A Management Paradigm for FPGA Design Flow Acceleration

Abhay Tavaragiri

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
Computer Engineering

Peter M. Athanas, Chair
Patrick R. Schaumont
Joseph G. Tront

July 7, 2011
Blacksburg, Virginia

Keywords: FPGA Management Technique, XML, Productivity, TORC

A Management Paradigm for FPGA Design Flow Acceleration

Abhay Tavaragiri

(ABSTRACT)

Advances in FPGA density and complexity have not been matched by a corresponding improvement in the performance of the implementation tools. Knowledge of incremental changes in a design can lead to fast turnaround times for implementing even large designs. A high-level overview of an incremental productivity flow, focusing on the back-end FPGA design is provided in this thesis. This thesis presents a management paradigm that is used to capture the design specific information in a format that is reusable across the entire design process. A C++ based internal data structure stores all the information, whereas XML is used to provide an external view of the design data. This work provides a vendor independent, universal format for representing the logical and physical information associated with FPGA designs.

Acknowledgements

I would, first of all, like to thank my academic advisor, Dr. Peter Athanas, for providing me the opportunity to work in the Configurable Computing Lab (CCM). He has been a source of inspiration to me throughout the duration of my research and it wouldn't have been possible for me to complete this work without his support and guidance. It has been a privilege and a tremendously rewarding experience to work with him.

I would like to thank Dr. Patrick Schaumont and Dr. Joseph Tront for serving as members of my committee. I also immensely enjoyed the courses that I took under their guidance.

I would like to express my gratitude to my family - my parents and my sister, for encouraging me to pursue higher studies. They have been a constant source of support throughout my graduate education.

I would also like to thank my lab mates - Jacob Couch, Andrew Love and Wenwei Zha for collaborating on this work with me. A special thanks to Tony Frangieh who patiently solved all my doubts and spent long hours with me debugging various issues. His inputs and feedback at different stages of this work were invaluable. Finally, thanks to all my CCM lab friends - Rohit, Umang, Mrudula, Sushrutha, Kavya and Karl who have made my time here truly memorable. I will look back with fond memories on the last two years.

Contents

Table of Contents **iv**

List of Figures **vii**

List of Tables **viii**

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 2

1.3 Organization 3

2 Background **4**

2.1 FPGA Design Flow 5

2.1.1 Traditional Flow 5

2.1.2 Incremental Flow 7

2.2 Related Work in Industry 12

2.2.1 NeoCAD 12

2.2.2 EDIF 13

2.2.3 OpenAccess 14

2.3 XML Based Schema 15

2.3.1 IP-XACT 16

2.4 TORC 17

3 qFlow **21**

3.1 The Problem Domain 22

3.2 Formulation 23

3.3 The Big Picture 25

3.4 The Conductor Functionality 28

3.4.1 Conductor-Consolidator Interface 28

3.4.2 Conductor-Architect Interface 29

3.4.3 Conductor-Librarian Interface 30

3.4.4 Conductor-Locator Interface 32

4 System Overview **33**

4.1 The FPGA Coordinate System 34

4.2 The XML Attributes 36

4.2.1 Name 36

4.2.2 Anchor 36

4.2.3 Ports 37

4.2.4 Boundary 39

4.2.5 Sub-Modules 41

4.2.6 Connections 42

4.2.7 Top-Level 43

4.3 Implementation 43

4.3.1 Overview 43

4.3.2 Boost and XML Support 44

4.3.3 Code Structure 45

4.3.4 Directory Organization 46

4.3.5 Application Development 49

4.3.6 Internal and External Data Structures 50

5 Results **51**

5.1 Test Designs 51
5.1.1 Embedded 52
5.1.2 High-Performance Computing 54
5.1.3 Software Defined Radio 55
5.2 Feature Comparison 56
5.2.1 Generic File Formats for ASICs 56
5.2.2 Internal Formats of Incremental Flows 60

6 Conclusion **63**
6.1 Future Work 64

Bibliography **65**

Appendix A : Description of the Interface Signals of the Embedded Application **69**

Appendix B : Description of the Interface Signals of the HPC Application **71**

List of Figures

2.1 Xilinx design flow 6
2.2 PATIS resource estimate file 10
2.3 VPR placement format 11
2.4 VPR netlist format 11
2.5 TORC block diagram 19

3.1 Consolidated qFlow model 22
3.2 qFlow run-times 25
3.3 qFlow components 26

4.1 Cartesian coordinate system 34
4.2 FPGA editor snapshot for the XC5VLX110T device 35
4.3 qBase class hierarchy structure 47
4.4 qFlow directory organization 48

5.1 Embedded application 53
5.2 High-Performance Computing application 54
5.3 Software Defined Radio application 55

List of Tables

2.1 Xilinx implementation times for two different designs 7

5.1 The initial set of test designs for qFlow 52
5.2 Feature comparison of qBase with IP-XACT and OpenAccess 57
5.3 Comparison of the internal file formats between Replace and qBase 60
5.4 Comparison of the internal file formats between PATIS and qBase 61

Chapter 1

Introduction

1.1 Motivation

Advances in semiconductor manufacturing technology have led to an increase in the size and complexity of modern Field Programmable Gate Arrays (FPGAs). FPGA implementation tools have not kept pace with this increase in density. As a consequence, modern FPGA designs take a long time to go through the entire implementation process. These long run-times are particularly undesirable during the design development phase. FPGA tool vendors provide incremental flows targeted at reducing compilation times; however, when working with complex designs even these flows require a complete re-implementation. This is analogous to rebuilding all the libraries of a software application even when a single file changes.

For example, a large FPGA design consisting of sixteen 1024-point Fast Fourier Transform (FFT) cores, targeted for the *convey-HC1* processor [30] takes around 4-5 hours for complete implementation. Now, even if this design undergoes a small change in logic, the entire process

is repeated and results in another 4-5 hours of compile time when most of the work done by the tools is a repetition from the previous iteration. Reducing the time taken for compiling incremental designs would result in a much more efficient process. A possible solution to this problem would involve implementing only the portion of logic that changed from the previous iteration. This would allow the designer to make small changes in his design and test it reasonably quickly even for complex designs.

The limitations of the existing implementation techniques motivate the creation of a new flow that reduces the time taken to compile big designs. The FPGA design cycle is a complex process with a lot of intermediate steps. A productivity flow typically focuses on optimizing a sub-set of these steps to provide accelerated compile times. The various tools and formats involved in the FPGA design process make the interaction between the different stages quite complex. An important aspect in this regard is the development of a management technique that can interlink the different processes. An efficient communication mechanism avoids the need for repeated file format conversions and results in an elegant incremental solution.

1.2 Contributions

The work presented in this thesis is part of a project that involves the development of a novel FPGA design flow targeted at improving productivity. This thesis presents a high-level overview of *qFlow*, a flow aimed at speeding up the back-end design stage of a typical FPGA design cycle by keeping track of the incremental design changes. The user can implement the front end part of the design using any standard technique and then invoke qFlow for the back-end process.

The focus of this thesis is on the creation of a management paradigm that is able to provide an efficient communication mechanism between the various stages in qFlow. Since this flow

focuses on the back-end part of the process, a unique approach is needed to capture the various physical parameters associated with an FPGA design. This thesis presents a data structure called *qBase* that has the required attributes to streamline the flow of design metadata and successfully execute qFlow.

The contributions of this thesis are summarized below.

- Motivate the need for a new incremental FPGA design flow
- Provide a high-level overview of qFlow
- Present a data structure, qBase, that helps the collaboration between the various components of qFlow
- Compare the features of qBase with existing design formats

1.3 Organization

This thesis is organized as follows: Chapter 2 presents the background information about the FPGA design cycle and the various efforts made by the industry and the research community towards improving FPGA productivity. It also describes the Tool for Open Reconfigurable Computing (TORC) framework [13] that is crucial to the design of qFlow. Chapter 3 provides a high-level description of qFlow including the motivation, the methodology used and the various high-level tasks involved. Chapter 4 describes the structure of qBase, the various attributes involved and the implementation details. The experiments and results associated with this work are discussed in Chapter 5. Chapter 6 presents the conclusions and discusses future work.

Chapter 2

Background

In this thesis, a flow management technique is presented that serves as a foundation for developing flows targeted at accelerating the FPGA design cycle with a focus on back-end physical design. The central idea behind this endeavor is to provide a standard format that allows users to capture and modify critical information associated with FPGA designs at any stage of the design flow. This information is encapsulated into qBase, a database constructed using C++ and externally represented using the Extensible Markup Language (XML). This provides an easy to use source of data that can be utilized for productivity flows independent of the FPGA vendor. Also, the technique is developed within the TORC framework, which is an open source FPGA tool set.

This chapter is divided into four sections to cover the background information needed to get a full understanding of this work. The first section provides a brief description of the standard FPGA design flow and enhancements made to it by FPGA vendors. It also details some of the academic research efforts towards incremental FPGA design flow. The second section discusses some of the previous work done in relation to developing standardized data models for FPGA or Application Specific Integrated Circuit (ASIC) design. The third

section describes the basic XML schema and some of the useful applications developed using it. The fourth and final section provides a brief introduction to the TORC framework that will help understand the use cases better.

2.1 FPGA Design Flow

FPGAs are integrated circuits that can be configured by the user based on specific applications. The flexibility offered by these devices as well as their low non-recurring engineering costs make them an attractive choice for prototyping, verification and development of digital systems; however, creating a design on FPGAs is a complex process involving a number of different steps and corresponding tools at every step. FPGA vendors provide a traditional flow as well as enhanced flows aimed at decreasing the design cycle time. Academic researchers have also extensively investigated incremental flows for improving productivity. The following sections describe these flows in detail.

2.1.1 Traditional Flow

This section describes the standard flow for implementing designs on an FPGA. It is based on the Xilinx development environment but is general enough to be applicable to devices from other FPGA vendors.

Figure 2.1 shows a flow diagram for traditional FPGA design flows. A typical FPGA design flow starts with a description of the design in a Hardware Description Language (Verilog/VHDL) followed by logic simulation to ensure correct functionality. The design is then synthesized into a technology independent netlist in terms of the generic logic elements (Look up Tables (LUTs), flip flops, logic gates) on the FPGA.

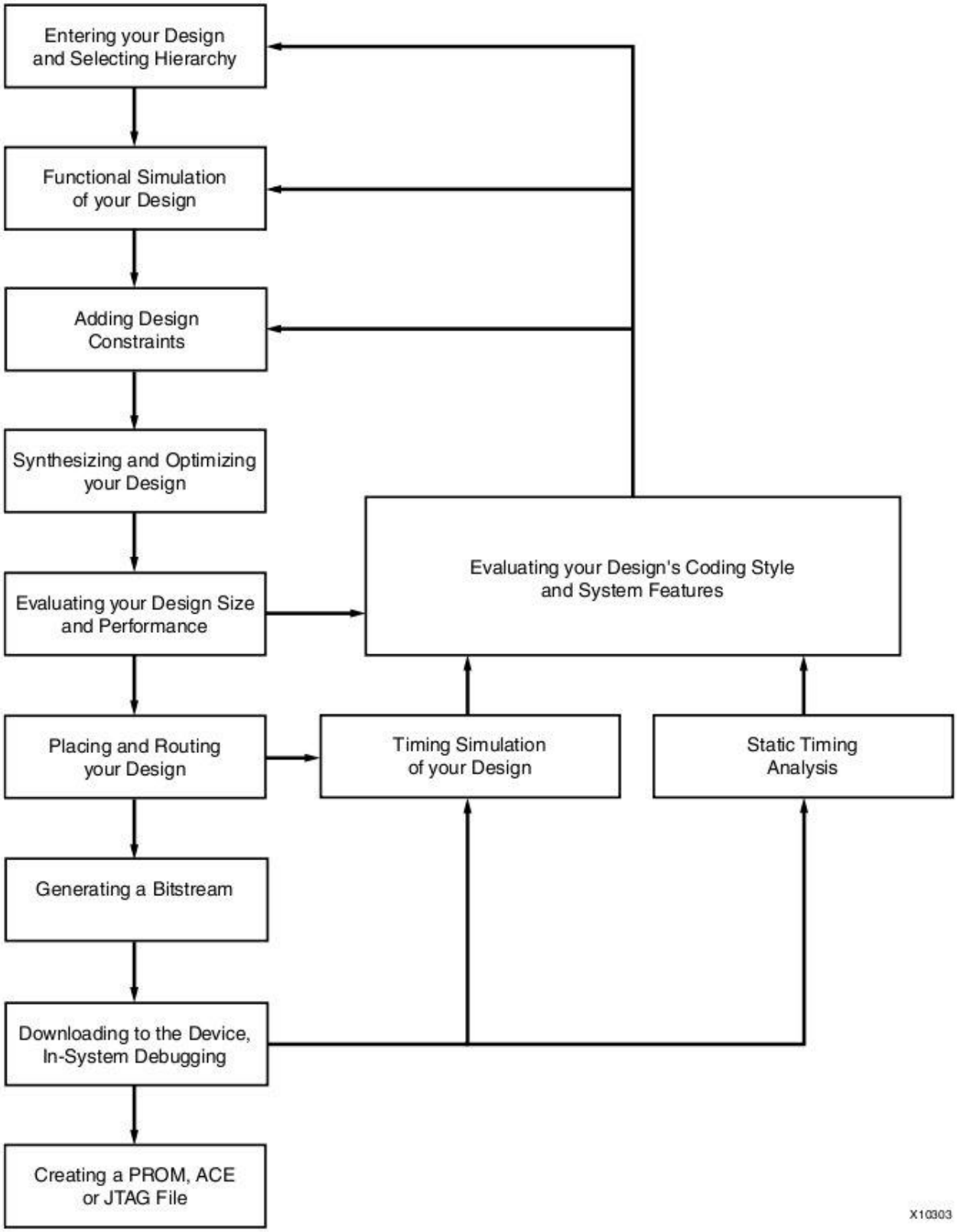

Figure 2.1 Xilinx design flow (figure from [14])

The translate stage reads any user specified constraints along with the synthesized design to generate a generic database file. This is followed by a mapping of the synthesized netlist onto the resources of the target architecture. The next stage involving the placement and routing of the mapped design is the most crucial and time critical phase of the flow. The design is first placed in the available resources on the FPGA and then routed to make the necessary connections. The routed design is then converted to a bitstream that is downloaded to the target FPGA.

Table 2.1 provides a comparison between the time taken for each step of the design flow, using the Xilinx tool set, for two distinct designs. The times are listed as a percentage of the total design cycle time. The first design is a simple MaxRGB filter targeted for the *XC5VLX110T* part occupying around 50 SLICEs. The second design is a much larger 256-point Fast Fourier Transform (FFT) module, implemented for the *XC5VLX330* part consisting of 8500 SLICEs. The numbers clearly show that the Place and Route (PAR) stage is the most time critical part of the entire process and its effect on the cycle time increases with an increase in the complexity of the design. This stage is, therefore, a perfect candidate for optimization in flows targeting enhancements to the existing design cycle.

Part Name	Design	XST	Translate	Map	PAR	Bitgen
xc5vlx110t	Max Filter	10%	6%	14%	38%	32%
xc5vlx330	256 pt FFT	16%	2%	6%	66%	10%

Table 2.1 Xilinx implementation times for two different designs

2.1.2 Incremental Flow

The standard design flow has certain limitations in case of large designs. Any small change in the logic at the HDL level leads to the entire flow being run again, which consumes a lot

of time. The fact that a majority of the design remains unchanged in such cases has been leveraged by FPGA vendors and researchers to come up with incremental flows. The next two sections present the details associated with these flows.

Vendor Efforts

Xilinx has come up with two different strategies for reducing design compile times using incremental analysis. The first approach called **SmartGuide** [1] involves keeping track of the changes in the current netlist with respect to the netlist of the previous run. These changes then serve as a guide to the tools to implement only the modified part of the logic. This is particularly useful for relatively stable designs that undergo only minimal changes in pin locations or timing constraints. The second technique provided by Xilinx is **Partitions** [2] that is useful for block based designs. In this case, the design is initially partitioned into blocks based on functionality or any other valid criterion. The tools then keep track of the individual blocks and the blocks that do not change are just "copy and pasted" from the previous iteration thus improving run times. Altera has also introduced the **Incremental Compilation** [3] feature that involves building a design incrementally by preserving results of unchanged portions from previous iterations.

While incremental design approaches do save time compared to the traditional flow, they are not without drawbacks. The block based approach needs the designer to specify the partitions at a very early stage introducing inflexibility. Also, incremental synthesis is a major component of these incremental flows. It is used for identifying changes at the netlist level but does not take into account any information about the physical location of the design. Since the major component of the design cycle is the placement and routing, the lack of knowledge about the physical aspects of the design leads to high implementation times even with the incremental flows. This showcases the need for a new productivity flow

that utilizes the physical characteristics of the design.

Research Efforts

Aside from the flows developed by Xilinx and Altera, there has been a lot of other research into formulating new incremental design techniques. Since the FPGA design process is composed of many different steps, focussing on optimizing a particular step can lead to a significant improvement over the existing flow. This section describes some of the efforts in incremental floorplanning and placement.

The PATIS framework [15] targets the FPGA productivity problem by implementing an incremental floorplanner. Unlike conventional static floorplanners, PATIS dynamically updates the floorplan in response to changes in the design. Multiple variants of the initial floorplan are generated based on certain module properties like susceptibility to change and timing sensitivity. When a design changes, PATIS first searches for a compatible floorplan from the existing ones and if unsuccessful, attempts to generate a new feasible floorplan. The tool reads in three separate inputs - a resource estimator, a resource map and a user constraints file (UCF). The resource estimate is done using PlanAhead [21] in conjunction with a top-level design EDIF file and then written into a text file that contains the number and type of each resource used as shown in Figure 2.2. The resource map is a class representation of a particular FPGA device created using XML and XDL [22] files generated for a particular design by PlanAhead. The UCF file specifies constraints associated with particular modules in a design. The tool extracts out the necessary information from these files to create an incremental floorplan that is written into a new UCF file.

```
Physical Resources Estimates
============================

Site Type  Available  Required  % Util
---------------------------------------
LUT        0          7239      No Sites
FF         0          6516      No Sites
SLICEL     0          4416      No Sites
RAMB16     0          12        No Sites


Carry Statistics
================

Type
--------------------------------------------------------------------------------
Number of carry chains    330
Longest chain             pblock_module0/pblock_module26/sub_cfft_2/cfft1/
  amulfactor/u1/u1/gen_pipe[10].Pipe/Maddsub_Xresult_cy<0>
Carry height utilization  N/A (5 CLBs)
```

Figure 2.2 PATIS resource estimate file (figure from [19])

Another interesting incremental design approach is showcased by Replace [16], a fast incremental placement algorithm. Replace is different from previous efforts in incremental placement in that it evaluates the cost function only at the end of the process thus making it more scalable. There are three inputs to the placement process - an initial placement before modification, a floorplan identifying the region to place the changed elements and a modified design. This algorithm is implemented in the Versatile Place and Route (VPR) [17] framework that uses its own proprietary format for describing the placement information as shown in Figure 2.3.

```
Netlist file: xor5.net   Architecture file: sample.arch
Array size: 2 x 2 logic blocks

#block name x     y     subblk      block number
#---------- --    --    ------      ------------
a           0     1     0           #0  -- NB: block number is a comment.
b           1     0     0           #1
c           0     2     1           #2
d           1     3     0           #3
e           1     3     1           #4
out:xor5    0     2     0           #5
xor5        1     2     0           #6
[1]         1     1     0           #7
```

Figure 2.3 VPR placement format (figure from [18])

The first line of the file lists the netlist and the architecture files (also in the VPR format) and the second line gives the size of the logic block array used by this placement [18]. All the following lines specify the details of the blocks included in the placement. These include the name of the block, the row and the column coordinates of the site in which the block is placed. The modified design is represented using the VPR netlist format. A snapshot of the VPR netlist file is captured in Figure 2.4. It captures three different circuit elements - inputs, outputs and functional blocks. With the all the appropriate inputs specified, the Replace algorithm generates a fast incremental placement for the modified design.

```
.input a                        #Input pad.
      pinlist: a                #Blocks can have the same
                                #name as nets with no conflict.

.input bpad
      pinlist: b

.clb simple                     # Logic block.
      pinlist: a b open and2 open          # 2 LUT inputs used,
                                           # clock input unconnected.
      subblock: sb_one 0 1 open 3 open     # Subblock line says the
                                           # same thing.

.output out_and2                      #Output pad.
```

Figure 2.4 VPR netlist format (figure from [18])

Both the approaches discussed above yield good quality incremental solutions; however, in each case there is a need for the specification of inputs in multiple formats. This involves repeated conversions between the proprietary vendor formats and the internal tool formats resulting in design time overhead. A single format that is standard across all the stages of the design flow would not only save the overhead resulting from the various conversions but also simplify the incremental algorithms being implemented. This thesis presents one such unified format, qBase, that is used as a standard for processing information at various stages of accelerated design flows.

2.2 Related Work in Industry

As discussed in the previous section, there a number of steps involved in implementing an FPGA design. At almost each and every stage of the design cycle, a different tool and hence a different file format are involved. Further, each of the major FPGA vendors have their own design suite. With so much variance, it becomes very complex to migrate between the various stages in the design flow or from one vendor device to another as there is a constant need to convert between formats. It is clear that a uniform model for representing data would not only simplify the process but also save a lot of development cycles.

Over the years, there have been various efforts by the industry towards building a unified standard. The next few sections detail some of the relevant work undertaken in this regard.

2.2.1 NeoCAD

In the early 1990s, a company by the name of NeoCad Inc. created a set of complete FPGA design development tools supporting FPGAs from multiple vendors [6]. NeoCad's

environment, known as FPGA foundry, was a back-end tool that supported Xilinx, Actel and Motorola devices. The majority of the functionality provided was related to the technology mapping, placement, routing and timing analysis [7]. This was one of the first efforts towards a unified, vendor-independent tool set for FPGAs even though it was restricted to the back-end part of the FPGA design flow. The quality of the NeoCad tools led to its acquisition by Xilinx and thus curtailed a possible move towards a tool supporting the entire FPGA design flow.

2.2.2 EDIF

The need for a unified format is not restricted to FPGAs but is equally relevant to ASICs. The ASIC design flow comprises of a series of incompatible file formats requiring data translation at various stages. In cases where this translation is attempted, the time and cost factors make the process not only complex but also error prone. In order to simplify this problem, a group of EDA companies together with the University of California at Berkeley came up with the Electronic Design Interchange Format (EDIF) [5]. The central idea behind EDIF is to be able to represent the design at various levels without loss of information. With universal acceptability, the EDIF has become a part of practically every vendor's tool set and allows different views of the design including electrical connectivity, geometric layout and behavioral description of the circuit. There have been several enhancements to EDIF one of which creates databases to provide better efficiency [4]. The success of EDIF reinforces the fact that an easy to use universal format is always likely to be widely adopted.

2.2.3 OpenAccess

A universal database concept for ASICs has also been explored by Cadence Design Systems, the leading provider of Electronic Design Automation (EDA) tools for ASIC design. The OpenAccess database [8], pioneered by Cadence, is an effort in that direction. OpenAccess is an advanced EDA database that provides a set of Application Programming Interfaces (APIs) to transform designs at various levels. With a single format representing both an HDL as well as a post-synthesis list, the design process is much simpler yet highly efficient. The ease of integration with existing flows and the breadth of information captured by the database has made OpenAccess the choice of development platform for many EDA companies. The OpenAccess data model has been developed to support a variety of features needed for chip design including connectivity, geometry and hierarchy. It provides 3 different views (*module, block, occurrence*) for representing designs with each covering a particular aspect of the design process. These views are discussed in detail below [9].

1.) Module View

This view captures the details associated with a module at the logical level. This primarily includes information about the logical connectivity and hierarchy of the design. A single entity in this view typically corresponds to an HDL (Verilog/VHDL) definition of a module. There are various attributes associated with a module that help in describing it. These include the module name, its connectivity with respect to the rest of the design and its parent/child modules.

2.) Block View

This view provides information about the physical implementation of a design. Properties of the design that are introduced at the implementation level are incorporated into the block definition. The main attributes included at this level are design characteristics such as the physical name of a module (name in Library Exchange Format (LEF) file), boundary of a module and its associated shape, and physical connectivity.

3.) Occurrence View

There are certain cases when tools need to work on flat designs to make exact modifications. This view provides a fully flattened representation of the design with all the logic instantiated in a single top-level file.

The OpenAccess effort is currently being led by the OpenAccess coalition, which consists of over 30 industry leaders in the EDA domain showcasing its growing importance.

2.3 XML Based Schema

The XML, a standard developed by the World Wide Web Consortium (W3C), is one of the most popular formats for representing metadata i.e data about data. The basic structure of an XML file, in the form of attributes, tags and elements, lends itself very amicably to capturing important data associated with the FPGA design process. Crucial parts of the design flow such as module names, net connections, hierarchy etc. can all be easily captured by simple XML structures. The data is easily readable and there is a lot of existing support for XML parsers, which makes navigating through the data a trivial process. Also, XML is a universal language that can be easily ported across different platforms and hence does not

have any dependency issues.

2.3.1 IP-XACT

The usefulness of the XML format is perfectly illustrated in the the implementation of the IP-XACT specification [10]. SPIRIT, a consortium of electronic system, Intellectual Property (IP) provider, semiconductor, and EDA companies has come up with an XML based schema to represent metadata associated with components and designs within an electronic system called the IP-XACT [11]. The motivation behind this endeavor is to be able to use a common standard for describing IP blocks from multiple vendors and also provide a single database for the various tools involved in the design process to interact with. The metadata associated with the design is captured in various objects and a set of APIs allow tools access to this data.

The IP-XACT specification defines seven different objects to capture the information related to a design - *bus definition, abstraction definition, component, design, abstractor, generator chain and design configuration*. The first four objects are relevant to the presented work and are discussed in detail below.

1.) Component

The component object is the top-level description of a given IP block that can be used to describe processor cores, memories, Direct Memory Access (DMA) controllers, buses etc. A component can be static or configurable and the hierarchy of the component is also specified based on whether it has sub-modules or not. A component object contains information about bus interfaces, address spaces, memory maps etc. associated with that component.

2.) Bus and Abstraction Definition

The bus and the abstraction definition are combined together to form an interface description that mainly deals with the connectivity of a particular module. The bus definition deals with the high-level attributes of the interface such as the connection method whereas the abstraction definition contains low level details such as the name, width and direction of the ports.

3.) Design

The design description is more like the architectural description of the design. It specifies all the instantiated components, their configurations as well as the interconnections between them. The interconnections are typically between interfaces or between ports of a component.

IP-XACT has been used as a base to build a number of useful applications that have showcased its easy adaptability and usefulness. The CHREC XML created by BYU is a perfect example of one such application [12]. It is intended to be a database that captures parameters associated with modules at various levels of abstractions. The capabilities of the tool also facilitate reuse of cores leading to a greater increase in productivity.

2.4 TORC

TORC is an open-source infrastructure and tool set for Xilinx FPGA design that was jointly developed by University of Southern California's Information Sciences Institute East (ISI East), Virginia Tech's Configurable Computing Machines Laboratory (CCM Lab), and Brigham Young University's Configurable Computing Laboratory (BYU CCL). The TORC tool set has the capability to manipulate generic and physical netlists, provide exhaustive

wiring and logic information for commercial devices and manipulate bitstream packets.

Figure 2.5 depicts the TORC tool set with all the different components. The entire TORC functionality is encapsulated into four APIs - *generic, architecture, physical and bitstream*, each of which is described below [13].

1.) Generic

This is the API that deals with the logical level netlists. It provides importers and exporters to read and modify the netlists in the EDIF format. Elements of a netlist including cells, libraries, ports, instances and nets can all be retrieved using this API.

2.) Physical

The physical API provides support for designs that have been mapped for a particular device. This is extremely useful in extracting information regarding the location of a port, the sources and sinks of a net, and other such physical characteristics of the design. Since there is only the bitstream generation after the physical implementation, changes to the design made at this level are retained in the generated bitstream as well.

3.) Architecture

This API serves as a database for the particular device being used. It contains extensive information about the device wiring resources, logic resources and also keeps track of the wire and arc usage.

4.) Bitstream

The Bitstream API provides the capability of reading, modifying and writing bitstream packets. Again, the possibility of directly manipulating the bitstream information to affect the functionality of designs is very useful for productivity gains.

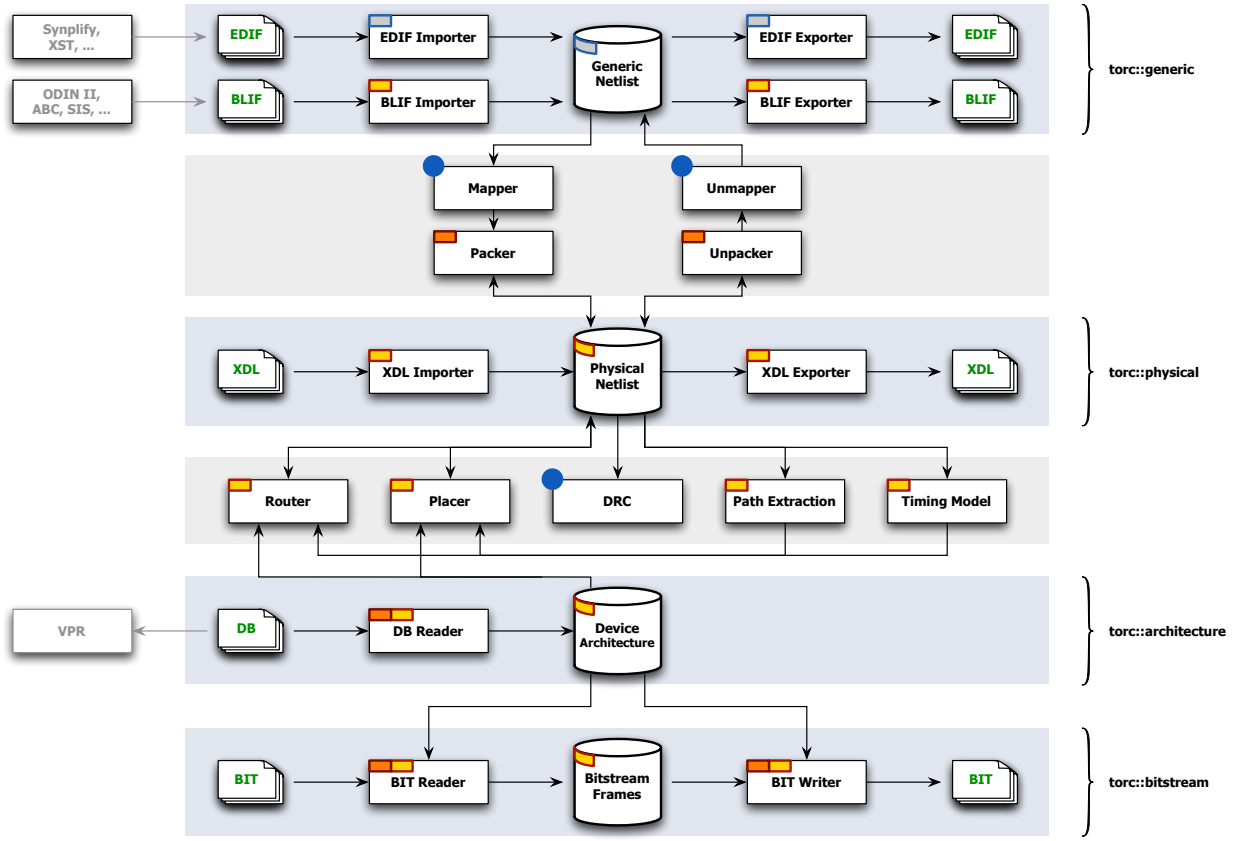

Figure 2.5 TORC block diagram (figure from [13]).

Besides these APIs, TORC also has its own set of tools including a packer, unpacker and a router that can be used instead of the vendor tools in certain cases. The vast range of functionality offered by TORC makes it the perfect framework for building new flows targeting productivity. Since qBase is central to the efficiency of the productivity flows, it is imperative that it is easily integrated with TORC. TORC is written in C++ and makes

extensive use of BOOST [20], which is a set of libraries complementing the standard C libraries. BOOST has built-in support for XML files in the form of archives and serialization, which makes it convenient to create XML files for designs being modified using TORC.

Chapter 3

qFlow

This chapter provides a high-level overview of a FPGA productivity flow. The central idea behind the development of this flow, called qFlow, is to accelerate the back-end stage of the FPGA design cycle using incremental knowledge of modified designs. The management paradigm presented in this thesis is central to the implementation of this flow and serves as a bridge between the different nodes in the process.

Figure 3.1 illustrates the central idea behind qFlow using a flow diagram representation. The rest of the chapter uses this diagram as a reference to describe the different facets of the flow. This chapter is divided into four sections. The first section discusses the problem domain that motivates the need for qFlow. The second section provides a brief introduction to the methodology developed and relates to the formulation step in the diagram. The third and fourth sections describe the different threads that work in synchronization and the the interfaces between them.

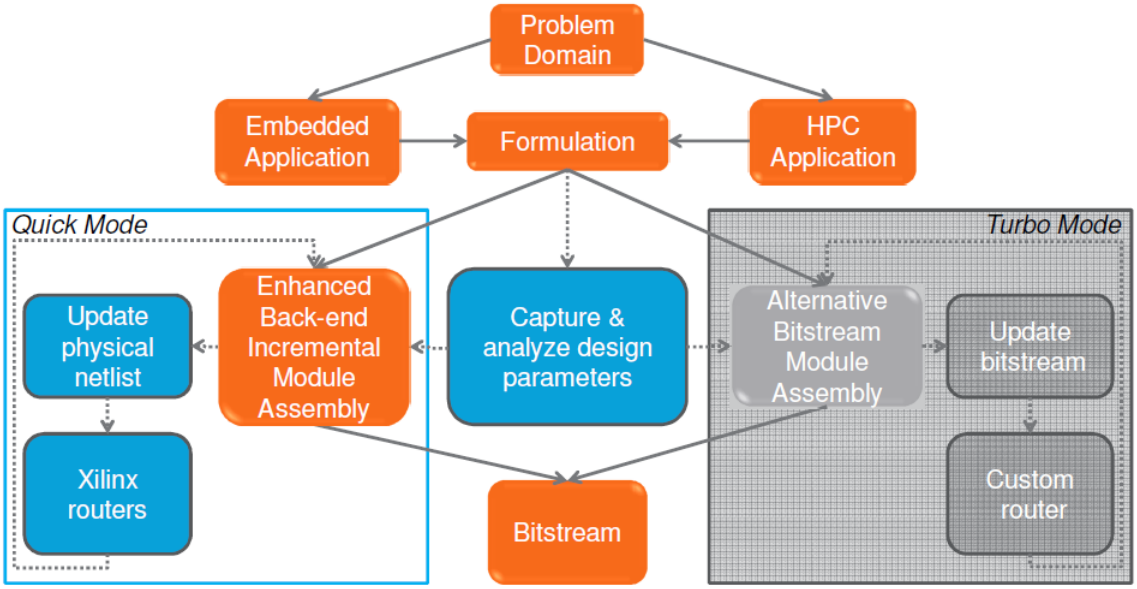

Figure 3.1 Consolidated qFlow model.

3.1 The Problem Domain

The standard FPGA flow is an iterative process of design entry, implementation and verification, until a functioning system is developed. Consequently, even small changes to the design need to go through the entire FPGA design cycle. The back-end processes of map, place and route are particularly time consuming. This considerably slows down the design process and makes it difficult to quickly update the design. qFlow aims to provide an incremental solution that uses information from previous iterations to substantially reduce the design cycle time with an emphasis on the back-end design.

qFlow is proposed as a generic flow that can be targeted for any kind of application. For the initial phase of its development, qFlow targets application designs in three domains - *embedded, high performance computing (HPC)* and *software defined radios*. These three domains are quite diverse and each have designs with their own set of requirements. The

HPC applications involve huge designs with large runtimes and are perfect candidates for improving productivity. The software defined radio is another domain that involves a lot of incremental modifications to the design. Such a diverse set of disciplines provide a perfect platform for illustrating the robustness and the flexibility of qFlow.

3.2 Formulation

For a successful implementation of any incremental flow, a unique approach is needed that is able to utilize the existing information about a design to guide future iterations. When a design changes, there is a need to preserve as much of the current structure as possible so that the run-time of the tools is substantially decreased. This section discusses some of the ideas that comprise the methodology for qFlow and the benefits of using them.

One of the prime motivations behind qFlow is to give the user the choice of design entry mechanisms. Since the focus of the flow is to accelerate the back-end, it is independent of the front-end design approach used. It could be a simple Verilog/VHDL file, an ImpulseC model [31] or any other graphical design entry tool that is provided as an input to qFlow after which the back-end process initiates.

The central concept behind the development of qFlow is the use of *module based design*. The idea here is to treat all logic associated with a design as part of some module. Glue logic associated with the design is also encapsulated into a modular form. A very simplistic view of the top-level will then be a collection of hierarchical modules and a set of connections between these modules. The concept of a *hard macro* [24] is interlinked to the module based design approach and is a critical component of the flow. Hard macros can basically be treated as pre-compiled modules that have been synthesized, placed and routed for future use. Typically, modules encapsulating a particular kind of functionality are created as hard

macros so that they can be treated like black boxes that can be dropped into any existing design. Since the placement and routing of the components internal to the hard macro is already done, it leads to rapid implementation times. Prior work done in this regard [25] shows that there is a significant improvement in the compilation times when using hard macros.

The hard macro creation, used in conjunction with the module based approach forms the basis of qFlow. As mentioned above, a user specified front-end design is the input to qFlow and triggers the back-end operation. Some parts of the input design, like the DDR/Ethernet interfaces, are fixed with respect to their functionality and location for a particular device. These need to be implemented only once even if the underlying design undergoes multiple changes. For the purposes of this thesis, such logic is referred to as *static logic*. The rest of the design is assumed to be composed of *dynamic modules*, which are implemented using hard macros. These macros are then inserted into the design and the tools are used to make the appropriate connections to complete the design. Any modification to the design is detected at the top-level and is treated as either modifying an existing macro(s) or creating a new one. Thus, adding new logic to an existing design is equivalent to inserting one or more modules into the design and making the connections for the new module. This is done without affecting the portion of the logic that was unchanged from the previous iteration. Since the mapping, placement and routing associated with the unaffected logic was preserved, it leads to a significant improvement in the design compile times.

Figure 3.1 shows two different modes for implementing a design using qFlow - *Quick mode* and *Turbo mode*. The underlying principle behind both of them is the same as described above; however, they use different techniques to generate the bitstream. The quick mode uses the vendor tools (Xilinx in this case) to generate the bitstream whereas the turbo mode uses the routing and bitstream manipulation capabilities of the TORC tool set to do the same.

Figure 3.2 shows an estimate of the speed improvements that are seen by using the qFlow (both quick and turbo mode) as opposed to the traditional FPGA flow. These are based on a set of initial experiments done on some embedded and HPC designs. The delta mode run times are of particular interest since they denote the time taken to perform incremental changes, which is the focus of this endeavor.

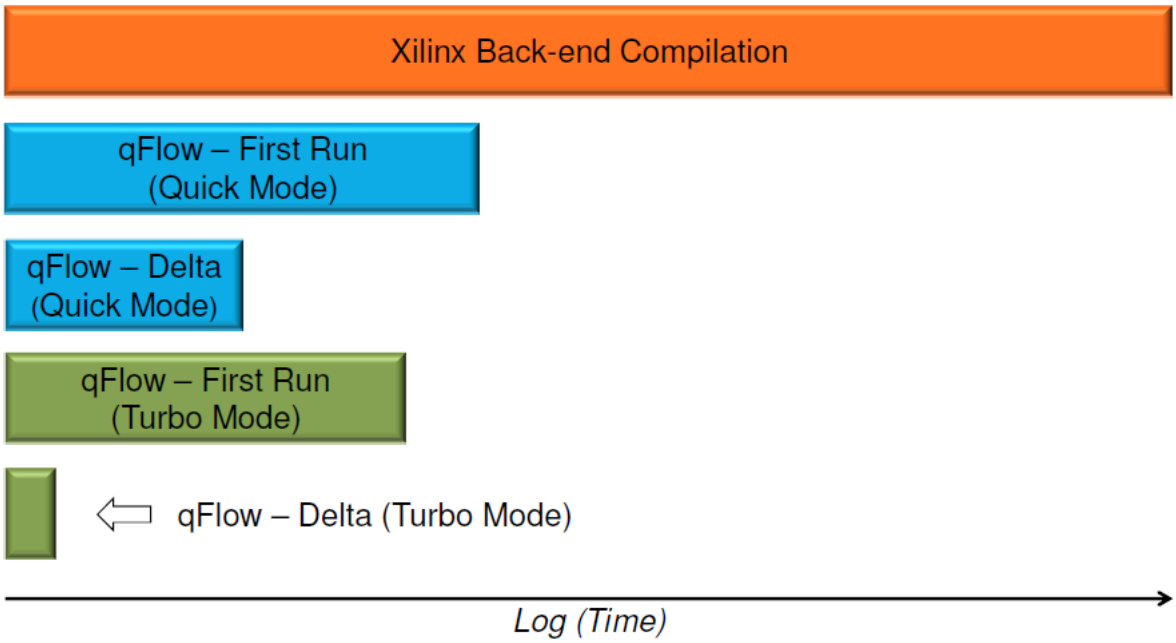

Figure 3.2 qFlow run-times.

3.3 The Big Picture

The entire qFlow design process can be thought of as a multi-threaded approach with five main threads. Figure 3.3 identifies these five distinct tasks as *the consolidator, the architect, the librarian, the locator* and *the conductor*. The following subsections briefly explain each of these.

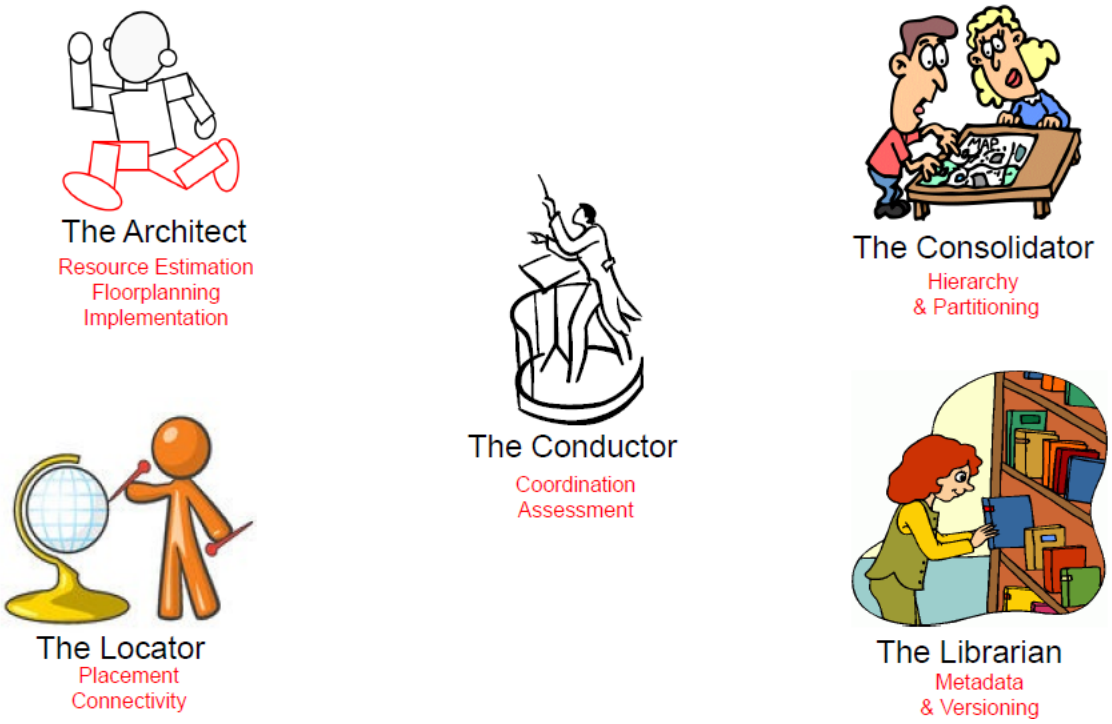

Figure 3.3 qFlow components.

1.) Consolidator

The consolidator is provided a netlist of the user design and extracts a hierarchy tree out of it. The partitioning of the design, where each partition is defined as a grouping of modules that is to be treated as a hard macro, is also done by the consolidator.

2.) Architect

The architect, as the name suggests, is primarily responsible for the creation of the hard macros that realize this flow. Taking a partitioned netlist as an input, the architect estimates the resource requirements for the various macros, decides on the shape of the physical implementation of these macros, and finally implements them for a given FPGA.

3.) Librarian

The librarian is tasked with the capture of the metadata associated with the design and the versioning of the different accepted solutions for a design.

4.) Locator

The main task of the locator is to find the best possible location for the placement of a hard macro generated by the architect. In case placement of a particular macro fails, the locator communicates it back to the architect who tries to floorplan the macro in a different way to better fit the design. Once a macro is successfully placed, the locator also ensures proper connectivity between the macro and the existing design.

5.) Conductor

The conductor is the controller of the qFlow process. It is the conductor who drives the implementation process and ensures synchronization between the various threads, which leads to a feasible solution. Any communication between the other threads has to go through the conductor. The conductor also has the final say on whether a proposed solution is acceptable or not, based on certain criteria.

The successful execution of each of these threads leads to a comprehensive design flow that significantly reduces the time taken to compile incremental FPGA designs. The next section explains in detail the interaction of the conductor with the other qFlow threads.

3.4 The Conductor Functionality

As described in the previous section, the conductor is the central node of qFlow and all the other threads need to interact with it. The following sub-sections detail the interface between the conductor and each of the qFlow threads.

3.4.1 Conductor-Consolidator Interface

The conductor passes a netlist in EDIF format to the consolidator who extracts the design hierarchy tree, partitions the design, extracts inter-partition connectivity and builds the top-level netlist. The inputs and outputs with respect to the consolidator are described below.

Inputs

- **EDIF Netlist**: This is the design netlist in EDIF format. It is important for the synthesis tool to KEEP hierarchy when generating the EDIF netlist.
- **Design Interfaces**: This is the list of modules that are considered as interfaces in the design. Often, the interfaces throughout a design lifetime are invariant and can be marked by the designer.

Outputs

- **Hierarchy Tree**: The hierarchy tree is a tree that shows all modules and the corresponding sub-modules in the design.

- **Partitions (EDIF Format)**: Every design is divided into partitions. A partition is a grouping of modules to be treated as a single hard macro. The partition size is bounded between a minimum and a maximum logic utilization. A partition should not contain any interface logic.

- **Connectivity List**: This contains the list of inter-partitions and partitions-interfaces nets. It serves as a guide for connecting the various hard macros during the implementation process.

- **Top-level**: The top-level file is in EDIF format, which consists of the design EDIF with all the computing logic removed. Moreover, all connectivity between the interface and the computing logic is to be done through bus macros which are inserted in the top-level EDIF by the consolidator. This will prevent the back-end tools from complaining whenever implementing the top-level, and will eliminate the need to route to IOBs, a requirement for the Turbo mode of qFlow.

3.4.2 Conductor-Architect Interface

The conductor takes the design partitions generated by the consolidator and forwards them to the architect. The architect, in turn, concurrently implements the different partitions. The inputs and outputs with respect to the architect are described below.

Inputs

- **Partitions (EDIF Format) along with interfaces**: Each partition is implemented as its own EDIF netlist. Moreover, for each partition, the list of input and output ports is needed.

- **Top-level (EDIF format)**: The top-level file generated by the consolidator.
- **Device Information**: The FPGA architecture and part being targeted.
- **User Constraints**: The user constraints specify the location where each hard macro is to be implemented on the FPGA die and any implementation relevant constraints.

Outputs

- **Hard Macros**: The architect generates two hard macros per partition. For each partition, both hard macros are identical except for routing: one hard macro is placed and routed whereas the other is just routed. Each hard macro implements the corresponding partition logic. It incorporates bus macros on the input side of the hard macro to handle nets fanouts, by shifting it from the hard macro interface to the inside of the hard macro.
- **Top-level**: This refers to the implemented top-level (mapped, placed and routed). The top-level implements the design interfaces and incorporates any necessary sandboxes where computing logic will be hosted.

3.4.3 Conductor-Librarian Interface

The conductor passes the metadata data structure to the librarian who is capable of serializing the C++ data structure and deserializing XML. The librarian also versions any solution (the ensemble of hard macros, metadata, placement information, etc...) that the conductor accepts. The librarian functionality can be further sub-divided into serialization, deserialization and versioning. The inputs and outputs with respect to each of these are described below.

Serialization Input

- **C++ Data Structure**: The C++ data structure holding the information to be serialized.

Serialization Output

- **XML File**: The XML file generated by serializing the C++ data structure.

Deserialization Input

- **XML File**: The XML file generated by serialization of the C++ data structure.

Deserialization Output

- **C++ Data Structure**: The C++ data structure holding the information specified in the XML file.

Versioning Inputs

A **design solution** would be the input to the versioning process. A solution consists of the following components.

- A top-level file.
- The various partitions implemented and the corresponding hard macros.
- All specified constraints, including UCF constraints and hard macro placement.
- All metadata, including top-level metadata and individual hard macros metadata.

There is no associated output for this process.

3.4.4 Conductor-Locator Interface

The locator takes the different hard macros, places them in the top-level, connects the hard macros and finally routes all the design nets. The inputs and outputs with respect to the locator are described below.

Inputs

- **Hard Macros**: The list of hard macros generated by the architect.
- **Connectivity List**: The connectivity list generated by the consolidator.
- **UCF**: The designer can control the placement of specific hard macros using this method, and any other constraints such as the timing constraints are passed using this file.

Output

- **Fully Routed Design**: The final routed design that is ready to be passed to Bitgen for bitstream generation.

Chapter 4

System Overview

The management paradigm, presented in this thesis, aims to provide a simple yet elegant solution for interlinking the various stages of qFlow. The main endeavor in this regard is to provide a common file format representing the design that can be read/modified at any level. This is accomplished by creating qBase, a comprehensive database, that captures all the necessary information required for a successful implementation of qFlow. Internally, qBase is implemented using C++ classes and it is viewed externally in the form of XML files.

This chapter describes in detail the various aspects related to the construction of qBase. It starts with a brief description of the FPGA coordinate system, the choice of which is an integral part of the whole process. Next, all the attributes comprising qBase that are used to capture the relevant design information are discussed. Finally, all the details associated with the implementation of qBase are explained.

4.1 The FPGA Coordinate System

Since qFlow targets the back-end FPGA design, information about the physical implementation of the design is a critical aspect of the flow. In case of a module, this typically includes the site coordinates of a given port or the boundary coordinates of the module. Capturing this information relative to a particular location on the module is integral to the flexibility and the robustness of qBase. The choice of an appropriate coordinate system is, therefore, extremely important. A typical cartesian coordinate system, shown in Figure 4.1, uses X and Y coordinates to uniquely represent a given location on a plane with respect to the origin. A similar concept of coordinate systems can be applied to FPGAs.

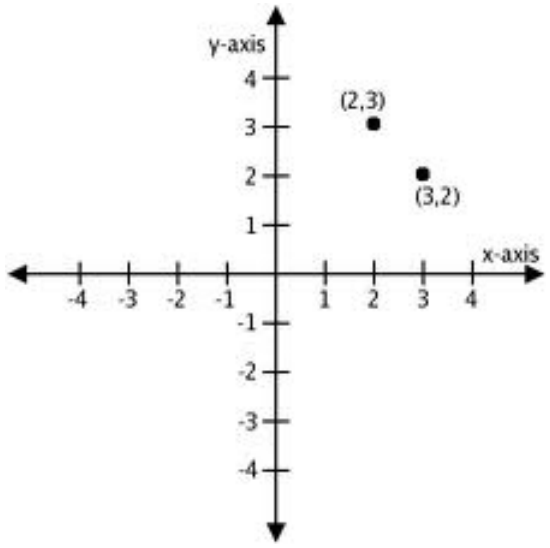

Figure 4.1 Cartesian coordinate system.

FPGAs, in general, can be associated with two kinds of coordinate systems - site and tile coordinates. Sites are resources in an FPGA that contain logic/buffers and include SLICEs, RAMB36s and DSP48s [23]. Tiles are elements within the FPGA two-dimensional grid that are composed of one or more sites and include CLBs, DSPs and BRAMs. Both sites and tiles have their own set of coordinate systems. However, while the site coordinate system is

specific to the site type, i.e a different coordinate system for SLICEs and RAMB36s, the tile coordinate system is uniform across the FPGA and independent of the type of the tile. The FPGA editor [26] snapshot in Figure 4.2 illustrates this point.

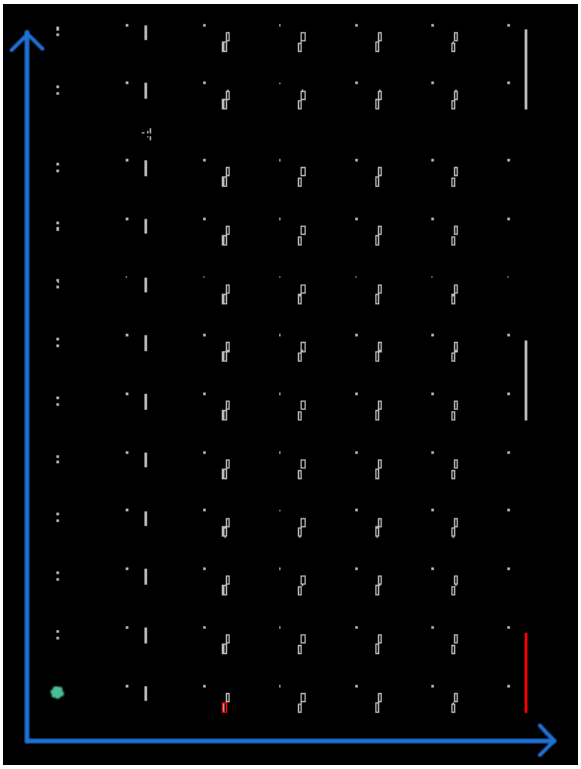

Figure 4.2 FPGA editor snapshot for the XC5VLX110T device.

The left most and the southern most tile (IOB in this case) is denoted by the green circular coordinate in the figure and acts as the origin. The blue horizontal and vertical axes indicate the positive X and Y directions respectively. All the tiles on the FPGA are referenced relative to the origin in the positive X and Y directions. For the site coordinate systems, however, every new type of site encountered, acts as a new origin. This results in multiple coordinate systems, one for each site type. For e.g, the first SLICE and the first RAMB36 for the *XC5VLX110T* part (highlighted in red in Figure 4.2) both have a site coordinate of X0Y0 but the tile coordinates for these are X1Y0 and X5Y0 respectively. Since a module can

contain different types of site locations, and each of them needs to be referenced relative to a common location, the tile coordinate is used as the reference coordinate system for all the locations, boundaries and other numerical information specified in qBase.

4.2 The XML Attributes

As discussed in the previous chapter, the top-level design is treated as a combination of modules and connections between those modules. Consistent with this modular approach, a set of attributes is selected that completely describes the logical and physical attributes of any particular module. These attributes form the basis for the creation of qBase. The following subsections describe in detail the various attributes associated with a module. The external view of each of these attributes in the form of XML definitions is also shown.

4.2.1 Name

The first attribute captured in a module definition is its *name*, which mirrors the name associated with the module in the top-level HDL.

4.2.2 Anchor

Every module has a special attribute associated with it called the *anchor point*. This is typically a site on the module that is used as a reference for all the other module attributes. The anchor point is described by the type of the tile (CLB, DSP, BRAM) used as the anchor and the absolute X and Y coordinates of the corresponding tile. A site_index is also defined that is explained in detail in section 4.2.3. All the other values captured in qBase are relative to this value. This is particularly useful in case of module relocation. If the placement of

a particular module changes, its just the new anchor point that needs to be updated while the rest of the attributes can be calculated by using the previously captured relative values.

A snapshot of the anchor attribute in XML form is shown below.

```
<anchor>
    <type>CLBLL</type>
    <x>10</x>
    <y>20</y>
    <site_index>1</site_index>
</anchor>
```

4.2.3 Ports

The next attribute to be captured is the *ports* of the module. The port attribute is best explained by breaking it down into two parts. The first part is the basic information about the ports provided by the HDL that includes the *port name, its direction (input/output) and its width*. The second part involves including information about the physical location of these ports.

There are different factors that need to be considered for representing the physical information of ports. A single port at the HDL level can map to multiple ports at the physical level in certain cases (for e.g hard macro creation). Every such location associated with a particular port needs to be captured, since the top-level connections are specified in terms of the HDL port names. Also, in case of a multi bit signal, a differentiation needs to be made between the individual bits since each of them might be associated with a different functionality. Taking into account all these aspects, a *tileloc* attribute is defined for every port. This is further composed of two sub-attributes - *bit* and *tile*.

The bit attribute specifies the particular bit of the signal whose location is being described. A single-bit signal would have a single entry with the bit attribute value set to 0. The tile attribute contains a detailed description of the physical location of the signal. The first field specifies the *type* of the tile, which could be a CLB, a DSP or a BRAM. The next two fields, provide the *X and Y coordinate* of the tile relative to the anchor point of the module. The next parameter is the *site_index* that is used for identifying the particular site of a given tile to which the signal is associated. For e.g, for the *XC5VLX110T* part, the tile CLBLL_X16Y40 consists of two slices, SLICE_X26Y40 (site_index=0) and SLICE_X27Y40 (site_index=1). Since these slices have a different site_index associated with them, it can be used to differentiate between them and pinpoint the correct slice. The next field is the *pin* that identifies the particular pin on the site to which the signal is connected to. Some examples of pins include the CLK or AQ pin of a SLICE or the DI pin of a RAMB36. The last field is the *pin_name* that is useful in uniquely representing every tile associated with a particular signal. As explained earlier, any particular bit of a signal has one port name but can be physically represented with multiple tiles and each of these tiles has a unique user-defined pin_name associated with it.

A snapshot of the XML representation of the ports is shown below.

```
<port>
  <name>rgb_in</name>
  <direction>in</direction>
  <width>1</width>
  <tileloc>
    <bit>0</bit>
    <tile>
      <type>CLBLM</type>
```

```
            <x>-5</x>
            <y>3</y>
            <site_index>0</site_index>
            <pin>B1</pin>
            <pin_name>rgb_in<0>_0</pin_name>
          </tile>
          <tile>
            <type>CLBLL</type>
            <x>1</x>
            <y>4</y>
            <site_index>1</site_index>
            <pin>C4</pin>
            <pin_name>rgb_in<0>_1</pin_name>
          </tile>
          <!-- continue for all sites of rgb_in, bit 0 -->
        </tileloc>
      </port>
```

4.2.4 Boundary

Another critical parameter associated with modules is the *module boundary*. This is crucial while trying to place the module, since the boundary estimate would give a fair idea of the resources needed to place it and if the existing design has enough free space to be able to place the module. An important concept here is the association of multiple shapes with a given module. There might be a possible scenario during the implementation of a design

such that a given module, with an associated shape, is not able to be placed in the existing design. However, it is possible that by changing the shape of the module implementation, the placement goes through. This requires a module to be able to maintain more than a single shape. The boundary attribute is thus defined to be composed of one more *region* attributes, with each region describing a shape associated with the module.

A region in this context is constrained to be rectangular. A region is further divided into four sub fields - *north-east X and Y* and *south-west X and Y* coordinates that are sufficient to denote a rectangular area. This area is not constrained by the tile types included inside of it and can encompass CLBs, DSPs and BRAMs. These coordinates are also specified with respect to the anchor point showcasing the importance of choosing the tile coordinate system as the reference.

The XML representation of the boundary attribute is shown below.

```
<boundary>
  <region>
    <sw_x>-1<sw_x>
    <sw_y>3<sw_y>
    <ne_x>3<ne_x>
    <ne_y>5<ne_y>
  </region>
  <region>
    <sw_x>4<sw_x>
    <sw_y>8<sw_y>
    <ne_x>7<ne_x>
    <ne_y>10<ne_y>
  </region>
```

```
</boundary>
```

4.2.5 Sub-Modules

Most of the modern FPGA designs have multiple levels of hierarchy with modules containing sub-modules and so on. It is very important for qBase to provide information about the hierarchy of a module and its building blocks. The *sub-module* attribute is provided for capturing the hierarchical information associated with a module. This attribute is composed of a set of *module instances* that are the building blocks for a module. Each module instance is defined as having a *name, instance and an anchor point.* The instance field is required in cases where there are multiple instantiations of the same sub-module. The anchor definition is similar to the one described in Section 4.3.2 and specifies the absolute location of that sub-module.

A sub-module definition in XML is shown below.

```
<submodule>
  <mod_inst>
    <name>LowPassFilter</name>
    <instance>lpf</instance>
    <anchor>
      <type>CLBLL</type>
      <x>19</x>
      <y>26</y>
      <site\_index>1</site\_index>
    </anchor>
  </mod_inst>
```

```
    <mod_inst>
      <name>MaxFilter</name>
      <instance>max_filt</instance>
      <anchor>
          <type>CLBLM</type>
          <x>55</x>
          <y>64</y>
          <site\_index>0</site\_index>
      </anchor>
    </mod_inst>
  </submodule>
```

4.2.6 Connections

In cases of hierarchical modules, another piece of information that needs to be captured is the connections between the sub-modules. Each of these sub-modules will have their own qBase representations that describe the corresponding port locations and boundary. The *connections* attribute describes the connections between each of the sub-modules and is composed of a set of *nets*. Each net is further decomposed into a *source and a set of sinks*. The connections attribute can be read to get the list of nets and then the individual sub-module qBase representations can be read to get the exact pin locations. Custom tools can then use this information to create the appropriate connections.

A snapshot of the connection attribute is shown below.

```
  <connections>
    <net>
```

```
        <source>lpf.read</source>
        <sink>max_filt.load<0></sink>
        <sink>max_filt.load<1></sink>
    </net>
</connections>
```

4.2.7 Top-Level

The previous sections described in detail the various attributes associated with a module definition. The top-level design comprises of only the modules that are instantiated in the top-level HDL file and the connections between them. The top-level qBase representation can therefore be treated as a special case of the module-level representation. The boundary, sub-module and the connections are the attributes of the module definition that are captured at the top-level as well. This essentially reduces the top-level to a set of instantiated modules and connections between those modules. The ports and the anchor attributes are unpopulated for a top-level qBase representation.

4.3 Implementation

4.3.1 Overview

qBase has been created such that it can be easily incorporated into the TORC framework. Since TORC is based on C++, all the functionality encapsulated into qBase is done using C++. Separate classes are created for every high-level attribute discussed in the previous section, with the sub-fields created as data members of that particular class. There are

certain sub-fields of an attribute that need to be specified multiple times. For example, the port attribute discussed in Section 4.2.3 can have multiple tiles associated with it. These are implemented using the *list* container from the Standard Template Library (STL). Lists perform better in inserting, deleting and moving positions within a container [27] and hence are preferred over *vectors and deques*. Every class also provides functions that enable the user to read/write the corresponding data members as well as traverse any list attributes.

4.3.2 Boost and XML Support

As discussed in Chapter 2, BOOST allows a seamless creation of XML files using the concept of *serialization and archives* [28]. Serialization refers to the conversion of an arbitrary data structure to a sequence of bytes. This can be used to regenerate the structure in a different program context. The term archive refers to a specific rendering of this stream of bytes, which could be a file of binary data, a text file or as used in this thesis, an XML.

The data structures used for the creation of the module attributes are serialized using the *serialize* function. BOOST provides the *BOOST_SERIALIZATION_NVP* macro that is invoked on every data member to serialize it. The serialization can be directly applied to an object of a class as well, thus preventing the need to serialize each member individually. Also, the BOOST serialization class should be declared as a *friend* [29] function of the given class so that it has access to all private and protected data members of the class.

A sample code of a class implementing the serialize function for a particular attribute is shown below.

```
class xml_module{
friend class boost::serialization::access;
void serialize (Archive & ar, const unsigned int version)
```

```
{
    ar & BOOST_SERIALIZATION_NVP(name);
    ar & BOOST_SERIALIZATION_NVP(ports);
    ar & BOOST_SERIALIZATION_NVP(anchor);
    ar & BOOST_SERIALIZATION_NVP(boundary);
    ar & BOOST_SERIALIZATION_NVP(module);
}
}
```

Once the data members have been serialized, it is trivial to convert it to an XML using file streams and the BOOST archives. The *xml_iarchive and xml_oarchive* are the BOOST archives that support the import and export of XML files. The top-level module object is passed as an argument to the serialization macro to generate the corresponding XML file. The following code snippet shows the creation of a new XML file that is written with the contents of the top object.

```
std::ofstream ofs("test.xml")
boost::archive::xml_oarchive oa(ofs);
oa << BOOST_SERIALIZATION_NVP(top)
```

4.3.3 Code Structure

As described earlier, all the module attributes are captured as a set of classes and their associated functions. A proper hierarchical structure is needed to clearly define the level at which a particular attribute resides with respect to the entire structure. Figure 4.3 depicts all the files comprising the framework and their hierarchical structure. The module is the topmost node of the diagram and the rest of the attributes are described with respect to

it. The bold and capitalized names represent all the individual classes created. The other names denote the respective data members of a particular class. The figure also illustrates the cases where an individual class acts as a data member for another class.

4.3.4 Directory Organization

Figure 4.4 captures the directory structure of the files created for qFlow. The qFlow directory has two sub directories - *applications* and *common*. The common directory further has the *xml* sub-directory that contains all the files defining the qBase attributes. The figure only shows the .cpp files but there are corresponding .hpp files for each of these to include the necessary header files. The common directory contains programs that use the qBase attributes to create a series of useful functions. The applications directory contains the top-level files that use the functions in the common directory to create programs for the appropriate applications. Consequently, the C++ *main()* function is found only in programs in the application directory. The TORC directory is parallel to the qFlow directory and allows the applications developed in qFlow to be easily integrated with TORC. At every level, there is a Makefile provided that is responsible for creating executables for the files at the same level and the top-level Makefile invokes all the other Makefiles.

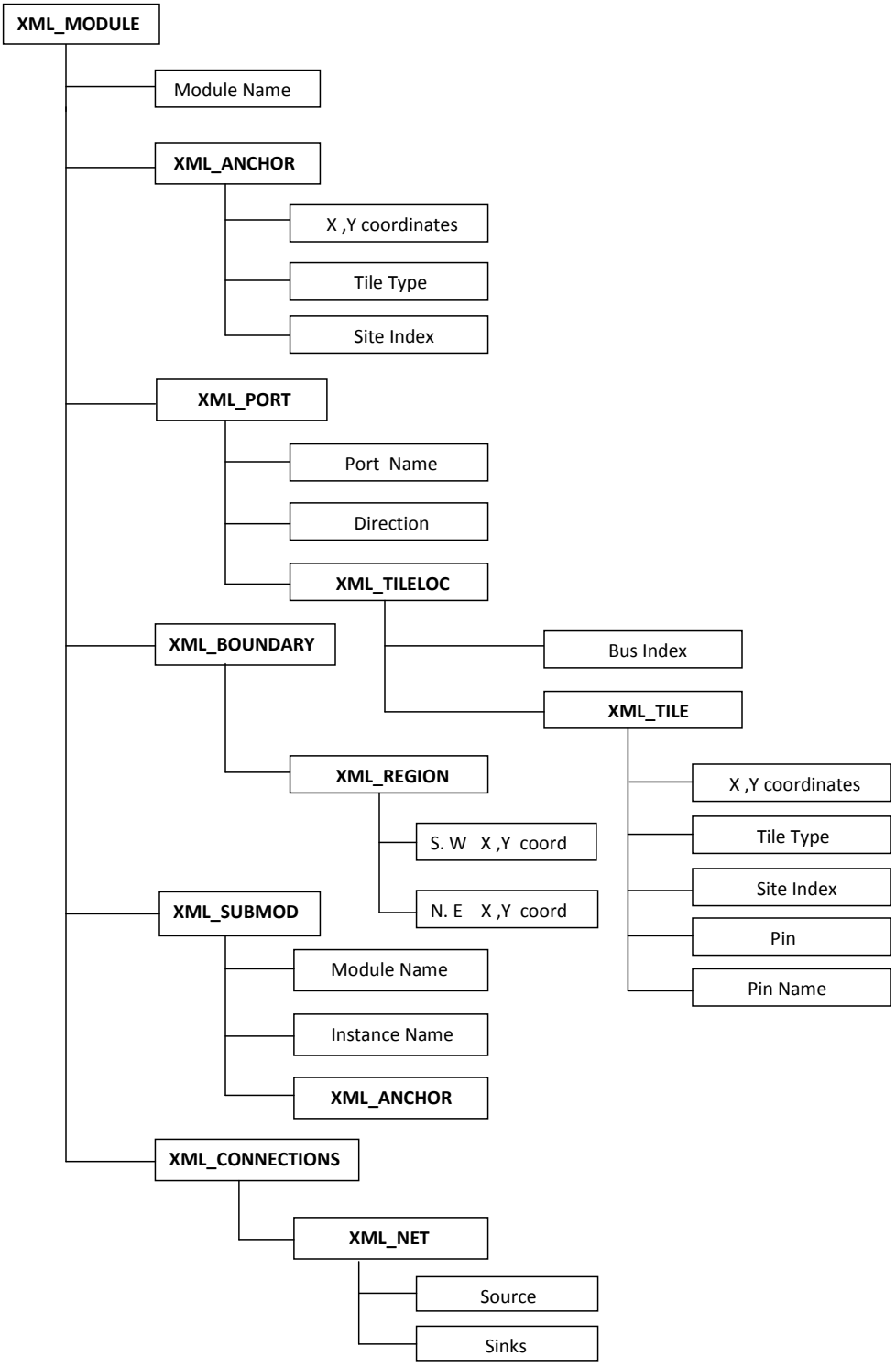

Figure 4.3 qBase class hierarchy structure

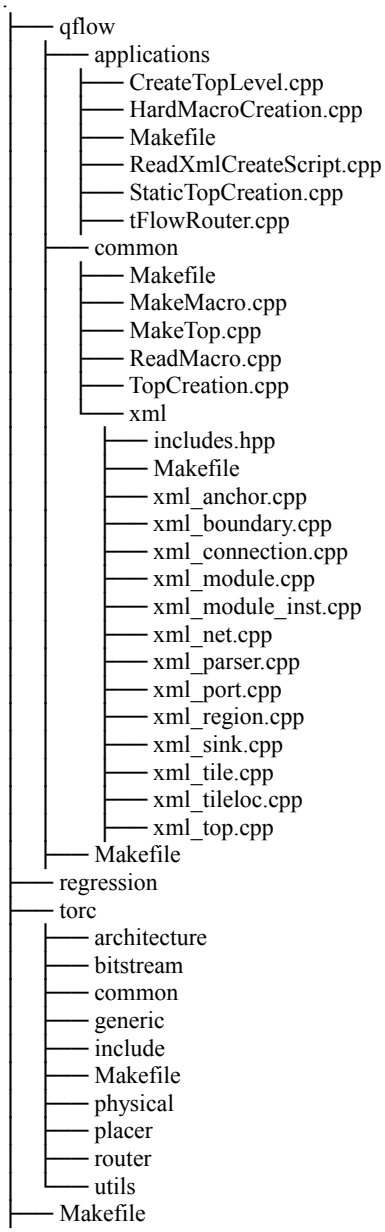

Figure 4.4 qFlow directory organization

4.3.5 Application Development

The classes described in Section 4.3 capture all the requisite information associated with a given design. These first need to be populated with an initial set of values and can then be read or modified based on the application being developed. A brief description about three different programs developed for qFlow, using the qBase data structure is given below.

1. Module Development

As discussed in Chapter 3, the dynamic modules used in qFlow are created as hard macros. The hard macro creation process is basically a program developed in the TORC framework using C++. By simple additions to this program, all the module attributes associated with the hard macro under creation are populated and stored into an XML associated with the module.

2. Static Development

The dynamic modules need to be placed in an existing design, which is generally a top-level design with static interfaces like Ethernet/DDR interfaces. A simple program is developed that populates the top-level module structure specifying the locations of the static design and the connections between the static and the dynamic modules.

3. Connectivity Creation

The information about the physical attributes of the dynamic modules as well as the top-level design are available after executing the above two programs. The creation program reads the connection list and generates a corresponding pin to pin connectivity using the

physical locations. This can then be used by custom routers to do the actual routing thus implementing the comprehensive design.

The use of these three programs in conjunction illustrates the utility of qBase and provides a perspective about possible avenues of using it. The above example uses only a certain subset of the attributes provided and full utilization of the available information can motivate other endeavors in this direction.

4.3.6 Internal and External Data Structures

All parameters of interest associated with a typical FPGA design are captured in some form or the other in the XML files. However, it is important to note here that the XML files are an external data structure capturing the design metadata. These files are easily readable by the user and provide a means of visually analyzing the information associated with a design. They are also useful in debugging issues regarding any incorrect data captured in qBase. However, internally it is the qBase data structures, created using the C++ classes that are actually the source of information. During any process of qFlow, qBase can be queried for information regarding the design. After the completion of the process, qBase is updated back with the appropriate changes. The corresponding XML files are then modified to reflect the updated design. Since there is a single uniform format that is representing the design at every level, the management of the data is straightforward and it provides a means of efficient communication between the various nodes of qFlow.

Chapter 5

Results

The aim of this chapter is to demonstrate the utility and applicability of the management paradigm created in this thesis. The first part of the chapter discusses the various test applications that have been developed using qFlow, with qBase queried at different stages for design information. The second part of the chapter presents a comparison between qBase and some of the generic file formats used in the ASIC design process. This section also discusses the internal file formats used in some of the current FPGA incremental flows and contrasts it with qBase.

5.1 Test Designs

The test designs for this work were chosen to demonstrate the validity of the information captured in qBase, leading to a successful execution of qFlow. As described in Chapter 3, these designs were chosen in three different domains - embedded, high-performance computing (HPC) and software defined radios. Table 5.1 provides a brief description of the three applications.

Domain	Part	Interfaces	Modules	Slice Count
Embedded	XC5VLX110T	VGA/DVI Ethernet DDR	Video Filter Address Swap Memory Testbench	2028
HPC	XC5VLX330	Memory	256 point FFT pair	24262
Software Defined Radio	XC5VLX110T	Ethernet	Low Pass Filter Zigbee Radio Zigbee Interface Signal Detector	3915

Table 5.1 The initial set of test designs for qFlow

5.1.1 Embedded

The embedded design application is targeted for the *XUPV5* board with the *XC5VLX110T* part. The top-level design comprises of three static interfaces - a VGA/DVI interface, an Ethernet interface and a DDR memory interface. The corresponding modules implemented are a Grayscale video filter, an Ethernet address swap module and a memory testbench module. The static logic is implemented once and then each of the individual modules are dropped in and appropriate connections to the static design are created using the physical location information stored in qBase. Figure 5.1 shows a block diagram for this application. The signals associated with the block diagram are explained in Appendix A.

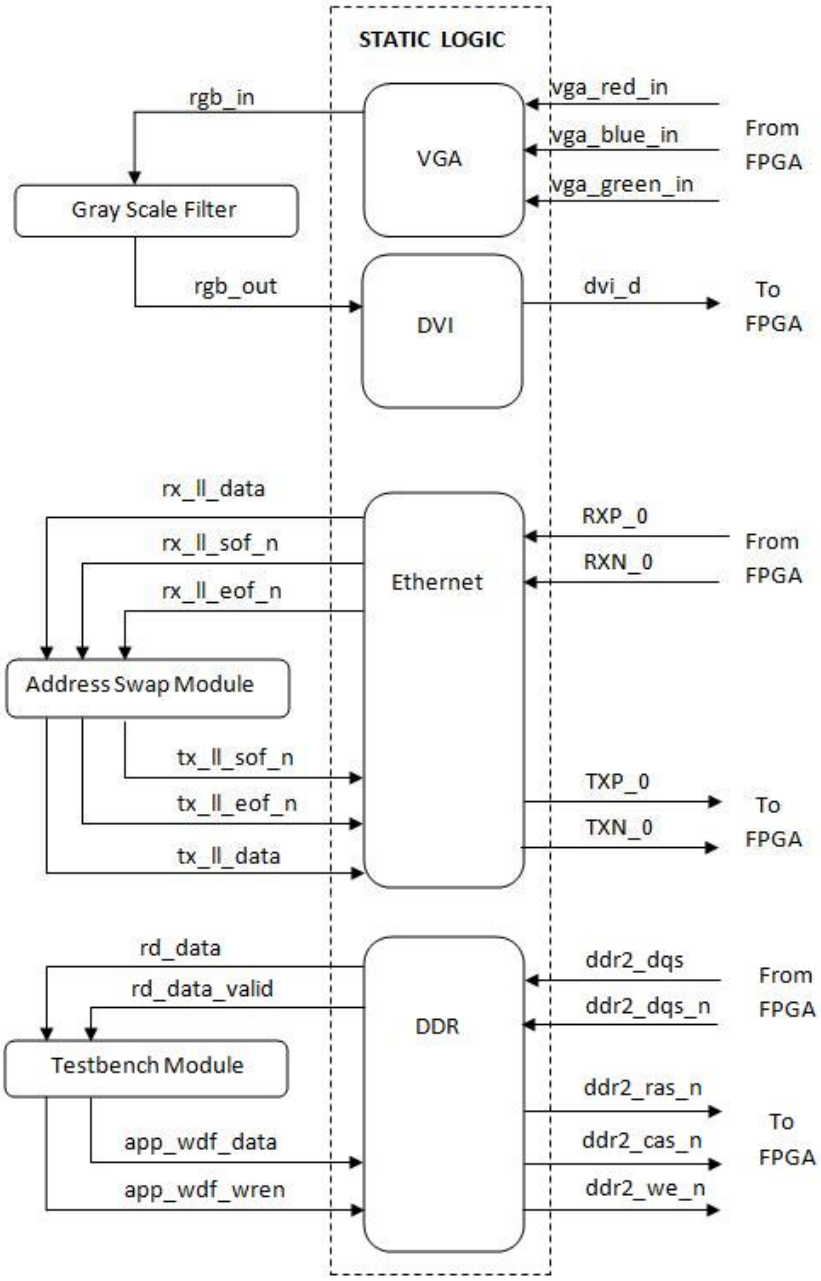

Figure 5.1 Embedded application

5.1.2 High-Performance Computing

Applications for high-performance computing are generally quite complex and as such are prime candidates for accelerated flows. The design used here is a pair of 256-point Fast Fourier Transform modules targeted for the *Convey-HC1* [30] processor with the *XC5VLX330T* part. The static logic in this case consists of the Convey Memory interface. The implementation process is the same as in the case of the embedded design. The seamless transition from a simple embedded design to a complex HPC design emphasizes the universal applicability of the framework. Figure 5.2 shows a block diagram for this application. The signals associated with the block diagram are explained in Appendix B.

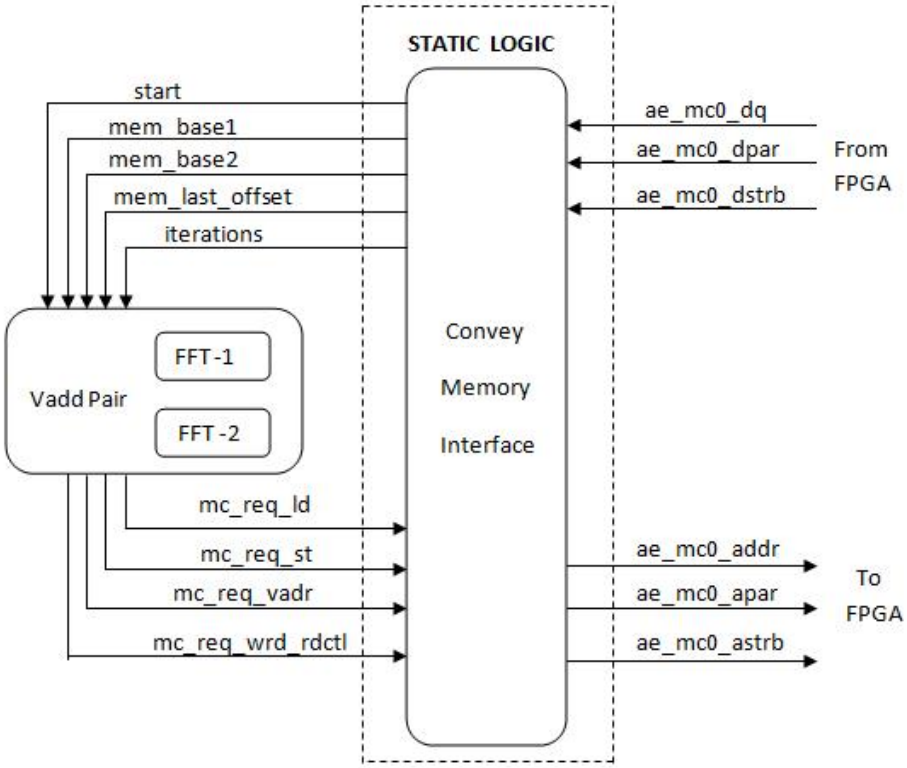

Figure 5.2 High-Performance Computing application

5.1.3 Software Defined Radio

Another interesting application is that of Software Defined Radios that is also targeted for the *XUPV5* board. It consists of a single static interface (Ethernet) and two static modules (packet encoder and decoder), but has multiple dynamic modules interacting with each other as well as the static logic. This application provides a dense design with respect to connectivity since the modules also connect to one another apart from the connections to the static design. A successful implementation of this design once again illustrates the robust nature of the framework. Figure 5.3 shows a block diagram for this application. The signals associated with the block diagram are similar to the signals of the Ethernet part of the embedded application that have already been detailed in Appendix A.

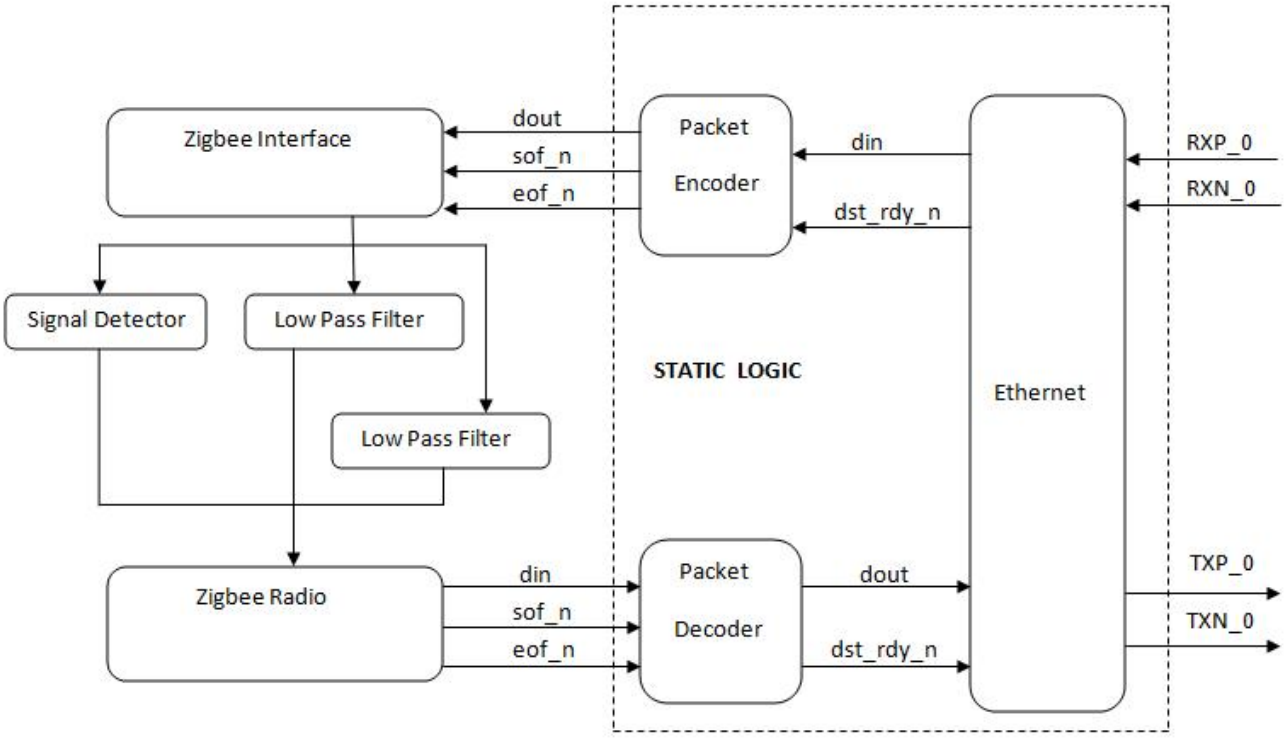

Figure 5.3 Software Defined Radio application

5.2 Feature Comparison

This section illustrates the unique features of qBase with respect to the FPGA design process. It is further divided into two sub-sections. The first one compares the salient features of qBase with some of the generic file formats currently in use. The advantages and disadvantages of qBase with respect to these are also enlisted. The second sub-section compares qBase with the internal file formats used in existing incremental flows and details the benefits.

5.2.1 Generic File Formats for ASICs

Chapter 2 provided a detailed background on generic file formats for ASIC design, with an emphasis on two popular formats - IP-XACT and OpenAccess (OA). Table 5.2 enlists some of the key features that characterize a database for FPGA design. These features are used as the basis for a comparison between qBase and IP-XACT/OA.

Domain	Feature	IP-XACT	OpenAccess	qBase
Logic Design	Hierarchy Support	Yes	Yes	Yes
	Ports	Yes	Yes	Yes
	Connectivity	Yes	Yes	Yes
	Front-End Flexibility	Yes	Yes	Yes
Physical Design	Ports	Yes	Yes	Yes
	Name Preservation	No	No	Yes
	Physical Location	No	No	Yes
	Connectivity	Yes	Yes	Yes
Geometry	Shape of a Module	No	Yes	Yes
	Multiple Shapes per Module	No	No	Yes
Miscellaneous	Uniform Format Throughout Flow	Yes	Yes	Yes
	Timing Attributes	Yes	Yes	No
	Easily Extendable	Yes	Yes	Yes
	External Data Structure	Yes	No	Yes

Table 5.2 Feature comparison of qBase with IP-XACT and OpenAccess

The table illustrates that for a high proportion of the features listed, all the three formats provide good quality support. However, for some features there is a difference in the capabilities of the three formats. The next sections discuss the advantages and disadvantages of qBase in relation to these features.

Advantages

The following are some of the key features where qBase provides better support as compared to IP-XACT and/or OA.

1. Name Preservation One of the most crucial advantages of qBase is the fact that the names of the ports remain the same when moving from the logical to the physical domain. The name of a particular signal captured at the front-end remains the same while updating its physical characteristics. In contrast, for both IP-XACT and OA, there are different names at the logical and physical level, which requires a mapping structure to move from the front-end to the back-end thus complicating the process.

2. Physical Location Another important feature is the information about the actual physical location. Both IP-XACT and OA have features to capture the physical name of signals. However, neither of them provides the ability to capture information specific to the location of a given signal. This can be attributed to the fact that these formats have been designed for ASICs and as such the location for signals is not of much importance. However, this information is crucial for FPGAs and qBase provides the capability of capturing location specific information.

3. Geometry A critical aspect of FPGA design is to capture the geometry associated with modules and is typically represented using shape attributes. IP-XACT does not provide any feature to capture shapes and while OA does allow the user to capture shapes, it is restricted to one shape per module. As discussed in Section 4.2.4, a module can be associated with multiple shapes if the resources consumed by the top-level design do not allow the placement of a previously valid shape. qBase provides support for associating multiple shapes with a module and keeping track of each them uniquely.

4. Front-end Flexibility The method of design entry supported by each format is quite flexible. In case of IP-XACT and OA, since they primarily deal with ASICs, the design entry methods are restricted to the HDL level. There is, however, support for different

techniques of creating the HDL. These can include Transaction Level Modeling descriptions (TLM), compiled core models, configurable HDL descriptions or traditional Verilog/VHDL descriptions. On the other hand, there are certain tools available for FPGAs including ImpulseC [31] and Viva [32] that bypass the HDL design stage and directly create a netlist. The structure of qBase ensures that independent of the front-end design method, the design information will still be captured by the same set of attributes.

5. External Data Structure In addition to the internal formats used for the data capture, it is always useful to have a corresponding external representation. OA does not provide any such utility but IP-XACT and qBase utilize the XML framework to encapsulate the internal data structure into XML files. These files are especially useful in debugging as well as getting a quick high-level understanding of the design structure.

Disadvantages

While qBase has some important advantages over the existing formats, it is limited with respect to the timing data of the design.

1. Timing Support The current version of qBase does not have any support with respect to the timing attributes. IP-XACT and OA provide features that capture various aspects of the timing information. This can be useful for identifying the critical paths of designs in cases where the timing fails.

5.2.2 Internal Formats of Incremental Flows

Section 2.1.2 described two incremental approaches for improving FPGA productivity - PATIS and Replace. This section compares the formats used in those processes with qBase.

1. Replace vs qBase

Replace implements an incremental placement algorithm and takes three different files as inputs. Table 5.3 compares the internal file formats used in Replace with qBase.

Input/Output	**Replace Internal Format**	**qBase Support**
Initial Placement	VPR placement format	Yes
Floorplan for changed elements	Text file	Yes
Modified design	VPR netlist format	Yes
Output Placement	VPR placement format	Yes

Table 5.3 Comparison of the internal file formats between Replace and qBase

The first file is the original placement of the design in the VPR placement format indicating the locations occupied by the original design. The same information can also be extracted from qBase by reading the boundary attribute of every module instance present at the top-level. The second input is the floorplan identifying the region for the changed elements. In the qFlow framework, every change in logic manifests itself as addition of a new module. A new module being inserted into the design would have a boundary associated with it that is estimated along with the creation of the module and serves as the floorplan for that module. The third input is the modified design in the VPR netlist format. Again, qBase captures this information at the very outset and all changes in the design are reflected in the database in the form of addition of new modules to the data structure at the appropriate hierarchy

level. The three inputs to the Replace algorithm require three different input formats as well as other interconversions to extract and process the information contained in them. On the other hand, each of these inputs can also be specified using qBase, without any loss of information. Moreover, the output of this process, which is a placement for the changed design, can also be easily captured by specifying the anchor point for the new module. There is no need to maintain files in multiple formats and the various interconversions are also avoided leading to a much more efficient process. This illustrates the uniqueness and the universal nature of qBase.

2. PATIS vs qBase

Table 5.4 shows a similar file format comparison between the incremental floorplanner of PATIS and qBase. The PATIS floorplanner also takes in three inputs.

Input/Output	**PATIS Internal Format**	**qBase Support**
Resource estimate	Estimate text file	No*
Resource map for changed elements	Map text file	Yes
Constraints	User Constraint File	Yes
Output Floorplan	User Constraint File	Yes

Table 5.4 Comparison of the internal file formats between PATIS and qBase

The first input in this case is the resource estimate for a design in a text file. Now, qBase in its current format does not have direct support for resource estimates. However, the calculation of a shape for a module involves resource estimation as an intermediate step. It would just require the creation of a new class with the required attributes and it can then be included as a data member of the module class. This illustrates the ease of extendability

of qBase based on the application. The other inputs and output of PATIS are similar to the ones discussed for Replace and hence fully supported by qBase.

Chapter 6

Conclusion

This thesis presented a management paradigm for capturing information associated with various stages of an accelerated FPGA design flow. The discussion started with a background of the basic FPGA design flow, along with a summary of the industrial and academic research projects targeting incremental flows. Some of the existing database formats used to represent ASIC designs at various levels were also discussed. This was followed by a high-level overview of qFlow, an enhanced FPGA productivity flow targeted at the back-end stage of the FPGA design flow. The next chapter described the details of the system implemented and explained the intricacies of the internal and external data structures involved. The final section enlisted the initial set of designs used for the validation of the approach discussed towards a successful implementation of qFlow. It also presented a feature-based comparison between qBase and some of the other formats described in the background section. This section also illustrated the universal applicability of qBase to the internal formats used in other incremental FPGA flows.

The technique presented in this thesis has been shown to be an efficient and elegant method for storing the information associated with a typical FPGA design. The use of a single format

throughout the flow eliminates the need to store files in different formats thus simplifying the whole design process.

6.1 Future Work

There is scope for improving the system designed by making certain enhancements to the system designed. The following points summarize the key areas of future work.

- The ease of extendability of qBase allows for making changes to its structure. In order to better incorporate incremental designs, extra attributes can be added to the data structure that help in identifying the modified portion of a logic. Hashes can be used to keep track of current versions of a module and a change in the hash can be represented by a dirty flag
- The current tests have been carried out on Xilinx FPGAs; however, the universal nature of the data structure allows for vendor independence. The next stage could involve adding the necessary capabilities to qBase for testing designs with Altera FPGAs
- The structure of qBase is quite similar to the IP-XACT schema for ASICs described in the background chapter. With some minor changes to the structure, it can be made IP-XACT compliant and distributed as a plug-in for IP-XACT

Bibliography

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

[2] Incremental design reuse with partitions. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp918.pdf

[3] Quartus II incremental compilation for hierarchical and team-based design. [Online]. Available: http://www.altera.com/literature/hb/qts/qts_qii51015.pdf

[4] W. Li and H. Switzer, "A Unified Data Exchange Format based on EDIF", Proceedings of the 26th ACM/IEEE Degin Automation Conference, 1989.

[5] D. Chalmers, "History of EDIF and experiences of CAD 031", Proceedings of the IEE colloquium on Electronic Interchange Format, 1988.

[6] Volatile FPGA design security - a survey. [Online]. Available: http://www.cl.cam.ac.uk/ sd410/papers/fpga_security.pdf

[7] Using EnFuzion for Electronic CAD. [Online]. Available: http://www.axceleon.com/appnotes/ecad.html

[8] M. Guiney and E. Leavitt, "An Introduction to Open Access: an open source data model and API for IC design", Proceedings of the 2006 Asia and South Pacific Design Automation Conference, 2006.

[9] Si2 OpenAccess Tutorial [Online]. Available: http://www.si2.org/openeda.si2.org/projects/si2oatools

[10] SPIRIT Consortium, IP-XACT, "A specification for XML meta-data and tool interfaces", 2008

[11] Accelera Home [Online]. Available: http://www.accellera.org/home

[12] A. Arnesen, N. Rollins and M. Wirthlin, "A multi-layered XML schema and design tool for reusing and integrating FPGA", Proceedings of the International Conference on Field Programmable Logic and Applications, 2009

[13] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: Towards an Open-Source Tool Flow", Proceedings of the 19th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2011.

[14] Xilinx Synthesis and Simulation Guide. [Online]. Available: http://www.xilinx.com/itp/xilinx10/books/docs/sim/sim.pdf

[15] T. Frangieh, A. Chandrasekharan, S. Rajagopalan, Y. Iskander, S. Craven, and C. Patterson, "PATIS: using partial configuration to improve static FPGA design productivity", in 17th Reconfigurable Architectures Workshop (RAW), 2010.

[16] D. Leong and G. Lemieux, "RePlace: an incremental placement algorithm for field-programmable gate arrays", in International Conference on Field Programmable Logic and Applications, 2009.

[17] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research", Proceedings of the 7th International Workhop on Field-Programmable Logic and Applications, 1997

[18] VPR and T-VPack User's Manual. [Online]. Available: http://www.eecg.toronto.ca/vpr/VPR_5.pdf

[19] A. Chandrasekharan, Accelerating Incremental Floorplanning of Partially Reconfigurable Designs to Improve FPGA Productivity, Master's Thesis, Virginia Tech, 2010.

[20] BOOST C++ Libraries [Online]. Available: http://www.boost.org/

[21] PlanAhead User Guide. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/PlanAhead_UserGuide.pdf

[22] Xilinx Integrated Software Development Environment 4.2i Documentation, XDL usage is described in $XILINX/help/data/xdl/xdl.html

[23] N. Steiner, A Standalone Wire Database for Routing and Tracing in Xilinx Virtex, Virtex-E, and Virtex-II FPGAs, Master's Thesis, Virginia Tech, 2002.

[24] S. Korf, D. Cozzi, M. Koester, J. Hagemeyer, M. Porrmann, U. Ruckert, "Automatic HDL-Based Generation of Homogeneous Hard Macros for FPGAs", Proceedings of the 19th Annual Symposium on Field-Programmable Custom Computing Machines (FCCM), 2011

[25] C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, and M. Wirthlin, Using Hard Macros to Reduce FPGA Compilation Time, Proceedings of the 20th International Workshop on Field-Programmable Logic and Applications, 2010

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

[27] STL Containers : List [Online]. Available:
http://www.cplusplus.com/reference/stl/list/

[28] Serialization [Online]. Available :
http://www.boost.org/doc/libs/1_46_1/libs/serialization/doc/index.html

[29] Friendship and Inheritance [Online]. Available :
http://www.cplusplus.com/doc/tutorial/inheritance/

[30] Convey-HC1 processor datasheet [Online]. Available :
http://www.conveycomputer.com/Resources/HC-1%20Data%20Sheet.pdf

[31] Impulse Accelerated Technologies [Online]. Available :
http://www.impulseaccelerated.com/

[32] Viva Software [Online]. Available :
http://www.starbridgesystems.com/viva-software/

Appendix A : Description of the Interface Signals of the Embedded Application

The signals comprising the interfaces of the embedded application are explained below.

- **vga_red_in:** An 8-bit signal indicating the red portion of the video signal received from the VGA interface.
- **vga_green_in:** An 8-bit signal indicating the green portion of the video signal received from the VGA interface.
- **vga_blue_in:** An 8-bit signal indicating the blue portion of the video signal received from the VGA interface.
- **rgb_in:** A 24-bit signal created by concatenating the red, blue and green signals. This is the input to the grayscale filter.
- **rgb_out:** A 24-bit output signal from the grayscale filter.
- **dvi_d:** A 12-bit signal sent as an input to the DVI interface.

- **TXP_0/TXN_0:** These comprise the transmit portion of the SGMII Ethernet interface.
- **RXP_0/RXN_0:** These comprise the receive portion of the SGMII Ethernet interface.
- **rx_ll_data:** An 8-bit signal indicating the input data to the address swap module.
- **rx_ll_sof_n:** A 1-bit signal indicating the start of receive frame.
- **rx_ll_eof_n:** A 1-bit signal indicating the end of receive frame.
- **tx_ll_data:** An 8-bit signal indicating the output data of the address swap module.
- **tx_ll_sof_n:** A 1-bit signal indicating the start of transmit frame.
- **tx_ll_eof_n:** A 1-bit signal indicating the end of transmit frame.
- **ddr2_dq:** A 64-bit signal that carries the DDR data.
- **ddr2_dqs:** An 8-bit strobe signal corresponding to the data signal.
- **rd_data:** A 144-bit data signal read from the DDR interface.
- **rd_data_valid:** A 1-bit signal indicating the validity of the rd_data signal.
- **app_wdf_data:** A 144-bit signal with the data that is written to the DDR.
- **app_wdf_wren:** A 1-bit signal indicating enabling of write to the DDR.
- **ddr2_ras_n:** A 1-bit signal indicating the DDR row address strobe signal.
- **ddr2_cas_n:** A 1-bit signal indicating the DDR column address strobe signal.
- **ddr2_we_n:** A 1-bit signal indicating the DDR write enable signal.

Appendix B : Description of the Interface Signals of the HPC Application

The signals comprising the interfaces of the HPC application are explained below.

- **ae_mc0_dq:** A 32-bit signal containing the memory data from a memory controller FPGA.
- **ae_mc0_dpar:** A 1-bit signal indicating the parity of the above data.
- **ae_mc0_dstrb:** A 1-bit signal that strobes the data.
- **start:** A 1-bit signal to indicate the start of the FFT operation.
- **mem_base1:** A 48-bit signal indicating the starting address of the memory for reading the input data to the FFT.
- **mem_base2:** A 48-bit signal indicating the starting address of the memory for writing the output data from the FFT.
- **mem_last_offset:** A 31-bit signal that indicates the number of entries from the starting address that need to be read from the memory.

- **iterations:** A 31-bit signal to indicate the number of iterations of the FFT operation that need to be performed.
- **mc_req_ld:** A 1-bit signal indicating a request for memory load.
- **mc_req_st:** A 1-bit signal indicating a request for memory store.
- **mc_req_vadr:** A 48-bit signal that contains the virtual address sent to the Convey memory interface for converting to an actual physical memory address.
- **mc_req_wrd_rdctl:** A 32-bit signal that serves as a write bus for write transactions and holds the transaction ID in case of read transaction.
- **ae_mc0_addr:** A 24-bit signal that contains the address sent to the memory controller FPGA.
- **ae_mc0_apar:** A 1-bit signal indicating parity of the above address.
- **ae_mc0_astrb:** A 1-bit signal that strobes the above address.