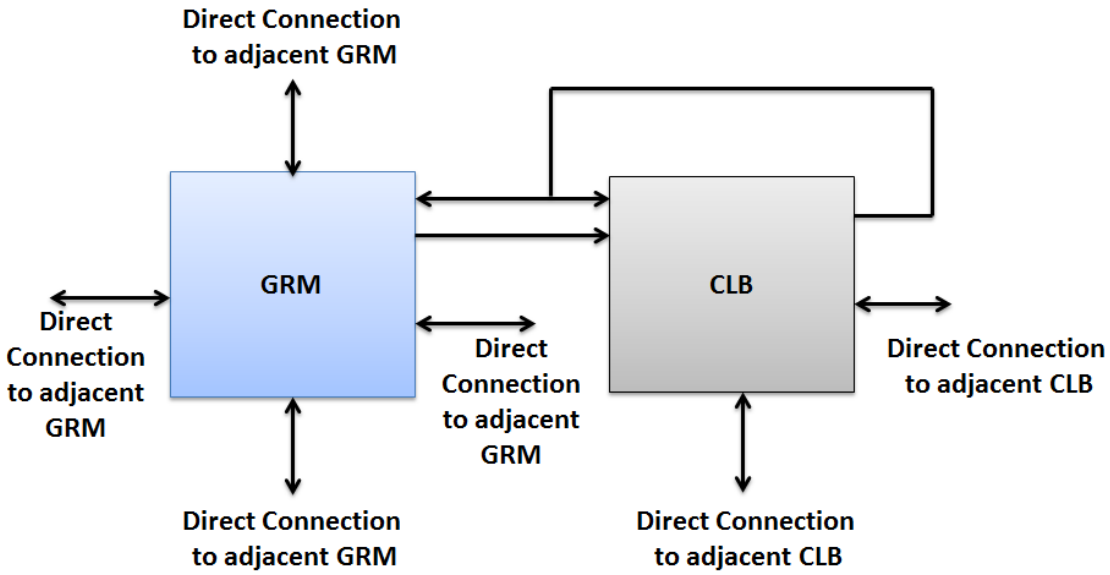


Figure 2.5 Routing Architecture (figure from [5]).

For example, a particular CLB in Virtex-4 contains doubles and hexes in all the four direction. The naming convention of the hexes and doubles follow a particular pattern. The first character indicates the direction in which the wire is travelling (N, E, W, or S). The second character either has "2" or "6" indicating if it is a double wire or hex wire respectively. The next three characters can be "BEG", "MID", or "END" indicating if the current point is the beginning, middle or the end of the wire and then finally the wire number. The naming convention can be clearly visualized from the Figure 2.6.

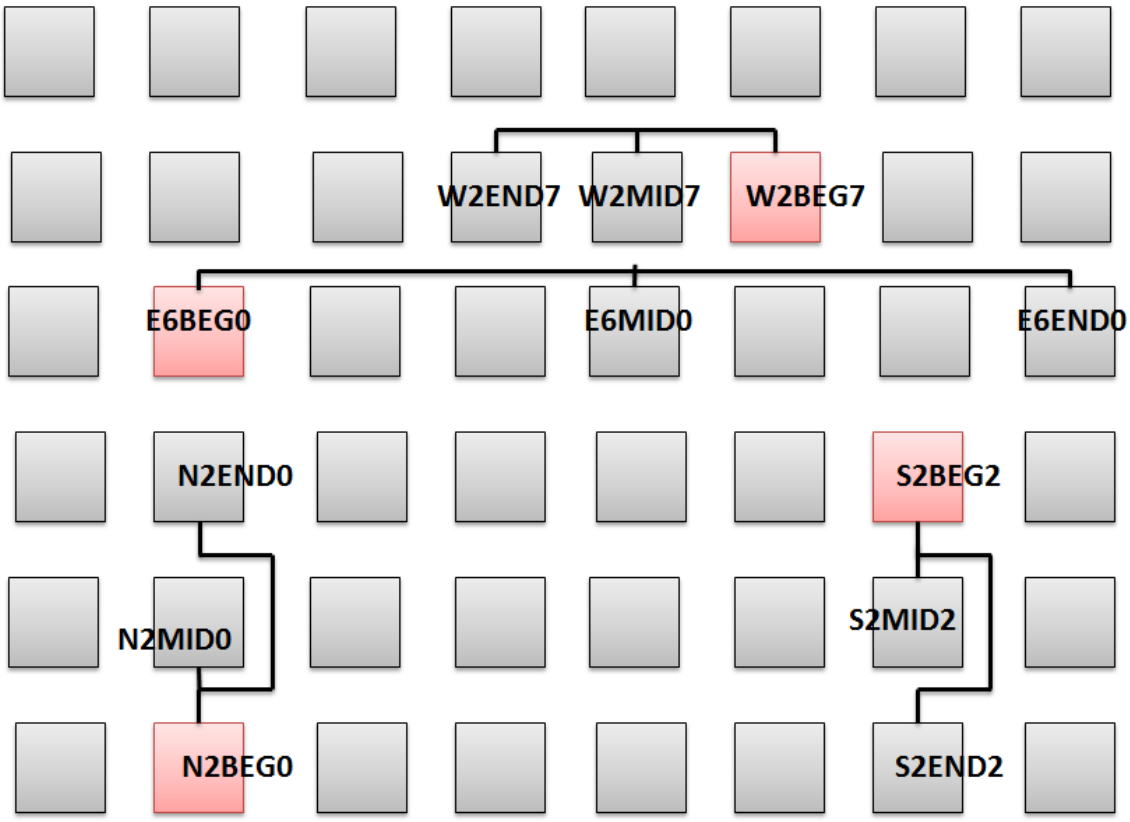


Figure 2.6 Virtex-4 Routing Resources.

2.2 Standard FPGA Design Flow

A typical FPGA design starts with the design being written in high level Hardware Descriptive Languages (HDL) such as VHDL, Verilog, or SystemC. The HDL is then synthesized into a technology independent netlist. The synthesized circuitry must then be mapped to the logic resources available on the target architecture. It must then be placed in the available resources and then routed to make the necessary connections between the logic resources. The FPGA Design flow can be seen in Figure 2.6

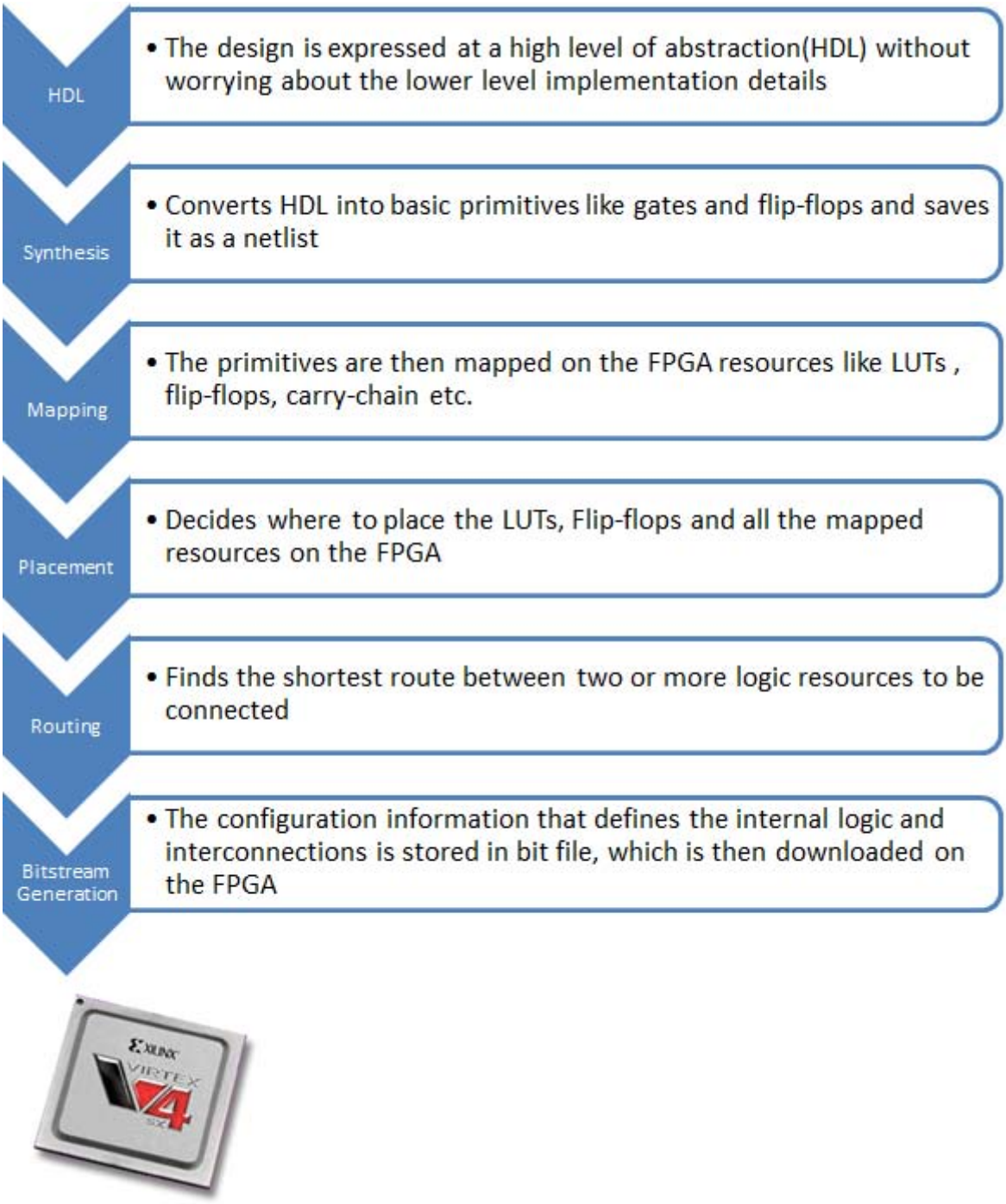


Figure 2.7 Standard FPGA Design flow.

2.3 RapidSmith

The Brigham Young University (BYU) RapidSmith [1] project provides a library of Application Programming Interfaces (APIs) for low-level manipulation of partially or completely placed-and-routed FPGA designs. RapidSmith provides useful Java-based APIs for design modifications such as placement and routing. RapidSmith is based entirely on the Xilinx Design Language (XDL). Using RapidSmith, designers can import XDL/NCD, manipulate, place, route and export designs back to XDL. Designers can make a variety of possible design transformations using the RapidSmith framework.

RapidSmith is organized into several packages as shown in Figure 2.7 (figure from [1]).

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.
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, ...).
router	This contains classes to route designs.
util	This contains miscellaneous support classes and utilities.

Figure 2.8 Packages in RapidSmith (figure from [1]).

2.4 Xilinx Design Language (XDL)

Since this thesis revolves around Xilinx FPGAs, it would be useful to understand XDL in detail. XDL is a human-readable version of Xilinx's Netlist Circuit Description (NCD) format. Documentation about the syntax of the XDL file is included within the XDL file itself.

The XDL file contains four sections that are sufficient to represent the complete design. The four sections contain module information, design information, instance information, and net information.

Module Information: This section contains information related to the module such as the name of the module, port names, instance names, port pins, and nets within the module. In the RapidSmith framework, this information is stored into Module class within the Design class.

Design Information: This section contains information related to the design such as the design name, part name, speed of the device, and ncd version used to create the design. The design information is stored in the Design class within RapidSmith as shown in Figure 2.8 [1]

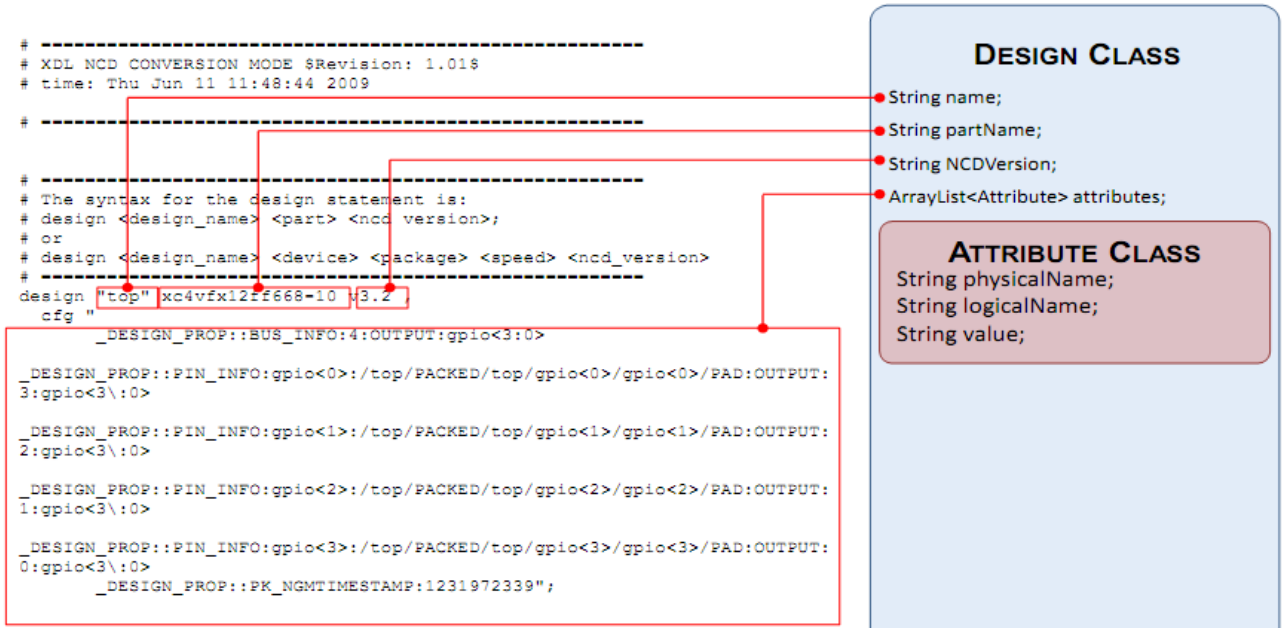


Figure 2.9 Design Class in RapidSmith (figure from [1]).

Instance Information: This section contains information related to the Instance such as the name of the instance, site definition, location where the instance is placed, and configuration string. The instance information is stored in the Instance class within Design

class in RapidSmith as shown in Figure 2.9 [1]

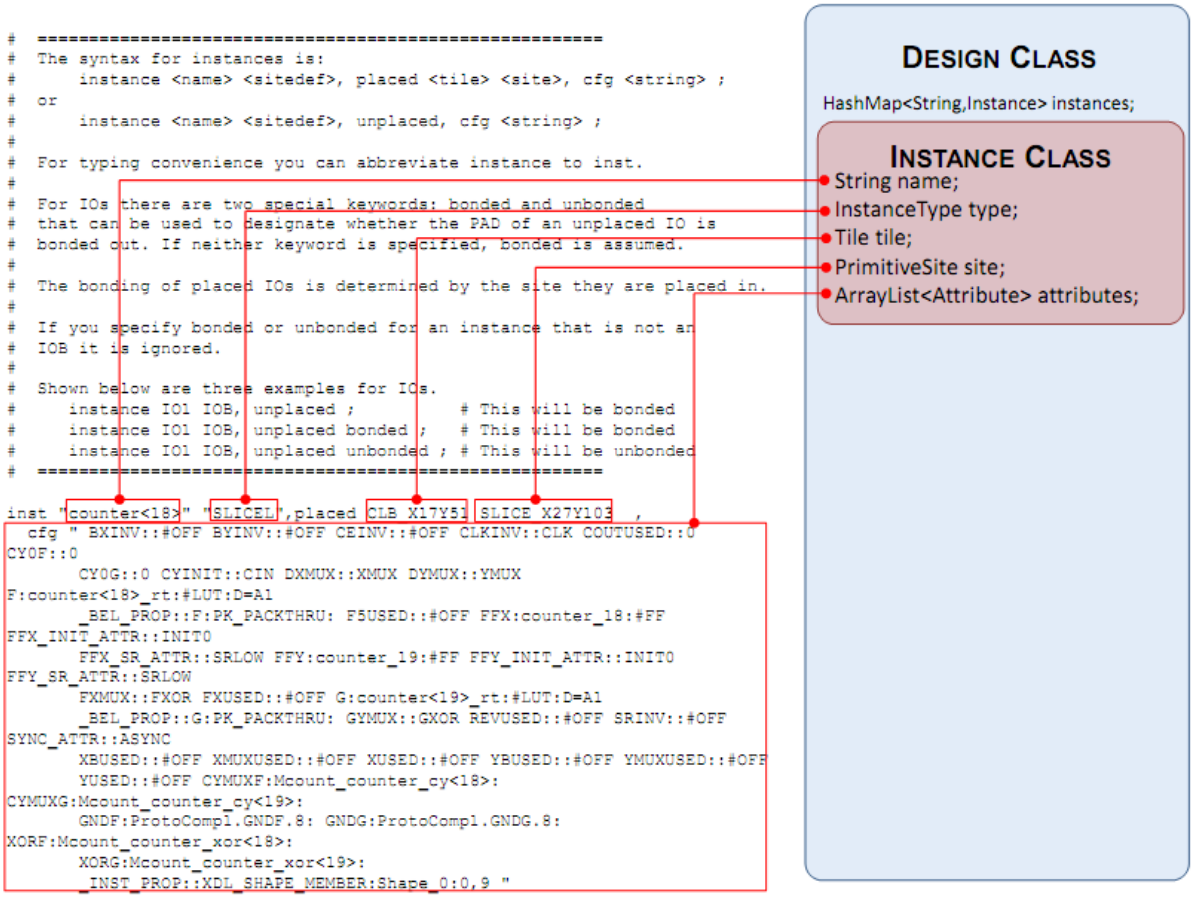


Figure 2.10 Instance Class in RapidSmith (figure from [1]).

Net Information: This section contains information related to the net such as the name, type, inpin(s), outpin(s), and PIPs that are turned on to connect the inpin(s) to outpin(s). The net information is stored in the Net class within the Design class in RapidSmith as shown in Figure 2.10 [1]

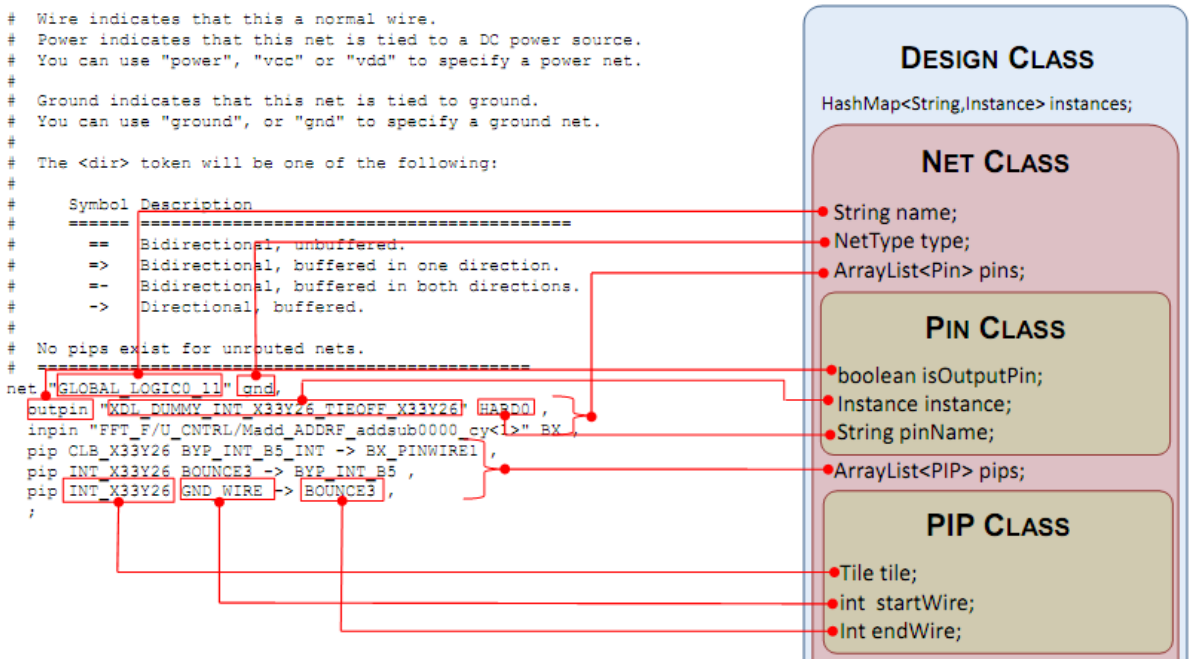


Figure 2.11 Net Class in RapidSmith (figure from [1]).

XDL can be used to represent designs that are unplaced/partially placed/placed and/or unrouted/partially routed/routed and/or contain hard macro definition and/or its instances. Figure 2.12 shows how XDL is used within Xilinx toolflow. The executable *xdl* can convert designs from NCD to XDL and vice versa. The conversion of XDL to NCD might fail if the XDL has errors such as unrouted nets, islands, or antennas. However, if the user still wants to convert the faulty XDL to NCD, *-force* flag can be used to force xdl executable to produce NCD even if there is an error in XDL. This information is useful for debugging purposes.

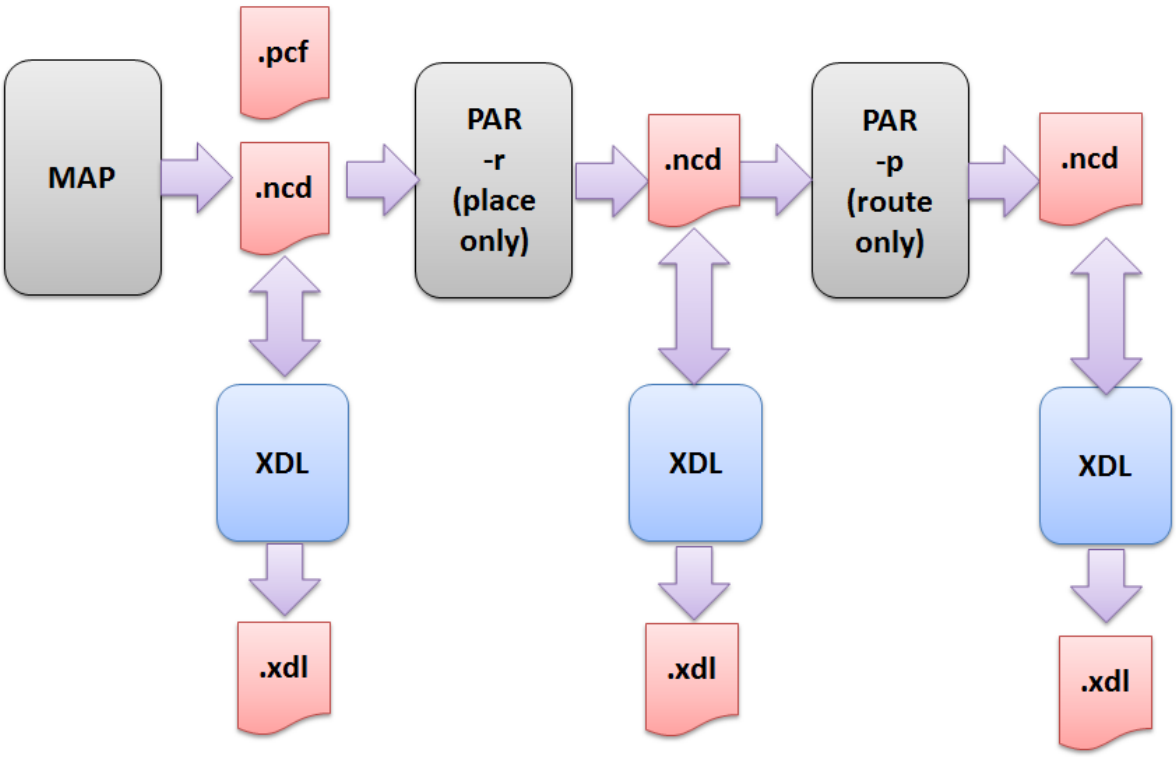


Figure 2.12 XDL in Xilinx toolflow (figure from [1]).

All the design modifications in the RapidSmith framework are done at XDL level. Importing XDL designs, writing modified designs, making design modifications such as adding/deleting modules and/or module instances, adding/deleting/modifying nets, and module relocation are all done at XDL level. Before using RapidSmith, it is highly advisable to get acquainted with XDL.

2.4.1 XDLRC Files

Xilinx XDL executable also contains a flag to generate report files about a particular Xilinx part. The generated reports are called XDLRC files (with extension xdlrc). These files contain all the descriptive information about Xilinx devices such as routing resources, tile layout, and primitive sites. While installing RapidSmith, XDLRC files are generated for

each device in a particular family such as Virtex-4. The XDLRC files are huge (generally in Gigabytes). RapidSmith parses the XDLRC files and saves them into much smaller device files, which contain all the useful information required by the mapper, placer, and router. [1].

xdl -report -pips v50pq240-5 v50pq240-5.xdlrc

This command produces v50pq240-5.xdlrc that contains all the PIP information for all the CLBs in the device v50pq240-5.

The XDLRC file provides the information about all the routing resources in a given device. This information can be used by the router to find out the routes between two given points on an FPGA. Both, the auto router and the hand-router use the routing information derived from the XDLRC files to produce valid routes at XDL level. The XDLRC file also contains information about the device architecture, which is used by the placer.

2.5 Similar Work

VPR (Versatile Place and Route) [8] has been an open-source FPGA research tool for several years. VPR performs no architecture-specific placement or routing, but deals only with abstract architectures. Our understanding is that VPR is currently limited to FPGAs, which can be described using VPR's architectural description facilities. We considered using VPR for the purpose of this project, but the idea was quickly dismissed because of limited support for commercially available Xilinx FPGAs.

Xilinx's JBits [15] provided APIs to manipulate Xilinx FPGA bitstream directly. This provided the capability of modifying circuits in the Virtex-II architecture. Built on JBits, JRoute [5] provided APIs to access the routing resources in Xilinx FPGA architecture. JBits and JRoute used Xilinx proprietary information. They are no longer supported for current

Virtex architectures. Figure 2.13 (figure from [15]) shows how design modifications could be made directly to the executable (.bit file) using the JBits framework.

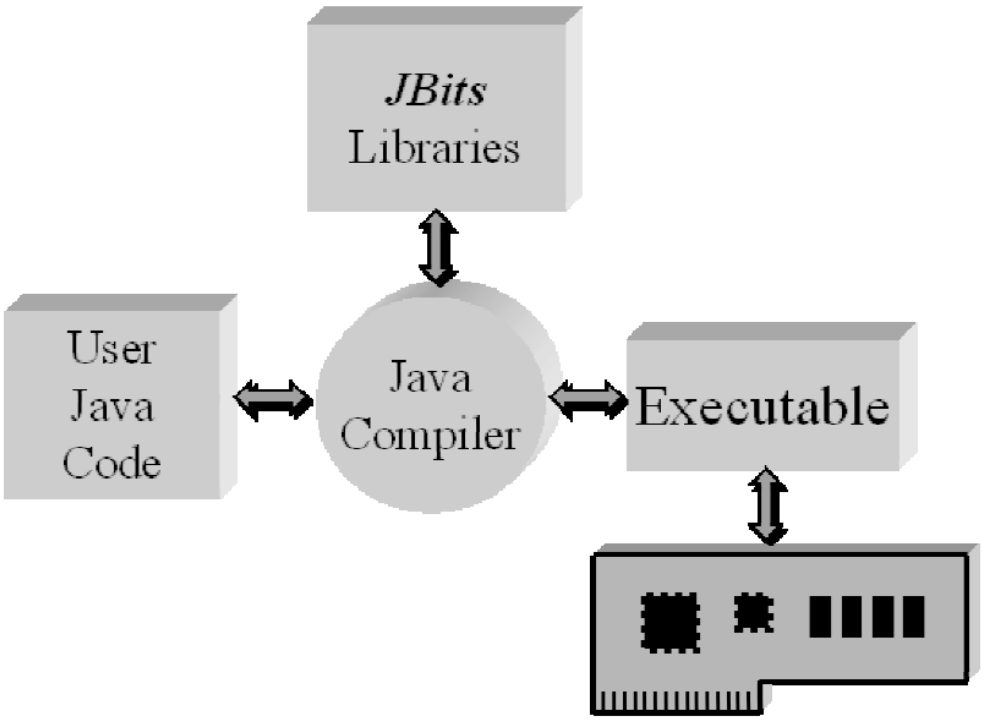


Figure 2.13 The JBits design flow (figure from [15]).

2.6 Previous Work

The mapper, placer, and router used in this thesis were originally developed for Autonomous Adaptive Systems (AAS) [13]. Autonomous Adaptive Systems was inspired from Autonomous Computing Systems (ACS) [2]. In both of these systems, depending upon external stimuli, the FPGA based system could autonomously adapt to the stimuli and change its configuration. Linux was ported to one of the PowerPC cores on a Virtex-4 device. The FPGA had tools such as the mapper, placer, and router running on Linux in an embedded environment. Based upon the stimuli, the FPGA could generate a new bitstream for itself.

The Autonomous Computing Systems project used JBits [15] internally, which used Xilinx proprietary information. The Autonomous Adaptive Systems project did the same things in an open-source environment.

Since the tools were required to run in an embedded environment, the goal for the tools, such as the mapper, placer, and router, was not to find the best option possible, but to find a decent solution quickly using very small amount of memory. The algorithms used to develop these tools were chosen to be memory efficient.

This thesis has integrated the tools, such as mapper and placer, from the AAS project. The router used in AAS was ported over from C++ to Java to make it compatible with the RapidSmith framework. The algorithm for the router remains the same for both AAS and the RapidSmith framework.

Chapter 3

System Overview

The toolkit, presented in this thesis, is provided to make modifications to an already existing design. The toolkit, as shown in Figure 3.1, contains tools such as the EDIF parser [9], mapper, placer, hand-placer, router, and hand-router. This chapter presents two separate flows in which the tools can be used. A detailed description about each tool in the toolflow, along with its interface is presented in this chapter.

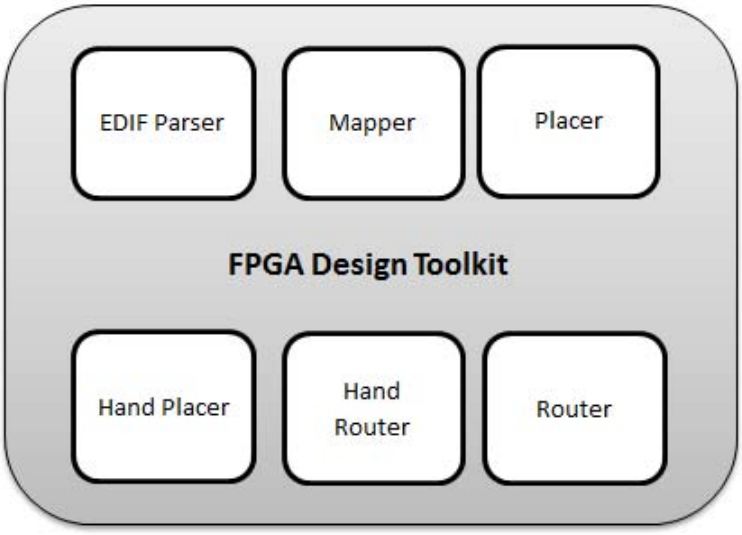


Figure 3.1 RapidSmith based Toolkit.

The toolkit offers two kinds of flows: one to add modules to an already existing design and the other one to make customized or controlled modifications to a design.

1. Module addition toolflow: To add new modules into the design, the EDIF netlist has to go through the EDIF parser, mapper, placer, and router, as shown in Figure 3.2. The EDIF netlist will be given as an input to the EDIF parser. The output of the EDIF parser will be used by the mapper and the placer. The mapper will map the module and implement the functionality of the module using the LUTs and flip-flops. The placer will place the LUTs and flip-flops in appropriate locations on the FPGA. Connections within the module will be made by the router using the output produced from the mapper. The output of the router is saved into the design. The module is now added into the design. The mapping, placement, and routing are all done at the XDL level. In this case, the user does not have control over the results produced by the tools. The user cannot constrain the tools to place and route the module at a particular location on the FPGA.

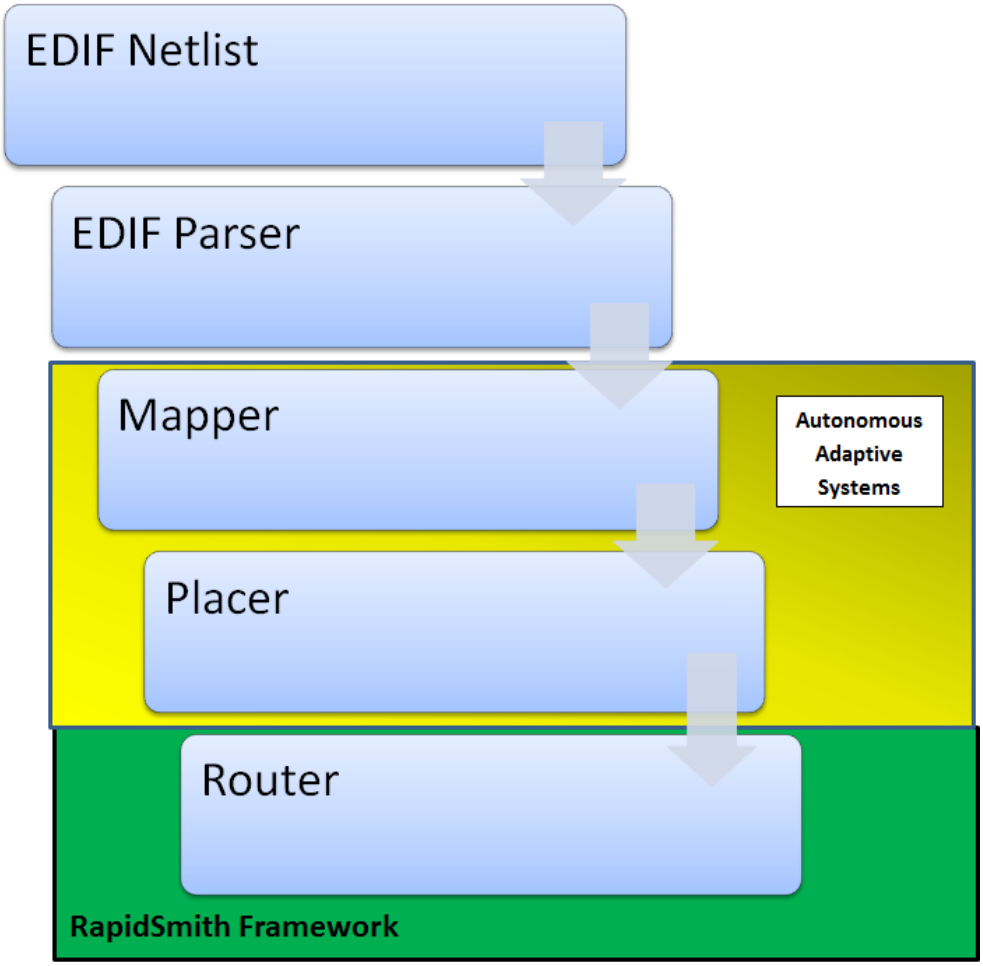


Figure 3.2 Toolflow for module addition.

2. Customized toolflow: Customized or controlled modifications can be made to a design using the hand-placer, hand-router, and/or router APIs. For example, if the output from the LUT has to be inverted in a design, the inverter can be placed manually into the design using the hand-placer. Since the net has to pass through the inverter, the route for the net will change. The hand-router can be used to route the net through the newly placed inverter. In this case, the designers get to place and route their modifications exactly how they want them to be. However, this approach can be tedious.

Figure 3.3 shows the toolflow for customized design modifications. A comparison between

Figure 3.3 and Figure 2.13 illustrates the similarities and differences between the toolkit and JBits. The JBits framework can modify the bitstream directly using the proprietary Xilinx information. Using the RapidSmith framework, the design modifications can be made only at the XDL level. The *ncd2xdl* conversion is used to convert the NCD format into XDL format, so that RapidSmith can use it. Based upon user modifications, the XDL file is modified and then converted back to NCD format using the *xdl2ncd* conversion. Xilinx bitstream generation tool can then be invoked to produce the bitstream for the modified design.

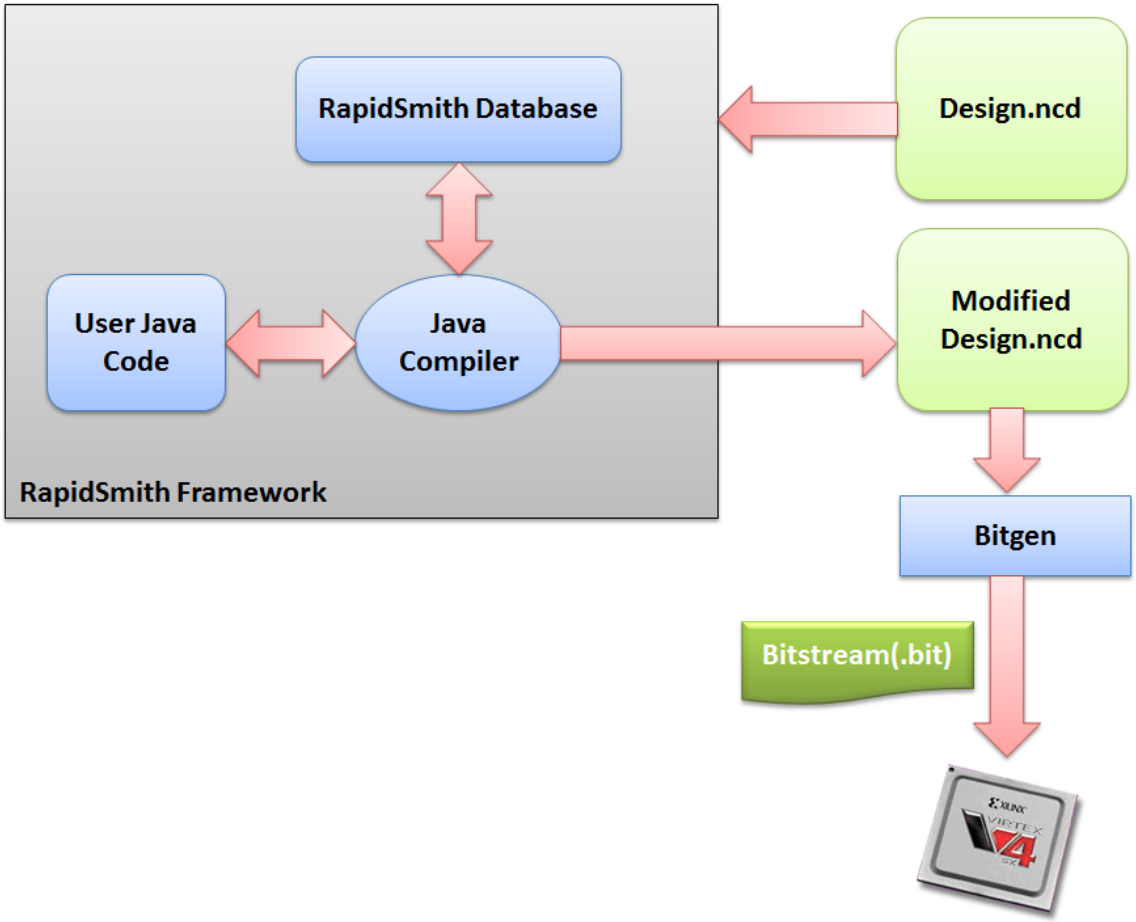


Figure 3.3 Toolflow for customized design modifications.

This chapter describes each tool in the toolkit along with its input(s) and output(s). The discussion in this chapter follows the toolflow described in Figure 3.2. It starts with the

discussion about the EDIF parser, then discusses the mapper and placer, and ends with detailed description about the router. Subsections on the hand-placer and hand-router are also presented within the placer and router sections, respectively.

3.1 EDIF Parser

The Configurable Computing Lab at Brigham Young University (BYU) provides two separate EDIF processing tools [9]. A complete and robust tool supports reliability studies for FPGAs exposed to radiation. An alternate and lightweight tool includes code and grammar that is suitable for the parser generation.

The lightweight version of the EDIF parser was used in AAS [13] project, taking into account that it was a time and memory constrained standalone embedded application. The output generated by the EDIF parser is a flattened netlist, i.e. the output will not be hierarchical in nature. However, if needed, the full blown EDIF parser can be easily integrated by the user. The EDIF parser, used in the toolkit, is borrowed directly from the AAS project.

The EDIF parser, as shown in Figure 3.4, parses the EDIF netlist and generates a data-structure for the mapper and placer to utilize. Information about the netlist such as the cells, cell connections, cell properties, ports of each cell, and nets, is extracted and stored in internal data structure.

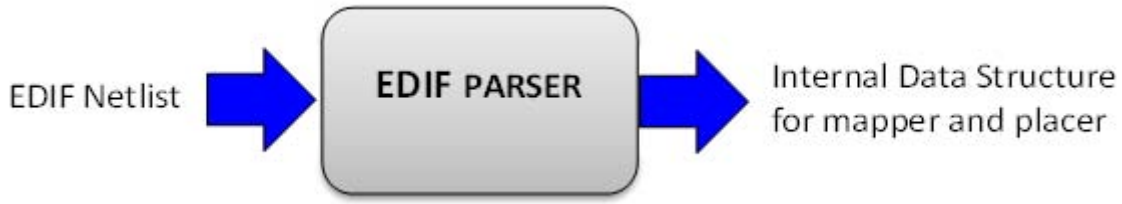


Figure 3.4 EDIF Parser.

3.2 Mapper and Placer

The mapper and placer reads in the information stored in the data-structure by the EDIF parser and produces a netlist for the router, as shown in Figure 3.5. Using the information from the EDIF parser, the mapper and placer build a circuit graph, where each node is a cell and each edge is a net connecting two cells. The EDIF netlist is assumed to be synthesized targeting a particular Xilinx architecture. As a result, each cell in the EDIF netlist is an instance of basic elements defined in the Xilinx Unified Library [16]. Currently, only the following subset of the library is supported: all kinds of LUTs, all kinds of flip-flops, multiplexers, and XOR gates. This is sufficient only for simple designs. Thus, it is recommended to use the mapper and placer only for adding simple modules to an existing design.

Figure 3.5 Mapper and placer.

Out of the two outputs from the mapper and placer, we only make use of the placed netlist and ignore the configuration primitives for ICAP and bitgen. The placed netlist from the mapper and placer acts like an input to the router. The router input file has the format as shown in Figure 3.6.

Number of nets					
3					
Source_clb_x	source_clb_y	source_net_name	dest_clb_x	dest_clb_y	dest_net_name
20	22	BEST_LOGIC_OUT_S0	26	30	IMUX_B7
--	--	--	--	--	--
--	--	--	--	--	--

Figure 3.6 Router input file.

The placer was originally developed to demonstrate proof-of-concept for the AAS project. Considering the time and memory constraints in the AAS project, a greedy algorithm was used to implement the placer [13]. The performance of such a placer is not optimal; however, the goal is not to develop an optimal placer. Any placement that can be routed by the router is considered an effective placement. The mapper and placer have been merged with this thesis to give a complete flow to the proposed toolkit.

3.3 Hand-Placer

The results from the placer are not very flexible. The user does not get control over where the new module will be placed. In situations where the designer wants a particular instance to be placed at a particular location on the FPGA, a hand-placer can be used. Figure 3.7 shows how easy it is to hand-place a design in RapidSmith. The RapidSmith framework provides a hand-placer so that the designer can write a program in RapidSmith to place a module instance anywhere on the FPGA. The hand-placer can be a handy tool for customized design requirements. However, hand-placing a large module can be tedious.

```
// 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
design.setPartName("xc4vfx12ff668-10");

// Create a new instance, set its name and its type
Instance myInstance = new Instance();
myInstance.setName("Bob");
myInstance.setType(PrimitiveType.SLICEL);

// Make the F LUT a 2-input adder
myInstance.addAttribute(new Attribute("F","LUT_of_Bob","#LUT:D=A1+A2"));

// Let's get the site information for slice X24Y20
PrimitiveSite site = design.getDevice().getPrimitiveSite("SLICE_X24_Y20");

//Now place the instance at the site
myInstance.place(site);

// We need to add the instance to the design so it knows about it
design.addInstance(myInstance);

//Now lets save the changes that we made into an helloworld.xdl
String fileName = design.getName() +".xdl";
design.saveXDLFile(fileName, true);
```

Figure 3.7 Hand-placer.

3.4 Router

The router is written in the RapidSmith framework using Java. As shown in Figure 3.8, the router takes in the output from the mapper and placer, and generates a list of PIPs to be turned on. The details about the device architecture and routing resources for a particular Xilinx device are derived from the XDLRC files by RapidSmith. Router uses the routing architecture to find out the routes for a given net. The router has been programmed to be architecture independent, i.e. it can be used for any Xilinx architecture. Thus, as long as the routing architecture is provided by RapidSmith, the router will find a route between any

two given points.

Since the XDLRC files do not provide timing information, the router cannot be a timing-driven router. For the purpose of this work, it is assumed that the delay increases with the number of PIPs to be turned on between two points. Thus, the goal of the router is to find a route with a minimum number of PIPs to be turned on.

Figure 3.8 Router.

The router was ported over from the AAS project. Because AAS was an embedded system, the memory and timing requirements were very stringent. Thus, the goal of the router was to find an effective route rather than finding the optimal route between the two points.

With all the routing resources available on an FPGA, finding a path between two points can be viewed as a graph search problem. Because of the memory and time limitation, a variant of A* algorithm was used to implement the router, as shown in Figure 3.9.

```
function A*(start,goal)
     closedHashMap := the empty HashMap
     openHashMap := HashMap containing the initial node
     f_score[start] := dist_between (start, goal)

     while openHashMap is not empty
         x := the node in openHashMap having the lowest f_score[] value
         remove x from openHashMap
         add x to closedHashMap

         if x = goal
             return reconstruct_path()

         foreach y in neighbor_nodes(x)
             if y in closedHashMap
                 continue
             f_score[y] := f_score[x] + dist_between(x,y)

             if y not in openHashMap
               if f_score[y] < f_score[x]
                 add y to openHashMap

     return failure
end
```

Figure 3.9 A* Algorithm for the router.

The fact that only the neighbors with the *f_score* less than the current *f_score* are inserted into the *openHashMap*, reduces the search space. This definitely reduces the memory requirement of the system and may reduce the runtime of the router as well.

3.4.1 Router API

The router API is very intuitive and user friendly with methods such as *route* and *unroute*. The method *isOn* can be used to ensure that a PIP has not been used for different nets. The router APIs provide auto-routing methods, which involve routing from single source to single sink and from single source to multiple sinks. The APIs are discussed in detail in the

following subsections.

1. route (source, sink)

This single source to single sink call allows for auto-routing of point to point connections. The source and the sink are defined by a row, column, and wire name. This method uses A* algorithm as described in Figure 3.9. This method returns a list of PIPs to be turned on for the source to be logically connected to the sink.

The following example shows how to create a source and sink, and how to use the router API to connect the source to the sink.

```
NetEndPoint source = new (5, 11, BEST_LOGIC_OUT_S7);
NetEndPoint sink = new (3, 13, IMUX_B11);
Router.route(source, sink);
```

The following example shows the output from the router, which contains the PIPs to be turned on to make a logical connection from the source to sink.

```
int_X3Y13 BYP_BOUNCE3 ->IMUX_B11
int_X3Y13 BYP_INT_B3 -> BYP_BOUNCE3
int_X3Y13 W2BEG0 -> BYP_INT_B3
int_X3Y13 W2END0 -> W2BEG0
int_X5Y13 W2BEG0 -> W2END0
int_X5Y13 N2MID0 -> W2BEG0
int_X5Y12 N2BEG0 -> N2MID0
int_X5Y12 OMUX_N13 -> N2BEG0
int_X5Y11 OMUX13 -> OMUX_N13
```

```
int_X5Y11 BEST_LOGIC_OUTS7 -> OMUX13
```

2. route(NetEndpoint source, NetEndpoint [ ] sink)

This single source to multiple sink call allows for auto-routing of one point to multiple connections. This function call internally uses single source to sink auto-route method. This method will first route from source to the first sink. To route other sinks, it will determine the closest point to the sink and then call the single source to sink auto-route method. This method also returns a list of PIPs to be turned on.

3. unroute(netName)

This method takes in the net name and unroutes it from the design. This method deletes all the PIPs associated with this net.

4. isOn(clb_x, clb_y, PipStart, PipEnd)

This method has been added to avoid contention. If a PIP is already being used by some other net, it cannot be re-used. Thus, it is essential to make sure that the PIP that is being considered is not already used by some other net in the design. This method returns true if the PIP is being used by some other net or else it returns false.

3.5 Hand-Router

The designer does not have control over the route that the auto-route chooses for a particular net. However, in some situations, the designers would want to control the way in which the

net is connected. For example, the user might want the net to pass through a particular CLB or the design requires the net to have maximum delay possible. The hand-router can be a handy tool in such situations. The hand-router starts from the source and provides the designer with all the possible options it has. The designer chooses the options that best suits the design requirements. Once the route is complete, all the PIPs that were selected by the designer will be added to the net.

Chapter 4

Experiments and Results

The aim of this chapter is to share the results of key experiments done with the toolkit. The main goal of this thesis is to address the problems related to the productivity of FPGA designers. Thus, the most important metric to quantify the effectiveness of the toolkit is to show how much time it saves as compared to the normal FPGA design flow. Time-savings obtained using the toolkit depends a lot upon the design size, and the number and size of design modifications. It is essential to compare the results of the toolkit with the mainstream tools. It is also essential to evaluate the toolkit and find out where it stands with respect to previously available tools and similar tools.

In this chapter, two experiments are discussed in detail. Experiment 1 involves relocating a module from one part of the FPGA to another part of the FPGA. This is a small design containing only 10 slice-LUTs. The step-by-step procedure to relocate a module using the toolkit is discussed. Finally, time comparisons are made between the standard Xilinx FPGA design flow and the toolkit. Experiment 2 involves calculating 16 1024-point Fast Fourier Transform (FFT) in a Single Instruction, Multiple Data (SIMD) architecture. This is a huge design, which involves 87364 slice-LUTs. Small design modifications are made to this

design and time comparisons are made. With the help of these experiments, some basic design modifications, such as module instance addition/deletion and net addition/deletion, can also be demonstrated.

4.1 Experiment 1: Module Relocation

In this experiment, a module will be relocated from one part of the FPGA to another part of the FPGA. For example, an adder module instance, *inst1*, is currently at CLB location X20Y20. The task is to relocate this module instance to another CLB location X24Y20. This would involve the following steps:

1. Module Instance Deletion: The current module instance, *inst1*, has to be deleted from CLB location X20Y20.

2. Net Deletion: Since the module instance does not exist, all the nets connecting the module to the design must be deleted.

3. Module Instance Addition: A new instance, *inst2*, has to be added at CLB location X24Y20.

4. Net Addition: To connect the new module to the original design, new nets must be added to the design.

Figure 4.1 shows the module at CLB location X20Y20. This is before the module is relocated. A net connecting the instance to other part of the design is shown as well.

Figure 4.1 Before module relocation.

Figure 4.2 shows the module at its new CLB location at X24Y20, after it has been relocated. The net connecting the new module instance to the rest of the design has been re-routed as well. The placer was used to place the new module to its new coordinates. The router was used to make the connections to the external circuitry from the new location.

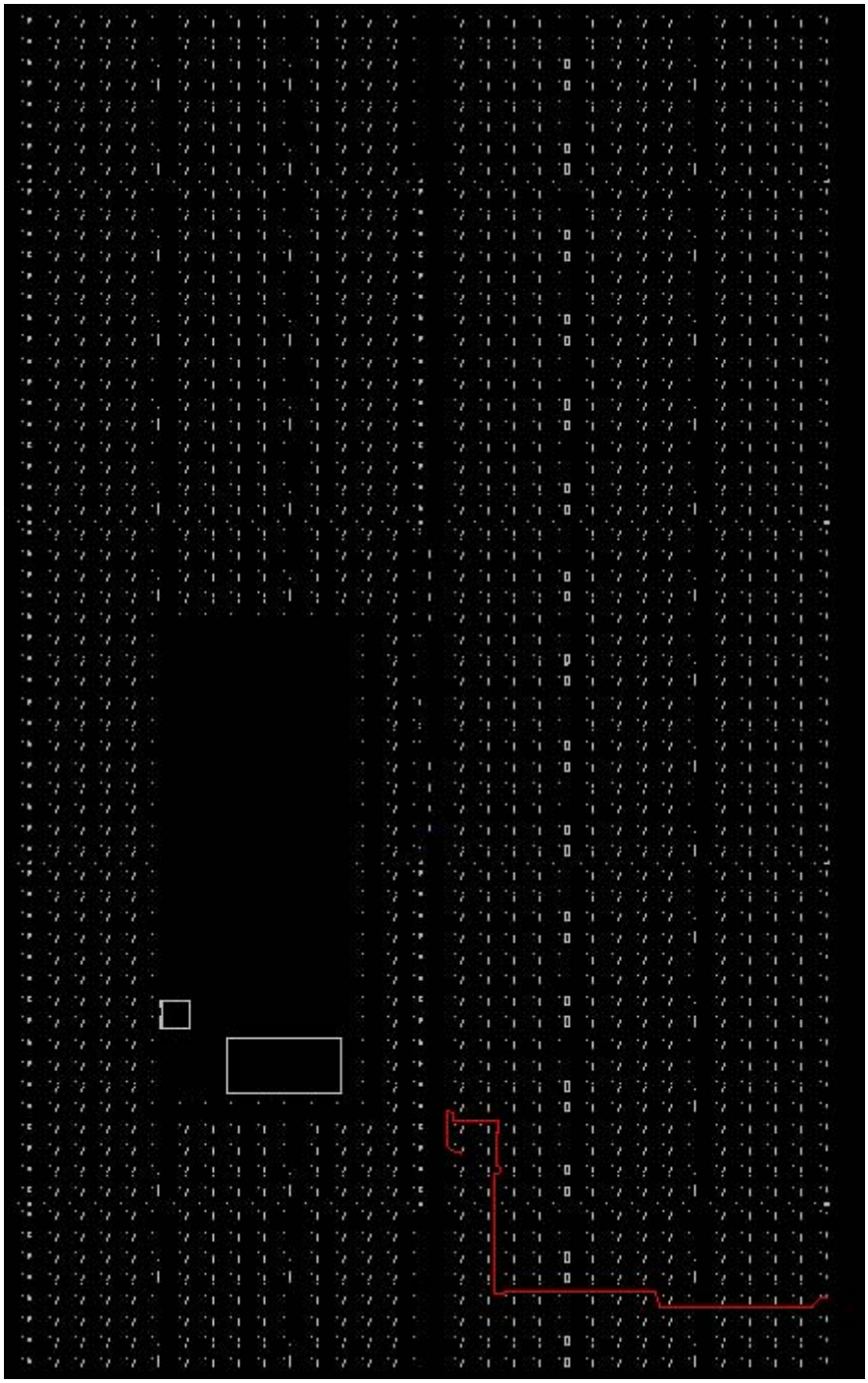


Figure 4.2 After module relocation.

Table 4.1 compares the results obtained by making design changes using the Xilinx ISE tool

versus the toolkit. The toolkit takes 5 seconds (22 percent time saved) less than Xilinx ISE to make the necessary design modifications. The Xilinx tools do not recognize the triviality of the design modification, and go through almost the entire Xilinx toolflow again. Since the designers know exactly what they want to do, the toolkit can be used to make the design modifications. Thus, designer knowledge is utilized to save design compilation time.

Table 4.1 Results: Module Relocation

	Original design time using ISE in seconds	**Modification time using ISE in seconds**	**Modification time using toolkit in seconds**
Synthesis	6	–	–
Translation	10	8	–
Mapping	13	7	–
Placement	13	10	1
Routing	2	2	1
ncd2xdl	–	–	7
xdl2ncd	–	–	9
Load Device Database	–	–	3
Total Time	**44**	**27**	**21**

A closer look at the table also exposes some of the issues with the toolkit. The actual time taken to place and route the design modifications was less than 2 seconds (less than 10 percent of the total toolkit runtime). This suggests that the overhead involved in this

process was more than the actual work (placement and routing) to be done. The time taken for *ncd2xdl* and *xdl2ncd* conversion, and to load the device database are considered as overheads. Amongst the overheads, the time for *xdl2ncd* conversion is the longest. In this case, the overheads take more than 90 percent of the total runtime of the toolkit.

The time-savings obtained for smaller designs would not be worth all the efforts put in to learn the RapidSmith framework and getting acquainted with the toolkit. Thus, it is not recommended to use the toolkit for small designs.

4.2 Experiment 2: Fast Fourier Transform (FFT)

This design is running 16 1024-point FFTs in a SIMD architecture. This design contains around 87364 slice-LUTs. The designer wanted to add inverters to some parts of the design. Table 4.2 shows the time it took to modify the design using Xilinx ISE tools against the toolkit. As it can be seen from the table, the toolkit is clearly a winner in this design. With ISE, it takes about 6 hours to make the design modifications, while with the toolkit it just takes 71 minutes (almost 80 percent time saved) saving 5 hours of valuable design/debug time. Again, the placement and routing using the toolkit do not take much time. The *xdl2ncd* conversion is taking up most of the time.

The two experiments presented above, showed a contrast. In the first experiment, the toolkit saved a mere 6 seconds (20 percent of the design time), while in the second experiment, the toolkit took 5 hours less time than ISE. The *xdl2ncd* conversion took the bulk of the implementation time for the toolkit. Using the toolkit for smaller designs with small design modifications would result in an unnecessary overhead. Hence, for such situations we recommend using the traditional Xilinx flow because it would be more time efficient. However, for bigger designs, it is recommended to use the Xilinx toolflow along with the toolkit.

Table 4.2 Results: Fast Fourier Transform

	Original time using ISE in minutes	**Modification time using ISE in minutes**	**Modification time using toolkit in minutes**
Synthesis	4	4	–
Translate	7	7	–
Mapping	178	172	–
Placement	172	166	1
Routing	32	23	1
ncd2xdl	–	–	9
xdl2ncd	–	–	55
Load Device Database	–	–	1
Total Time	**393**	**372**	**67**

When should the designers use the toolkit and when should they use ISE? The answer is for small designs, the overhead associated with the toolkit is more, and hence it is recommended to use ISE tools. However, for medium/large designs, where *xdl2ncd* conversion is very small as compared to the actual tool runtime, using the toolkit would be beneficial.

Table 4.3 shows the comparison of RapidSmith based toolkit with other tools such as VPR and JBits/JRoute. The productivity rating is lower as compared to JBits/JRoute because JBits/JRoute used Xilinx proprietary information to modify the bitstream directly. With RapidSmith, the modifications are made to a placed and routed netlist, one step before bitstream generation. In addition, there is an extra overhead of converting the NCD to XDL and back to NCD format. As it was seen from previous results, XDL to NCD conversion is a time consuming process. Thus, the productivity is less as compared to JBits/JRoute. JBits and JRoute are now obsolete. VPR would have been the best tool available for making design modifications if it could have targeted commercially available FPGAs.

Table 4.3 Comparison with other tools

	VPR	**JBits/JRoute**	**RapidSmith**
Xilinx FPGAs	No	Yes	Yes
Current Status	Stable	Obsolete	Work in Progress
Time Savings	–	Very good	Good
Open-source	Yes	No	Yes
Ease of Design	–	No	Yes
Support	Very good	Very good	Work in progress
Productivity	–	Good	Very good
Debugging	Very Good	Very Good	Work in progress

Perhaps the biggest drawback is that the toolkit requires that the user be very familiar with the FPGA architecture. The Xilinx device architectures have been completely documented in the databook, but most users have never had the need to know such details. Thus, like JBits, it is expected that the necessary understanding of the underlying architecture will be the greatest barrier to widespread acceptance of the toolkit.

Nonetheless, the framework provides details related to the Xilinx FPGA architecture and routing architecture in good data-structures and provides easy to use APIs to access those data-structures. The toolkit provides good APIs to make design modifications to Xilinx FPGA designs and save the FPGA compilation time and thus improve the productivity of FPGA designers. The RapidSmith framework is still under development. Many researchers and enthusiasts are working on this framework to make it a better tool for the FPGA designers.

Chapter 5

Future Work

The work presented in this thesis demonstrates a proof-of-concept as to how open-source tools will help improve the productivity for FPGA designers. As with any research, there exist multiple avenues for expanding the breadth and depth of this work. Along the lines of expanding the depth of this work, the Quality of Result (QOR) should be the focus. The breadth of this work can be expanded by adding extensions to the tool.

5.1 Quality of Result (QOR)

The tools built in current work had a different objective, and as a result, the results produced by the tools were not optimal. Therefore, providing good QOR should definitely be one of the ways to proceed in the future. The QOR of the toolkit will increase if efforts are made to optimize the placer and the router in the toolkit.

1. Placer improvements: Simulated annealing algorithm or its variant should be used for the placer for better QOR [6] [7].

2. Router improvements: Single source to multiple sink nets are not handled very efficiently in this toolkit. Efforts should be made to reuse the routing resources as much as possible. New features such as guided routing and template routing should be included.

5.2 Extensions

This thesis focuses only on Virtex-4 devices. Extensions to other Xilinx devices and Altera FPGAs should be provided as well. This will make the toolkit available to a broad range of designers and increase their productivity.

1. Mapper and placer extensions: Currently, only a subset of the Xilinx Unified Library for Virtex-4 device is supported by the mapper. Efforts should be made to support all the elements of all the families in Xilinx Unified Library.

2. EDIF Parser: Efforts should be made to integrate the full blown EDIF Parser along with the toolkit. It gives unflattened netlist as output. This would help to preserve the modularity of design in RapidSmith.

3. Altera FPGAs: Currently, this work concentrates only on Xilinx FPGAs. The RapidSmith framework can easily be extended to support Altera FPGAs as well. Altera, through its QUIP program [18], provides information about its tools and architecture details. This information can be utilized to create a database, which can be used by the mapper, placer, and router.

Chapter 6

Conclusion

The objective of this research is to improve productivity of FPGA designers by reducing the implementation tool runtime for modifications made to an already existing design. If modifications are made to a particular module in a design, the tools re-run for almost the entire design. This increases the overall tool runtime. The question that is being addressed in this thesis is how can the implementation tool runtime be minimized for design modifications made to an already existing design? This thesis presents an open-source toolkit under the belief that the implementation tool runtime can be significantly reduced if the designers are given the freedom to make changes to their designs at a lower level of abstraction. The toolkit presented in this work contains all the tools necessary for design modification such as the EDIF parser, mapper, placer, hand-placer, router, and hand-router. The toolkit has easy to use APIs so that it makes it easier for the designers to make low level modifications to their designs.

One important conclusion that can be drawn based upon the experiments is that the time-savings depend heavily upon the design size, and the size and number of design modifications. Using the toolkit is strongly recommended to make small design modifications for medium

to large design sizes. The *xdl2ncd* conversion adds a lot of overhead and forms the majority of the toolkit execution time. Thus, using the toolkit is recommended only in the situations where the *xdl2ncd* conversion is less than the design implementation time. The toolkit presented in this thesis is by no means a replacement to the existing Xilinx toolflow; however, it should be used as an auxiliary tool along with the Xilinx tools to enhance productivity.

The biggest barrier to widespread acceptance of the toolkit is the necessary understanding of the underlying architecture. In addition, the debugging issues with the toolkit would be painful and require expert level understanding about the architecture and tools. In spite of these limitations, the toolkit indicates that it is a powerful and a useful tool for making design modifications to Xilinx FPGAs. The toolkit presented in this thesis definitely addresses productivity issues of FPGA designers, and it should definitely be used for rapid FPGA system deployment.

Bibliography

[1] RapidSmith — Download RapidSmith software for free at SourceForge.net. [Online]. Available: http://sourceforge.net/projects/rapidsmith/. [Accessed: 19-Oct-2010].

[2] Neil Joseph Steiner, Autonomous Computing Systems, 30-Apr-2008. [Online]. Available: http://scholar.lib.vt.edu/theses/available/etd-04102008-194601/. [Accessed: 19-Oct-2010].

[3] FPGA and CPLD Solutions from Xilinx, Inc.. [Online]. Available: http://www.xilinx.com/. [Accessed: 20-Oct-2010].

[4] Elgris / EDIF Implementation. [Online]. Available: http://www.elgris.com/content/edif_overview.html. [Accessed: 19-Oct-2010].

[5] E. Keller and Xilinx. Inc, JRoute: A Run-Time Routing API for FPGA Hardware, 2000.

[6] V. Betz and J. Rose, VPR: A New Packing, Placement And Routing Tool For FPGA Research, in Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications. Springer-Verlag London, UK, 1997, pp. 213222.

[7] C. Mulpuri and S. Hauck, Runtime And Quality Tradeoffs In FPGA Placement And Routing, in Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays. ACM New York, NY, USA, 2001, pp. 2936.

[8] J. Luu et al., VPR 5.0: FPGA cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling, in Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays, pp. 133-142, 2009.

[9] FPGA Reliability Studies - BYU EDIF Tools Home Page. [Online]. Available: http://reliability.ee.byu.edu/edif/. [Accessed: 20-Oct-2010].

[10] Incremental Design Reuse with Partitions [Online]. Available: http://www.xilinx.com/support/documentation/application notes/xapp918.pdf

[11] FPGA Editor User Guide [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp401.pdf

[12] Using SmartGuide [ISE Help]. [Online]. Available: http://www.xilinx.com/itp/xilinx9/help/iseguide/html/ise_using_smartguide.htm.

[13] Autonomous Adaptive Systems. [Online]. Available: http://www.ccm.ece.vt.edu/twiki/bin/view/Main/AASProject. [Accessed: 20-Oct-2010].

[14] ISE 12.1. [Online]. Available: http://www.xilinx.com/support/documentation/dt_ise12-1.htm. [Accessed: 21-Oct-2010].

[15] Xilinx Products: JBits SDK. [Online]. Available: http://www.xilinx.com/labs/projects/jbits/. [Accessed: 23-Oct-2010].

[16] Virtex-4 Libraries Guide for HDL Designs (ISE 10.1). [Online]. Available: http://www.xilinx.com/itp/xilinx10/books/docs/virtex4_hdl/virtex4_hdl.pdf

[17] FPGA Design Flow Workshop and Teaching Materials. [Online]. Available: http://www.xilinx.com/university/workshops/fpga-design-flow/index.htm. [Accessed: 24-Oct-2010].

[18] Quartus II University Interface Program. [Online]. Available: http://www.altera.com/education/univ/research/quip/unv-quip.html. [Accessed: 27-Oct-2010].

[19] Using Hard Macros to Reduce FPGA Compilation Time, in Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL'2010), August 2010.