I, Priya Vattyam, hereby submit this as part of the requirements for the degree of:

Master Of Sciences

in:

Computer Engineering

It is entitled:

Performance Modeling Methodologies using PDL+ and ARC

Approved by:

Dr. Ranga Vemuri
Dr. Karen Davis
Dr. Santosh Pande
Performance Modeling Methodologies
using PDL+ and ARC

A thesis submitted to the
Division of Research and Advanced Studies
of the University of Cincinnati
in partial fulfillment of the
requirements for the degree of

Master of Science

in the Department of
Electrical and Computer Engineering and Computer Science
of the College of Engineering

March 2000

by

Priya Vattyam

BE(Hons), Electrical and Electronics,
Birla Institute of Technology and Sciences,
Pilani, India, June 1997

Thesis Advisor and Committee Chair: Dr. Ranga R. Vemuri
Abstract

Analysis for Reconfigurable Computers (ARC) is a software environment in which systems can be modeled and executed. The models in the ARC system are described in a language called Performance Description Language (PDL+). A model of a system in PDL+ is described as a collection of components, attributes, and rules for computing these attributes. Functional attributes of a system describe the functionality of the system, whereas performance attributes are used to evaluate the characteristics of the system. Examples of some typical performance attributes of Very Large Scale Integration (VLSI) systems are clock frequency, area, power, and latency. ARC is a performance modeling environment and PDL+ provides a powerful yet concise notation for specifying generic performance measures of systems.

Though PDL+ as a language had been developed a few years ago, performance modeling methodologies using PDL+ need to emerge. Most of the models written were test cases that demonstrated some selected features of the language. This was partly due to the continuous enhancements that were being made to the language and its environment. Another drawback was that a new language had to be learnt by potential users. The existing documentation has been in the form of language reference manuals and some basic examples, that were clearly not enough to motivate and guide a new user of the language. Through this thesis, we provide a comprehensive guide to the ARC system. We first highlight the various features of the ARC environment and the modeling language. We then demonstrate methodologies for developing performance models in the ARC system. The modeling methodologies are presented with the help of eight different case studies. These models provide good modeling insights to guide future model developers.

Each of the case studies addresses different areas of VLSI systems and computer aided design (CAD) such as, high level synthesis, component libraries spatial partitioning, layout level delay estimation, reconfigurable processors, and power estimation. These models belong to various abstraction levels in the design flow and demonstrate modeling methodologies over different levels of abstraction. Some of the models have also been integrated with existing design flows. We elaborate the usage context, estimation procedure, and model development methodology for these models and also provide their annotated PDL+ implementations. We have also provided an accompanying diskette that contains the models, sample design files, and command files to execute them in the ARC system.
To my wonderful parents
ACKNOWLEDGMENTS

I would like to thank Dr. Vemuri for his continuous support, guidance, and encouragement without whom this work would’ve never been possible. I am grateful to him for all that he has done. I would also like to thank Dr. Davis and Dr. Pande for serving on my thesis committee and for their valuable comments on my work.

This work was supported in part by DARPA/ITO under the Adaptive Computing Systems Program under contract DABT63-97-C0030. I would like to thank Jose Munoz, Al Scarpelli, and Kerry Hill for their suggestions, guidance, and support through the ARC project.

I would like to thank Jeff for all his help and guidance about the ARC system. I thank Ela and Sree for reviewing my thesis draft and for the patience they have shown. I also thank Satish, Sujatha, and Vinoo for helping me throughout the work.

I will always cherish the great times I had at DDEL and thank all DDELites for that. I would specially like to mention the great times I had with Nagu, Sriram, Satish, Sujatha, Sree, and Abhijit. I also had a lot of fun times with Rajesh, Awartika, Iyad, and Vinoo. My interactions with Alex, Adrian, and Naz are memorable. I will also cherish the times with Meenakshi, Karam, Srini, and Amit. I thank Vasantha for the wonderful parties and her enthusiasm. The times with Bhargav are also memorable.

I thank Shakes, PG, Dhyana, SK, Cam, Suj, and Sriram for putting up with me and my antics through my years at UC. I also thank my roommate Ashima for her support and cooperation during the last few months. I also thank Jayanth, Srini, Sugi, Bakshi, and Sunil for constantly encouraging me for as long as I can remember.

I greatly appreciate the love of my sisters Raji and Suchi without which my life would not be half as fun. I also thank Rahul and Gopi for all their encouragement. I am overwhelmed by the love of my parents and their faith in me and will always be grateful to them for everything they have done. I ofcourse would like to thank bunny for being such a darling.
# Contents

1 Introduction .......................................................... 1

1.1 Previous Work ..................................................... 2
1.2 Motivation .......................................................... 3
1.3 Contributions ....................................................... 3

1.3.1 Identifying Systems ............................................. 4
1.3.2 Identifying Metrics ............................................. 4
1.3.3 Estimation Algorithms .......................................... 4
1.3.4 Modeling in PDL+ ............................................. 5
1.3.5 Evaluating the Models .......................................... 5

1.4 The Case Studies .................................................... 5

1.4.1 Library Manager ................................................. 5
1.4.2 RTL Estimator .................................................. 5
1.4.3 Behavioral Estimator .......................................... 6
1.4.4 Partition Estimator ............................................ 6
1.4.5 FPGA Delay Estimator ........................................ 6
1.4.6 Striped FPGA Estimator ....................................... 6
1.4.7 FPGA Power Estimator ....................................... 7
1.4.8 Modeling Reconfiguration ..................................... 7

1.5 Organization of the Thesis ....................................... 7
# PDL+

2.0.1 Design Components .......................................................... 9
2.0.2 Attributes .......................................................................... 10
2.0.3 Evaluation Rules ............................................................... 11
2.0.4 Attribute Propagation ....................................................... 12

2.1 Common Expressions and Statements ...................................... 12
2.1.1 Expressions ...................................................................... 12
2.1.2 Functions ......................................................................... 12
2.1.3 Built-in Functions .......................................................... 13

2.2 Reconfiguration Modeling ....................................................... 13

2.3 Summary ............................................................................. 17

# The ARC System

3.1 Compiler and Evaluator ......................................................... 19
3.1.1 Compilation ..................................................................... 19
3.1.2 Evaluation ...................................................................... 20

3.2 Design File Specification ....................................................... 21
3.2.1 PDL netlist format (PNF) .............................................. 21
3.2.2 Hierarchical netlist format (HNF) .................................. 21

3.3 Run-time environment .......................................................... 24
3.3.1 Shell Commands ........................................................... 24
3.3.2 Procedural Interface ....................................................... 28
3.3.3 Plotting the Evaluated Data .......................................... 30

3.4 Evaluation ........................................................................... 31
3.4.1 Full Evaluation ............................................................. 32
3.4.2 Partial Evaluation ......................................................... 32

3.5 Summary ............................................................................. 32
## Library Manager

4.1 Usage Context .................................................. 35

4.2 Programmable Architectures ...................................... 36

4.2.1 XC4000e Series Architecture ................................ 37

4.3 Library Characterization Procedure ............................... 39

4.4 Modules and Attributes .......................................... 40

4.4.1 ComponentLibrary type .................................... 41

4.5 Estimation Function ............................................ 42

4.6 Executing the model ............................................ 43

4.6.1 Through the API .......................................... 45

4.7 Results .......................................................... 47

4.8 Summary ....................................................... 47

## RTL Estimator

5.1 Estimation Methodology .......................................... 48

5.2 PDL+ Implementation ........................................... 50

5.2.1 Module : Component ....................................... 50

5.2.2 Module : Design ........................................... 52

5.3 Results .......................................................... 52

5.4 Summary ....................................................... 53

## Behavioral Estimator

6.1 High-level Synthesis: Estimator’s View .......................... 56

6.1.1 Scheduling .................................................. 57

6.1.2 Register Optimization ...................................... 58

6.1.3 Interconnect Generation ..................................... 58

6.1.4 Datapath-Controller Generation ............................. 58
6.2 Usage context of the Behavioral Estimator .............................................. 59
6.3 Building Blocks: Operation Graph .................................................. 59
6.4 Estimation Algorithms ................................................................. 61
   6.4.1 Evaluation Rules: Operation Module ........................................ 61
   6.4.2 Evaluation Rules: Behavior Module ......................................... 62
   6.4.3 Attributes: op_clbs .......................................................... 64
   6.4.4 Attribute: reg_clbs, ffs .................................................. 66
   6.4.5 Attribute: mux_clbs ....................................................... 67
   6.4.6 Attribute: clock_period .................................................. 68
6.5 Results ......................................................................................... 69
6.6 Summary ..................................................................................... 70

7 Partition Estimator ............................................................................ 72
   7.1 Partitioning for RCs: Estimator’s View ......................................... 73
   7.2 Usage Context ............................................................................ 74
      7.2.1 Target Architecture .......................................................... 74
   7.3 Estimation Methodology ............................................................... 76
      7.3.1 Area Penalty .................................................................... 76
      7.3.2 Interconnect Penalty ......................................................... 77
      7.3.3 Latency ............................................................................ 77
   7.4 PDL+ Implementation ................................................................. 78
      7.4.1 Architecture Model: rc.pdl ................................................ 79
      7.4.2 Application Model: dfg.pdl .............................................. 79
      7.4.3 Evaluate Model: bind.pdl ................................................ 81
   7.5 Evaluation Rules ....................................................................... 81
      7.5.1 Module: pe ..................................................................... 82
      7.5.2 Carrier: channel ............................................................... 82
10.3 The Power Model ............................................ 118
  10.3.1 Port: ioprt ........................................... 118
  10.3.2 Carrier: connect ...................................... 120
  10.3.3 Module: gate .......................................... 121
  10.3.4 Module: clb .......................................... 122
  10.3.5 Module: fpga .......................................... 122
  10.3.6 Functions Used ....................................... 125
10.4 Results .................................................. 127
10.5 Summary ................................................ 127

11 Reconfiguration Modeling 130
  11.1 Modeling Reconfiguration ............................ 130
      11.1.1 Modeling Using Modes ............................ 132
  11.2 Performance Model of the FPGA ...................... 133
      11.2.1 Evaluation ....................................... 135
      11.2.2 Usage Context .................................... 135
  11.3 Reconfigurable Coprocessor Model .................. 136
  11.4 Programmable Interconnect .......................... 136
  11.5 Time-Multiplexed FPGAs ......................... 139
  11.6 Summary ................................................ 140

12 Conclusions 143
  12.1 Modeling Guidelines .................................. 143
  12.2 Future Work ............................................ 144

A Characterized Component Data 145

Bibliography 152
List of Figures

2.1 PDL+ Design Component Declarations ............................... 9
2.2 Sample PDL+ Program ........................................ 10
2.3 Attribute of the Gate Module ..................................... 10
2.4 Synthesized and Inherited Attributes ............................... 11
2.5 Inherited over Synthesized ........................................ 11
3.1 The ARC Environment.............................................. 18
3.2 Allowed Dependencies among Attribute Types ...................... 20
3.3 Design File: PNF .................................................. 22
3.4 Design File: HNF .................................................. 23
3.5 Sample PDL+ program ............................................. 33
3.6 Attributes of the Components ..................................... 34
3.7 Evaluation Rules for the Attributes ................................ 34
3.8 Partial Evaluation of a Performance Model ........................ 34
4.1 Library Manager: Usage Context .................................. 36
4.2 XC4000e Series CLB .............................................. 38
4.3 XC4000e Series Interconnect ..................................... 38
4.4 Logic and Layout Synthesis ....................................... 39
4.5 The Component Module ........................................... 41
4.6 ShapeClbRec Record .............................................. 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7 LibraryComponent Datatype</td>
<td>42</td>
</tr>
<tr>
<td>4.8 The get_num_clbs Function</td>
<td>44</td>
</tr>
<tr>
<td>4.9 Sample Design File</td>
<td>45</td>
</tr>
<tr>
<td>4.10 Sample Design File with library Data - lib1.pnf</td>
<td>45</td>
</tr>
<tr>
<td>4.11 The Interface Header File</td>
<td>46</td>
</tr>
<tr>
<td>5.1 Typical RTL Netlist</td>
<td>49</td>
</tr>
<tr>
<td>5.2 Foreach Expression Over Discrete Range</td>
<td>50</td>
</tr>
<tr>
<td>5.3 Max Function</td>
<td>50</td>
</tr>
<tr>
<td>5.4 Component Module</td>
<td>51</td>
</tr>
<tr>
<td>5.5 Design Module</td>
<td>52</td>
</tr>
<tr>
<td>5.6 CLB Counts</td>
<td>53</td>
</tr>
<tr>
<td>5.7 FF Counts</td>
<td>54</td>
</tr>
<tr>
<td>5.8 Clock Speed</td>
<td>55</td>
</tr>
<tr>
<td>6.1 Operation Graph</td>
<td>57</td>
</tr>
<tr>
<td>6.2 Scheduled Operation Graph</td>
<td>58</td>
</tr>
<tr>
<td>6.3 Interaction between HLS System and Behavioral Estimator</td>
<td>60</td>
</tr>
<tr>
<td>6.4 PDL+ Representation of the Operation Graph</td>
<td>61</td>
</tr>
<tr>
<td>6.5 PDL+ Representation of a node and iport Declaration</td>
<td>61</td>
</tr>
<tr>
<td>6.6 Scheduled Operation Graph</td>
<td>62</td>
</tr>
<tr>
<td>6.7 Design File using Operation and Behavior Modules</td>
<td>63</td>
</tr>
<tr>
<td>6.8 Attributes and Evaluation Rules of the Operation Module</td>
<td>64</td>
</tr>
<tr>
<td>6.9 Behavior Module and its Attributes</td>
<td>65</td>
</tr>
<tr>
<td>6.10 User Defined Type: op_type</td>
<td>65</td>
</tr>
<tr>
<td>6.11 Number of Plus CLBs</td>
<td>66</td>
</tr>
<tr>
<td>6.12 Set of Plus Sizes</td>
<td>66</td>
</tr>
</tbody>
</table>
6.13 Register Evaluation Rule ........................................ 67
6.14 Average Size and Register Area ..................................... 67
6.15 Muxes for Plus Operator ............................................... 69
6.16 Typical Datapath Output of an HLS Tool ............................. 69
6.17 CLB Counts .......................................................... 70
6.18 FF Counts ........................................................... 71
6.19 Clock Speed .......................................................... 71

7.1 Task for an RC Compiler ............................................... 73
7.2 Usage Context of the Partition Estimator ............................. 75
7.3 WildForce Architecture ................................................. 75
7.4 Estimator Framework .................................................. 76
7.5 Architecture Model ..................................................... 79
7.6 Application Model ...................................................... 80
7.7 Evaluate Model ........................................................ 81
7.8 Module: pe ........................................................... 82
7.9 Carrier: channel ......................................................... 83
7.10 The Wildforce Board Architecture .................................... 84
7.11 Module: op ............................................................ 85
7.12 Port: opport .......................................................... 86
7.13 Carrier: flow .......................................................... 86
7.14 Module: dfg ........................................................... 87
7.15 The Evaluate Module .................................................... 88
7.16 Attribute: plus_area in each pe ....................................... 88
7.17 Mux Area in PE ......................................................... 89
7.18 Attribute:used_bits in channels ..................................... 89
7.19 Function: member ...................................................... 89
7.20 Attribute: wire_transfer_latency ........................................... 90
7.21 Attribute: memory_transfer_latency ........................................ 91
7.22 FFT1D Wildforce XC4013 ..................................................... 92
7.23 FIR16 Wildforce XC4013 ..................................................... 93
7.24 DCT4x4 Wildchild (XC4036) ................................................ 94
7.25 DCT4x4 WildFire (XC4025) ................................................ 94

8.1 Usage Context of Delay Estimator ......................................... 96
8.2 Sample net delay ............................................................... 97
8.3 Delay with fanout ............................................................. 98
8.4 The Toplevel design Module ............................................... 99
8.5 The clb Module ................................................................. 100
8.6 Components in the CLB Module ......................................... 101
8.7 Carrier wire ................................................................. 102
8.8 Function Definitions ......................................................... 103
8.9 Small Nets ................................................................. 104
8.10 Shuffle Exchanger: Critical Path ....................................... 104
8.11 TLC: Critical Path ......................................................... 105

9.1 Striped FPGA Architecture ................................................. 107
9.2 Hardware Virtualization ..................................................... 107
9.3 Estimator Framework ......................................................... 108
9.4 Physical and Virtual Stripe Modules .................................... 108
9.5 Eval Module ................................................................. 109
9.6 Function mod_int ........................................................... 110
9.7 Evaluation Rules ............................................................ 110
9.8 Evaluation Rules ............................................................ 111
11.13 Model of a Time-Multiplexed Logic Block
# List of Tables

4.1 Characterized Data for the Adder Macro ........................................ 40  
7.1 Latency and Area for Single FPGA ................................................. 91  
7.2 Estimated Vs Actual WildForce board ........................................... 93  
8.1 Delays Through the CLB ................................................................. 96  
8.2 Element Delay Values ................................................................. 97  
10.1 Capacitance Values of XC4003A Device ....................................... 117  
  A.1 Characterized data - I ............................................................... 146  
  A.2 Characterized data - II ............................................................... 147  
  A.3 Characterized data - III .............................................................. 148  
  A.4 Characterized data - IV .............................................................. 149  
  A.5 Characterized data - V .............................................................. 150  
  A.6 Characterized data - VI .............................................................. 151
Chapter 1

Introduction

Advances in semiconductor technology have led to reduction in device sizes and an increase in component densities. This growing complexity of designs has increased the need for efficient computer aided tools. The tools provide sophisticated environments for design entry, verification, and automatic hardware generation.

The primary focus of a designer is to ensure the design’s functional correctness. There are well established simulation techniques and environments that enable functional modeling. It is important for the designer to simulate and verify the functionality of the design before it is implemented on hardware. It is equally important to ensure that the design meets its performance requirements before it is committed to hardware. The term performance may refer to any non-functional attribute of the system. Some typical performance attributes of systems include clock frequency, power, latency, and area. The performance of a system is usually associated with a metric or a goal that has to be met by the resultant implementation and it is essential to be able to estimate these metrics earlier in the design process.

For instance, the performance of a microprocessor may be specified in terms of its clock frequency and the cycles per instruction, while the performance of an application specific integrated circuit (ASIC) may be a combination of its clock frequency, area, and the pins required. The metrics used to describe the performance of a system depend largely on the usage context and it is not realistic to have a universal tool that estimates all of the desired metrics. Hence, most of the available performance tools are specific to the metrics they are estimating. For example, a timing estimation tool models the delays through the design to predict the clock speed. A power estimation tool predicts the overall power dissipation of the hardware implementation. A typical design cycle requires the use of several such tools and considerable effort is spent in integrating these tools together. Hence a generic performance modeling environment where models for system performance can be specified, would be useful. A generic modeling environment allows designers to specify models that are not specific to a given metric or design flow.
1.1 Previous Work

Existing hardware description languages like VHDL and Verilog have minimal support to specify non-functional attributes and computations to evaluate these attributes [1]. Further, it would be advantageous to be able to develop models that are independent of their functional models and of specific designs. This would enable the reuse of the same model for a number of designs. Independence from functional models and specific designs is not possible using HDLs. Traditional programming languages like C/C++ are not suitable for specifying structural and temporal information that is essential to describe hardware properties. They are also not user friendly which is an important requirement of a modeling language.

The development of a modeling environment for generic performance analysis is a relatively new approach and most of the work has been done at the University of Cincinnati, by Vemuri et al [1]. Mandayam [2] identified the following properties that are essential for a performance modeling language. They are:

- Independence from functional models
- Independence from abstraction level
- Independence from specific designs
- Independence from design methodology
- Succinctness and efficiency
- User friendliness

Based on these specifications, Mandayam developed a framework based on attributed nodes-only graph grammars [3, 4] to model and evaluate performance attributes of systems. He developed a language called Performance Description Language (PDL), that can be used to model performance of a system using attributes and evaluation rules. PDL combines the hierarchical and structural design modeling features of hardware tools with the flexibility and power of formal language theory [2]. In the modeling framework, a performance model and a specific design netlist are used to create an executable performance model that is evaluated in a specific environment. PDL supports static and dynamic attribute types that may be used to describe non-temporal and temporal performance metrics. These attribute types are sufficient to model a wide range of metrics targeted to ASICs.

Adaptive or reconfigurable systems [5] are one of the most innovative technologies of the recent past. A reconfigurable device can be programmed to perform a specific task and reconfigured a number of times to perform different tasks if required. Reconfigurable systems have been used as logic emulators and ASIC prototyping platforms until recently. Now with increasing component densities and improved performance, these systems are also being used in real time, performance critical applications. Though they are inherently reconfigurable, implementing the design to check if it meets performance constraints increases the design time indefinitely. Implementing a design on these
devices requires complex design flows that are time consuming. Hence, modeling the performance of reconfigurable systems plays a major role in their design flow.

Walrath [6] enhanced PDL to target it to adaptive computing systems. The enhanced language is referred to as PDL+. Additional features, (such as quasi-dynamic attributes) that may be used to model these reconfigurable systems, were added to the original language. Walrath also developed an execution environment known as the analyzer for reconfigurable computers [7] to execute these models. The new environment provides more flexibility than the earlier execution system.

1.2 Motivation

Though PDL as a language had been developed a couple of years ago, performance modeling methodologies using PDL need to emerge. Most of the models written were test cases that demonstrated some selected features of the language. This was partly due to the continuous enhancements that were being made to the language and its environment. Another drawback was that a new language had to be learnt by potential users. The existing documentation has been in the form of language reference manuals and some basic examples, that were clearly not enough to motivate and guide a new user of the language.

1.3 Contributions

The main aim of this thesis has been to develop efficient modeling methodologies using PDL+ in the ARC environment. The modeling methodologies have been demonstrated with the help of case studies that model performance metrics of different systems. The main tasks involved in developing these case studies were:

- Identifying systems to be modeled
- Identifying metrics that define the performance of the system
- Developing appropriate estimation algorithms for these metrics
- Modeling the systems in PDL+
- Evaluating the models in the ARC environment

These tasks are documented in this thesis such that the models developed and the development procedure may guide future model developers. We have also included the models, sample design files, and guidelines to execute these models, in the accompanying diskette.
1.3.1 Identifying Systems

Identifying real life systems whose performance could be modeled was one of the challenging aspects of this thesis. The models were developed such that they demonstrate features of the language and also model robust systems with a reasonable degree of accuracy. Accuracy of models however depends on the amount of available information to the model developer. Sometimes it might be necessary for the model to interact with the system being modeled in order to obtain the necessary information. This interaction was possible for models of systems being developed at the University of Cincinnati and these models were eventually integrated with the existing design flows. Different modeling issues needed to be demonstrated for which, systems from a variety of areas were identified. The thesis attempts at attaining a balance between the two requirements of real life modeling and methodology development.

Systems were chosen such that they belonged to various abstraction levels in the design flow so that a variety of attributes would be required to aptly model them. Hence each model addresses different areas of VLSI systems and computer aided design (CAD), such as high level synthesis (HLS), component libraries, spatial partitioning, layout level delay estimation, reconfigurable processors, and power estimation. Due to the wide range of systems that were modeled we hope to demonstrate various aspects of the language and modeling methodologies.

1.3.2 Identifying Metrics

Once the systems were identified, a set of metrics that suitably define the performance of each of the systems was identified. These metrics depend on the usage context of the system to be modeled. For example, the performance metrics of a high level synthesis system that converts a behavioral description of a design to a register-transfer level (RTL) description could be the area occupied, and latency of the resultant RTL design. Each of the desired metrics were expressed as a computable attribute of the model components.

1.3.3 Estimation Algorithms

The estimation algorithms depend on the information available to the model. Estimation methods were associated with each of the desired metrics. Depending on the model these methods were either developed by us or adapted from existing systems. As with most estimation methods, we present one possible method of estimation for the chosen metrics.

The simplicity and user friendliness of PDL enables easy modification of the estimation techniques. This is an important feature because the estimation methods are usually refined continuously for better accuracy, or even changed to account for new features. For example, the delay estimation techniques for a microprocessor would differ over process technologies and modifying the existing methods becomes essential. The estimation methods used in the models are described in detail in each of the chapters.
1.3.4 Modeling in PDL+

The system, metrics, and estimation algorithms were mapped to the data types, attributes, and evaluation rules in PDL. This mapping process refers to the methodology development and the rest of the thesis describes this process in detail. We appropriately exploit different aspects of PDL+ such as, hierarchical and structural composition, lateral and hierarchical attribute propagation, static, dynamic, and quasi dynamic attribute types, and so on. Each of these properties will be highlighted through the remaining chapters. We hope that these models would help the reader in developing a thorough understanding of the ARC environment and the use of PDL+.

1.3.5 Evaluating the Models

The run-time environment of the ARC system provides two alternative methods of querying the executable model. Depending on the usage context of the model the appropriate method has been chosen. We provide description of how to execute each of the models in the accompanying diskette.

1.4 The Case Studies

We first provide a description of the ARC system and the PDL semantics, following which we present each of the eight case studies. These models address different issues of VLSI systems and computer aided design. System design using computer aided design tools can be classified on the basis of levels of abstraction. The three main levels of abstraction are the behavioral, structural, and the physical level. The case studies were chosen such that they belong to various levels of abstraction in the design flow. In the following subsections we provide a description of the models developed through the remaining chapters.

1.4.1 Library Manager

The library manager models the performance of a library of components that are typically used by most hardware synthesis systems. This model illustrates a simple usage of the PDL design components and the use of ARC’s application procedural interface (API). The model predicts the area, and delay attributes for components that belong to a component library. Since a component library is input to most synthesis systems, the library manager model is used by other estimators that model synthesis and partitioning systems in the later chapters.

1.4.2 RTL Estimator

The Register Transfer Level estimator predicts the area and clock speed of a register transfer netlist of components. An RTL netlist description of a design consists of instantiations of components, and is input to the estimator. The
model uses functions of the library manager to obtain the individual area and delay attributes for the components, using which it predicts the area and clock speed metrics for the entire design. This model corresponds to the structural level in the design flow.

1.4.3 Behavioral Estimator

The estimator for Field Programmable Gate Array (FPGA) behavioral compilers predicts the area and clock speed for design descriptions specified at the behavioral level. The estimator consists of rules that predict the synthesis optimizations that may be performed by a behavioral compiler. We illustrate aspects of hierarchical composition and the attribute propagation mechanism in PDL. The behavioral estimator has been interfaced with a high level synthesis system called Asserta [8], that uses these estimations to improve the quality of its designs.

1.4.4 Partition Estimator

The partition estimator is used in the context of multi-FPGA systems. The estimator models the performance of partitioning and synthesis tools for reconfigurable computers (RCs). The estimator interacts with a generic spatial partitioning tool and illustrates the use of performance models in the context of retargetable RC compilers. This model also belongs to the behavioral level in the design flow. The models described from chapter 4 through 7 model different stages in hardware synthesis and were integrated with the synthesis and partitioning systems developed at the University of Cincinnati.

1.4.5 FPGA Delay Estimator

The FPGA delay estimator models the performance of designs on an FPGA architecture, at the physical level. The model predicts the delays of routed nets of a Xilinx 4000 [9] series FPGA. The delay estimation methodology is based on the method developed by Kurdahi and Xu [10] for the Xilinx 4000 series FPGAs.

1.4.6 Striped FPGA Estimator

The striped FPGA estimator models the mapping of an application to a striped FPGA architecture [11]. Striped FPGAs allow simultaneous configuration and execution of designs on the physical fabric. Reconfiguration of the fabric enables the reuse of the hardware and hardware virtualization. We emulate the hardware virtualization on the architecture and predict the execution time, throughput, and the configuration schedule for designs mapped on to the stripe architecture. The model uses dynamic attributes to describe the performance.
1.4.7 FPGA Power Estimator

We describe a model that estimates the power dissipated on a Xilinx 4000 series FPGA device. This model belongs to the physical level in the design flow. A design netlist at the layout level is input to the estimator which predicts the overall power dissipated on the Xilinx architecture, for the given design. The estimation methodology is adapted from a method developed by Kusse et al [12].

1.4.8 Modeling Reconfiguration

We illustrate a methodology to model reconfiguration of systems. We describe the use of qdynamic attributes and the development of scalable performance models in PDL+ through this case study. We also describe the modeling of different FPGA architectures in this chapter.

1.5 Organization of the Thesis

The rest of the thesis explains the PDL model development procedure for each of the case studies and aims to be a guide for future model developers. The main emphasis is on the model development process. The material in the thesis is self contained in the sense that, all the background information required to understand the models has been included. The thesis provides good insights for model developers fluent with the language and for first time users, it introduces features of the language and demonstrates how to translate a problem into an efficient performance model. Hence the thesis would be useful for both beginners and readers familiar with the language. Also the readers are introduced to different issues in VLSI systems, such as synthesis, partitioning, delay estimation, and some methodologies to model them.

Chapters 2 and 3 provide an overview of the ARC environment and features of PDL+. They provide a good introduction to the modeling framework and describe the working of the ARC system.

Chapters 4 through 11 document the case studies that demonstrate the modeling methodologies. Each of these chapters describe the problem to be modeled, the usage context of the model, the development methodology, and results obtained from executing the model. The thesis is organized such that the earlier chapters model different aspects of hardware synthesis and belong to a higher level of abstraction. The models in the later chapters model performance attributes of the Xilinx 4000 series FPGAs.

Chapter 4 illustrates a simple usage of the data types in PDL+ and the steps involved in creating an executable model. The model developed in the chapter is a library manager that estimates the performance of components in a component library. Chapter 5 describes an area and clock speed estimator for register transfer level designs. Chapter 6 describes the development of a hierarchical model used to estimate the performance of a high level synthesis system. Chapter 7 describes a performance model of a spatial partitioning tool. Chapter 8 is an FPGA delay
estimator that illustrates structural modeling in PDL.

Chapters 4 through 8 model static systems with non-temporal attributes, unlike the chapters that follow. Chapter 9 models a striped reconfigurable processor and estimates the throughput and latency of designs to be executed on the processor. Chapter 10 describes a FPGA power model used to estimate the total power dissipated. These two models require temporal information and are examples of dynamic systems.

Chapter 11 models reconfiguration in a hypothetical reconfigurable architecture and uses quasi-dynamic attributes that are a part of the enhanced PDL+ language.

Each of these chapters address different modeling problems and describe a methodology to translate the problem to an executable performance model in the ARC environment.
Chapter 2

PDL+

We describe modeling techniques using Performance Description Language plus (PDL+) by presenting selected case studies through the remainder of this thesis. Before these models can be studied, it is necessary to understand the features of PDL+ and its execution environment. This chapter serves as a quick introduction to PDL+ for those new to the language and highlights the basic features of the language. The following chapter elaborates other aspects of the execution environment. A model of a system in PDL+ is described as a collection of components, attributes, and rules for computing these attributes. We describe the basic design components, attributes, attribute evaluation rules, and methods of attribute propagation in this chapter. This is followed by a description of the common expressions supported by the language. All of the concepts mentioned in this section have been used in developing the models that are described in the chapters to follow. A comprehensive description of all the features and syntactic conventions of the language can be found in the language reference manual [13].

2.0.1 Design Components

The system whose performance is to be modeled is visualized as a collection of the basic design components of PDL+. These are modules, carriers, and ports. Modules may contain any number of other modules, carriers, and ports. Hierarchy information within a system is captured by modules. A carrier is primarily used to propagate attribute values among various modules. They may not be contain other carriers, but can contain other ports. Ports. Carriers represent flow or transport type mechanisms within a design. Ports are primitive component types and cannot contain any other component. They are typically used to connect modules and carriers to each other. Figure 2.1 shows the declaration of the basic design components in PDL+.

```
module gate  carrier wire  port iport
end module;  end carrier;  end port;
```

Figure 2.1: PDL+ Design Component Declarations
Figure 2.2 shows a sample PDL+ model that consists of gate modules and ioport ports. The design module consists of a collection of gates and the interconnection among the gates is specified through the ports. Here the design

```plaintext
port ioport
end port;

module gate
ports
  inputs[] : ioport;
  output : ioport;
end module;

module design
ports
  inputs[] : ioport;
  outputs[] : ioport;
modules
  gates[] : gate;
end module;
```

Figure 2.2: Sample PDL+ Program

component containment is specified using sets. Arrays may also be used to specify the containment of components.

### 2.0.2 Attributes

Performance metrics that need to be evaluated are modeled as attributes associated to the design components. Each of the design components may be associated with any number of attributes. The attributes are classified in terms of their evaluation semantics and their data type. The three kinds of attributes based on evaluation semantics are - static, dynamic, and qdynamic. These are described in section 3.1.2.

An attribute may be of any of the allowed data types. The basic data types in PDL+ are integer, real, character, string, and boolean. They could also be any of the homogeneous data types like lists, sets, or arrays. The other data type that is supported by the language is the record type that is used to store heterogeneous data. The attributes may either have evaluation rules associated with them, or may be declared primitive. Primitive attributes are those that are supplied by the environment. Figure 2.3 shows an integer attribute area, associated with the gate module.

```plaintext
module gate
attributes
  area: int;
end module;
```

Figure 2.3: Attribute of the Gate Module
2.0.3 Evaluation Rules

Apart from representing the system composition and metrics associated with these components, the PDL+ program also consists of evaluation rules for these attributes. Unique evaluation rules need to be associated with the attributes. This is done by classifying the rules as being either synthesized or inherited.

In a hierarchical description, the parent component may read and assign values to its children’s attributes. When an attribute of a design component A is assigned within the component A, it is said to be a synthesized attribute. When a design component A contains another component B, and an attribute of component B is assigned from within A, then the attribute of B is inherited.

Synthesized and inherited attributes are automatically determined by the compiler. Figure 2.4 illustrates the difference between synthesized and inherited attributes. An attribute that is defined in the scope of its composition is a synthesized attribute [2]. The total\_area attribute is a synthesized attribute of the design module. The area attribute

```module design
modules
  gates{ } : gate;
attributes
  total\_area: int;
rules
  gates{ } . area = 5;  -- inherited
  total\_area = sum\_int(\text{foreach g : gate in gates \{g'area\}});  -- synthesized
end module;
```

Figure 2.4: Synthesized and Inherited Attributes

of the gate module is defined in the scope of the design module and is therefore an inherited attribute. If a component has both synthesized and inherited rules for an attribute, the inherited rule overrides the synthesized rule. Figure 2.5 illustrates this feature. Even though the gate module defines the area to be 1, the gates\{\} area in figure 2.4, are

```module gate
attributes
  area: int;
rules
  area = 1;
end module;
```

Figure 2.5: Inherited over Synthesized

assigned the value 5 due to the inherited rule, while the area attribute of gates that do not belong to the set of gates\{\} are assigned the value 1.
2.0.4 Attribute Propagation

The scope of attributes is restricted by the language semantics. Most of the evaluation rules require to access attributes of other design components. The attributes between components are propagated either hierarchically or laterally.

All attributes of a child component are visible to the parent. The child attributes can be propagated bottom-up to the parent, or a parent’s attribute can be propagated top-down to its children. This refers to hierarchical attribute propagation.

Lateral attribute propagation is achieved by carriers and ports. An attribute that models the longest path delay may be evaluated by the lateral propagation of delays from input port to the output port.

Depending on the metric to be evaluated and the rules of evaluation, these propagation methods are used appropriately in the models described in the following chapters.

2.1 Common Expressions and Statements

Evaluation rules for the attributes can be specified using functions, or through a variety of expressions that are supported by PDL+.

2.1.1 Expressions

The language supports mathematical, if-then-else and case expressions. PDL+ provides a foreach expression that is useful to iterate over homogeneous data types, or can be used to iterate over a range of values. The begin-end expression can also be used to specify evaluation rules. These are similar to in-line functions. They must end with a return statement that is assigned to the attribute. Several statements types can be specified within begin-end blocks. These statements are – variable declarations, if-then-else, case, while, foreach, and the return statement. Values cannot be assigned to attributes from within a begin-end expression. These expressions can be nested arbitrarily and result in concise rules that are also readable and easily modifiable.

2.1.2 Functions

When evaluation rules are specified in the body of functions, the return value of the function is assigned to the attribute. Similar to begin-end expressions, no attribute can be assigned from within the function body. Thus functions in PDL+ are side-effect free. Functions can also contain any of the statement types that can occur in a begin-end expression.
2.1.3 Built-in Functions

A number of built-in functions are supported by PDL+. They are useful to manipulate lists and sets. These functions can be used to append elements to lists, delete elements, append lists together, and return sizes of lists. The case studies also make use of some of the built-in functions provided by the language.

A combination of statements and expressions are used to specify evaluation rules of the different attributes of the models described in the remaining chapters.

2.2 Reconfiguration Modeling

FPGA devices can be configured to perform any number of tasks and can be reconfigured any number of times during their operation. Performance prediction in each of the modes of operation might require different evaluation rules. PDL+ provides a notation to specify how the performance is to be evaluated for each configuration. Thus, when a new design configuration is input, the performance model automatically adapts the evaluation rules to calculate the appropriate metrics.

Two built-in attributes called mode and trigger can be associated with the module, carrier, and port declarations. The mode attribute can be of any of the allowed data types while the trigger attribute is always a boolean data type. Both these attributes can be either primitive or have an evaluation rule associated with them. Additionally, a mode attribute can also be of any module, carrier, or port component type that is declared within the PDL+ program. This is a unique feature of the mode attribute because no other attribute type in PDL+ can be assigned to a component type.

The mode and trigger attributes are quasi-dynamic attribute types. Qdynamic attributes are useful to transfer information from one configuration to the other. When the trigger attribute is set to true, a model reconfigures and restarts the evaluation process. The evaluation rules for the attributes are specified in the PDL+ program for all the possible modes of the module. As long as the trigger value is false, the model continues to evaluate in its current configuration.

There are two different methods for defining and modeling the particular modes of an object. Both these methods are illustrated by the following two examples [13].

**Modeling with Modes : Method I**

```plaintext

type
gate_type : enum {andg, org, notg, ff};
end type;
```
port iopart
attributes
delay : real;
end port;
module gate : gate, type := andg
ports
inputs {} : iopart;
output : iopart;
attributes
max_delay : real;
delay : real;
max_ff : real;
rules
trigger = true;
max_delay = max( foreach x: iopart in inputs{x'delay});
delay = case curr mode of
   andg : 10.0;
   org : 8.0;
   notg : 5.0;
   ff : 7.0;
   others : 0.0;
end case;
max_ff = if(curr mode == ff) then
   max_delay + delay else 0.0
endif;
output'delay = if (curr mode == ff) then 0.0
   else delay + max_delay
endif;
end module;

In the first example, the attribute delay is assigned the appropriate value depending on the mode of the gate module. The delay values for each of the modes is specified in the rule definition. The declaration of the gate module assigns a default mode-type to the module. This type of a declaration specifies that the gate module represents a reconfigurable component.
Modeling with Modes : Method II

type
gate_type : enum {and, org, not, ff} ;
end type:
port ioport
attributes
  delay : real;
end port:
module gate : {and_clb, or_clb, not_clb, ff_clb} :=
    and_clb map{ attributes{ max_delay->max_delay,
                                 delay->delay,
                                 max_ff->max_ff,outdelay->outdelay } }
    ports
    inputs {} : ioport;
    output : ioport;
    attributes
    primitive gtype : gate_type;
    max_delay : real;
    delay : real;
    outdelay : real;
    max_ff : real;
    rules
    max_delay = max( foreach x:ioport in inputs {}'delay ) ;
    output'delay = outdelay;
end module;
module and_clb
attributes
  delay : real;
  outdelay : real;
  max_ff : real;
  max_delay : real;
rules
  delay = 10.0;
  max_ff = 0.0;
  outdelay = delay + max_delay;
end module;
module or_clb
attributes
  delay : real;
  outdelay : real;
  max_ff : real;

max_delay : real;

rules
  delay = 8.0;
  max_ff = 0.0;
  outdelay = delay + max_delay;
end module;

module not_clb
attributes
  delay : real;
  outdelay : real;
  max_ff : real;
  max_delay : real;

rules
  delay = 5.0;
  max_ff = 0.0;
  outdelay = delay + max_delay;
end module;

module ff_clb
attributes
  delay : real;
  outdelay : real;
  max_ff : real;
  max_delay : real;

rules
  delay = 7.0;
  max_ff = max_delay + delay;
  outdelay = 0.0;
end module;

In the second example, the *mode-type* declaration allows the user to declare the built-in attribute *mode* to be a component type. Here all the possible component types are listed as a set \{*and*clb, *or*clb, *not*clb, *ff*clb\}. Additionally, the attributes are *mapped* to the appropriate values depending on the mode of the component. This is done at the declaration of the gate module using the built-in *map* function.

Chapter 11 describes a model that uses the mode and trigger attributes to model performance attributes of an FPGA architecture.
2.3 Summary

This chapter provides the necessary background information required to understand the features of PDL+. The basic design components, attribute types and notation of the language were highlighted. Some of the features of PDL+ regarding attribute propagation, and containment were described. The features of the language described in this chapter will be referred to wherever appropriate throughout the remainder of the thesis.

The next chapter describes the execution environment for models specified in PDL+. 
Chapter 3

The ARC System

The execution environment for PDL+ is known as the analyzer for reconfigurable computers (RCs). The language together with the execution engine constitutes the Analysis for Reconfigurable Computers (ARC) system. It is so called because the original language (PDL) and its execution engine were enhanced to enable modeling certain aspects of reconfigurable devices [6]. In this chapter we describe the components of the ARC system.

The ARC system is a software environment in which models of performance can be described, executed, and queried. Figure 3.1 shows a simplified view of the ARC environment. The compiler and evaluator are the core of the execution engine of the ARC system. The inputs to the compiler are a performance model and a design netlist. The compiler creates the executable performance model that is input to the evaluator. The evaluator accepts inputs for the primitive attributes and evaluates the executable model. The results of the evaluation may be queried through the shell or through the application procedural interface. The following sections describe the working of the compiler and evaluator of the ARC system.

Figure 3.1: The ARC Environment.
3.1 Compiler and Evaluator

Section 1.1 identifies features of a performance modeling environment. One of these properties is *independence from specific designs*. Hence in the ARC system, performance description is separated from the design netlist. This enables the reuse of the same performance model for a number of designs. The separation of design netlists from model specifications, requires special features for the compiler of the language and will be addressed in section 3.1.

The PDL+ descriptions of a system’s performance are referred to as *performance models*. The performance model and a specific design netlist are input to the compiler that constructs the *executable performance model* for the specific design. A performance model of a system consists of construction rules that describe the system composition, attributes, and the evaluation rules associated with these attributes. The design netlist represents an *instance* of the system that needs to be evaluated. Hence, there has to be a correlation between the performance model and the design specification.

Figure 3.1 shows the two inputs to the compiler. The compiler accepts designs specified in either the hierarchical netlist format (HNF) [6] or the PDL netlist format (PNF) [14]. The hierarchical format is more *memory efficient* and is suited for regular design descriptions, while the flattened format that is simpler to use, is suitable for smaller designs. We provide a description of these two design specification formats in section 3.2. We now briefly describe the compilation process.

3.1.1 Compilation

The compilation process consists of two stages. First the performance model (PDL+ program) is parsed and semantically analyzed. This is followed by the parsing and semantic analysis of the design file. The output of the compiler is a set of C++ functions that are linked with the execution environment. The compiler checks for the conformance of the design file with the model. Formally we can say that the PDL+ program represents a *grammar* and the compiler checks if the design is an element in the *language* of the grammar. The compiler then associates appropriate attributes to elements of the design and determines dependencies between different attributes. The output of the compilation stage is the *executable performance model*.

The main task during compilation is *dependency generation* between different attribute instances. This is necessary because unlike traditional programming languages, the order in which rules appear in the PDL+ program does not affect the sequence of evaluation. The evaluation order is automatically inferred by the compiler after performing a total dependency analysis for all attributes associated with components in the design.

In order to generate unambiguous dependencies there are certain restrictions on the evaluation rules and on the scope of different attributes. One such restriction is that an attribute cannot depend on itself, as this would result in a cyclic dependency. A formal description of the compilation process can be found in Mandayam’s thesis [2].
3.1.2 Evaluation

After determining the evaluation order, primitive attributes are supplied to the compiled model from the ARC runtime environment. The evaluator then executes the compiled performance model with these initial values, in the order determined by the earlier phase.

Depending on the type (static, dynamic, or quasi-dynamic) of an attribute, different evaluation semantics are associated by the evaluator. This also restricts the dependency between attributes of different types. Figure 3.2 shows the allowed dependencies. A qdynamic attribute can depend on other qdynamic, dynamic, and static attributes. A dynamic attribute can depend on other dynamic, and static attributes, while a static attribute can only depend on other static attributes.

Evaluation of attributes proceeds in cycles. Static attributes are evaluated once, while dynamic attributes are evaluated every cycle. The number of evaluation cycles can be controlled by the user. In the previous section we mentioned that an attribute cannot depend on itself as this would result in a cyclic dependency. However, in the case of dynamic or qdynamic attributes, an attribute can depend on its own value from any of its previous evaluation cycles. Also static and dynamic attributes can depend on qdynamic attribute values from the previous evaluation cycle. This is indicated by the dotted line in figure 3.2.

A stream of input values is assigned to primitive dynamic attributes. Each time a dynamic attribute is assigned a new value, all dynamic attributes of the model are evaluated. The evaluation rules of dynamic attributes however remain the same over all the evaluation cycles.

Quasi-dynamic or qdynamic attributes can be considered to have reconfigurable evaluation rules. The rules associated with these attributes can change dynamically during evaluation. All static and dynamic attributes are re-evaluated during a qdynamic evaluation cycle. Qdynamic attributes are useful to transfer information from one configuration to the other. When the trigger attribute is set to true, a model reconfigures and restarts the evaluation process. As long as it is false, the model continues to evaluate in its current configuration. The models described in the following chapters make use of a combination of attribute types and illustrate these different evaluation cycles.

The evaluator has two modes of operation that can be chosen by the user. These modes are full evaluation (default mode) and partial evaluation. The evaluation modes are described in section 3.4.
3.2 Design File Specification

A performance model specified in PDL+, and a design netlist are required to create an executable performance model of a system. The ARC system supports two design netlist specification formats. These are – the older flattened netlist specification (PDL+ netlist format, or PNF), and the newer hierarchical netlist format (HNF). We describe the basic features of these formats in the following subsections. More detailed descriptions can be found in the respective design specification manuals [15] [14]. The design files represent instances of a system that need to be evaluated using the rules specified in the performance model. The compiler matches the object instances in the netlist file with the corresponding object definitions in the PDL+ program.

3.2.1 PDL netlist format (PNF)

In the PNF netlist format the hierarchy and connectivity information of design components are specified by explicitly instantiating each component in the design. Consider the PDL+ program shown in figure 2.2. A design file for the model specified in the PNF format is shown in figure 3.3. The port connections are set usually at the end of the design file using the \textit{net} keyword.

All ports that are connected in the design file are merged into one \textit{logical} port. The connected ports are therefore assigned the same attribute values. Some design specifications might lead to more than one value driving a given port. In this case, an error is generated unless a \textit{resolution function} is defined. This resolution function is used to calculate a single value for an attribute having multiple values. We illustrate the use of resolution functions through the case study in chapter 10. Also sometimes it is likely that a port may not be connected and its attributes are assigned the default value, \textit{unknown}.

Attribute values may be set from within the design file. It is a good idea to set \textit{design specific} values from within the file. Dynamic attribute types require stream of values, and it is not possible to set a stream of values from within a design file. All attribute types that are valid in the PDL+ program are also valid in the netlist specification.

The PNF netlist format is suitable to specify small designs as \textit{each} module and carrier object is instantiated and this would require a large amount of memory. Also this format does not provide a convenient method of representing regular and repetitive design structures.

3.2.2 Hierarchical netlist format (HNF)

The hierarchical netlist format is more memory efficient than the flattened PNF netlist format. It is specifically useful to describe regular and repetitive designs. Flattened designs can also be specified in the HNF netlist file.

Design files in the hierarchical netlist format consist of the \textit{cell} notation. This allows for repetitive subsystems of a design to be defined, thereby eliminating the need to repeatedly instantiate the same subsystem. A cell can be a
module design = d1
  ports
    inputs = {i1,i2,i3,i4};
    outputs = {o1};
  modules
    gates = {g1,g2,g3};
end module;

module gate = g1
  ports
    inputs = {i1,i2};
    outputs = {o1};
end module;

module gate = g2
  ports
    inputs = {i1,i2};
    outputs = {o1};
end module;

module gate = g3
  ports
    inputs = {i1,i2};
    outputs = {o1};
end module;

net d1.i1, g1.i1;
net d1.i2, g1.i2;
net d1.i3, g2.i1;
net d1.i4, g2.i2;
net g1.o1, g3.i1;
net g2.o1, g3.i2;
net g3.o1, d1.i1;

Figure 3.3: Design File: PNF
cell module gate = gengate
composition
    in1: ioport;
    in2: ioport;
    o1: ioport;
sets
    inputs = {in1, in2};
    outputs = {o1};
end module;

cell module design = cd(loop: int:=3, loop1: int:=4)
composition
    foreach i in 1 to loop {
        inp[i]: ioport;
        inputs += inp[i];
    };
    foreach i in 1 to loop1 {
        g[i]: gengate;
        gates += g[i];
    };
sets
    outputs = {o1};
end module;

d1: cd(4,3);

Figure 3.4: Design File: HNF

module, carrier, or a port cell. This when used effectively produces memory efficient design file representations. A cell definition is not an instantiation of the component and cells have to be appropriately instantiated.

Figure 3.4 shows a possible hierarchical representation of the design file whose flattened description is shown in figure 3.3. The d1 instance of the cell cd consists of 4 input ports and 3 gates. Each of the gates has 2 inputs and 1 output port. We have not shown the port connections. Designs with different number of gates can be instantiated by changing the parameters of the d1 cell instance.

In the composition part in a cell definition design components of other cells, and design objects are instantiated. There are four statement types that can appear in a composition section. These are cell/object declaration, set addition, foreach generate loop, and port assignment. We illustrate the first three statements in figure 3.4. All objects declared within a cell are visible only within the entire cell. The ports that are declared within a cell are however visible outside the cells.

The sets part of the cell definition may be used to compose the instantiated components. Set addition is also possible from with the foreach generate loops. These aspects have also been illustrated in figure 3.4.

A complete description of the HNF syntax can be found in the design specification user’s guide [15].
We summarize some of the useful features of HNF design specifications below.

- Set default values for attributes.
- Instantiate required number of components using *foreach* statements.
- Can reuse cell definitions over different design files.
- Files are smaller than the flattened specifications.

We have used a combination of these design specification formats in the case studies and these files are included in the diskette.

### 3.3 Run-time environment

The ARC run-time environment supports interaction with the PDL+ evaluation engine either through the *shell* or through its *application procedural interface* (API). The shell is best suited for manual interaction while the API is useful for interfacing the model with other Electronic Design Automation (EDA) tools. We provide an overview of both these methods of interaction in the following subsections.

#### 3.3.1 Shell Commands

The shell is the heart of the ARC system that executes various commands to generate, evaluate, and display the results of the executable models. There are two primary modes of operation - interactive, and batch. In the interactive mode, the commands are entered at the *Arc* command prompt. The commands are specified in a file for the batch mode and these are executed one at a time in the order that they appear in the file. The accompanying diskette includes command files for executing each of the models. The procedure to create an executable model from within the shell is shown below -

- load program `<pdlfile>`
- load design `<hnf/pnffile>`
- set value `<componentinstance> <attributename> <value>`
- evaluate
- show value *
**load**

The *load* command is used to read a PDL+ program, design, or a previously saved model file. When a load program command is issued, the file is parsed and error checked for correctness. The load program command has to be issued before a load design command. A *load design* will load the design file successfully if the file matches the current PDL+ program that is loaded.

**evaluate**

The *evaluate* command is used to execute an executable performance model that resides in the ARC system.

When the *evaluate* command is issued without a value, the model evaluates all static attributes, then dynamic attributes, and finally quasi-dynamic attributes.

The *optional-value* applies only to dynamic attributes. It controls the number of evaluation cycles performed within one *evaluate* command. Dynamic attributes contain streams of values, where each value in a stream corresponds to one particular evaluation cycle. Each new evaluation cycle reads the next value from an input stream and generates new output values that are also stored in queues.

**save model**

This command is used to save executable models that are created within the ARC system. These saved models can be loaded into the ARC system using the *load model* command.

**set**

The *set value* command is used to set attribute data for the components. Primitive attribute values are also set using this command. Streams of values for dynamic attributes are supplied by a list of values that are comma separated. This is shown in the command file for the power model in chapter 10.

The *set control* is used to control the number of evaluation cycles for a dynamic attribute. A boolean expression is the argument for this command and the evaluation continues as long as the expression is true. An application of this is shown in the striped FPGA estimator in chapter 9.

The *set output* command is used to specify a file to direct all display information that is generated by ARC. This is useful while executing a model in the batch mode. The *set memsave (on | off)* command stores all values of any evaluation for all attributes, if it is set to off. Thus, when memory saving is on, the evaluator will only save the current and new values of all dynamic and qdynamic attributes. Thus, on the next evaluation cycle, all current values are deleted, the previous new value become the current value and new value are evaluated. For very large models, it
might be useful to use the memsave on option.

The set partial command is used to toggle partial evaluation of an executable performance model during evaluation. When partial evaluation is off, all evaluation of a performance model is full evaluation. That is, if a value for an attribute is unknown or does not have a value, all expression relying on that attribute are also unknown and are not evaluated. Thus, the results will be printed as **unknown**.

However, when partial evaluation is on, a model can be partially evaluated. That is, all attributes which have no value are unknown. However, any expression relying on unknown attributes are still evaluated and all known data within the expression is evaluated. Thus a model can be partially evaluated.

**show**

The show commands are used to display information contained in the executable model to the screen. The show value command is used to display the value of particular attributes contained within an executable performance model. For example,

\[ Arc> \text{show value on} * \] shows the value of all attributes in the entire model. Another example,

\[ Arc> \text{show value gate1} * \]

shows the value of all attributes in the gate1 object.

The show flattened command is identical to the show value command except that it flattens all expressions. With partial evaluation, it is possible to have expressions containing references to other incomplete expressions. With this command, all expressions are flattened; that is all attribute references are replaced with the corresponding expressions thus eliminating the attribute reference directly.

**extract**

The extract commands are used to generate special files that are used by tools external to the ARC system. These commands typically generate files in a pre-defined format that can be used by other tools. The extract template command is used to generate a file that contains a concise definition of the various objects and attributes defined in a program specification.

**rewind**

These commands are used when a performance model was only partially evaluated. They provide a mechanism for rewind the evaluation cycles to previous values and allowing the user to modify values and re-evaluate the model. These commands are primarily intended to be used only when a model has been partially evaluated although they
can be used on models that have been fully evaluated. The *rewind evaluate* command is used to move back to previous evaluation cycles within a performance model. This command only applies to the evaluation cycle of dynamic attributes within the current configuration mode.

The *rewind evaluate* command is used to move back to previous configuration modes within a performance model. This command rewinds the evaluation back to the initial evaluation cycle within the specified configuration mode.

**run**

The *run file* command is used to execute other files within the ARC system. This gives the user the flexibility of performing batch mode type operations within an interactive shell.

**stats**

This command is used to display the statistics of the current executable performance model. It displays the number of modules, carriers, ports, total number of attributes, and total number of components.

**timer**

The *timer* command is used to control an internal timer within the command shell. The syntax for the command is:

```
Arc> timer (start | stop | show | elapsed)
```

The timer is initially stopped when the command shell starts. Via this command, it can be started and stopped. Additionally, the current value of the timer can be shown using the *timer show* command. The *timer elapsed* command can be used to show the time of the timer while it is running so that it does not have to be stopped. The *time show* command does not work on a running timer. This command shows the time the timer was started when used while the timer is running.

**reset**

The *reset* command is used to restore an executable model to its initial state that it was in immediately after it was loaded. Thus after some number of evaluations a model can be reset to initial values using this command. All data that was generated during the evaluations is deleted. Thus, this command is different from the *rewind* command because it does delete all data.
**quit and exit**

These commands cause the ARC system to stop executing. In the interactive mode, ARC shuts down and returns control back to the host environment. In batch mode, these commands cause ARC to stop executing commands in the batch file.

A complete list of the shell commands can be found in the ARC system user’s guide [16]. The other commands that are used to generate plots from the evaluated data are described in section 3.3.3.

### 3.3.2 Procedural Interface

The API is an interface to the ARC system that allows the performance models to interact with other software and CAD tools. It contains an include file and a library that can be linked with other software tools. The API consists of a set of functions that allow for a model to be loaded, compiled, evaluated, and queried.

The user must specify the location of the header file using the `-I` option of the compiler. To use the libraries, the user must use the `-L` option to specify the location of the include files and `-lARC.gnu` or `-lARC.sunpro` to link in the libraries. An example of a link would appear as:

```bash
g++ -o cad main.o test.o clb.o -I/home/jwalrath/lib/arc/include \
   -L/home/jwalrath/lib/arc/lib/ -lm -lARC.gnu
```

**The API**

There is one class definition called `ArcSystem`. This is an entire ARC object that includes compiling, evaluation, etc. All interaction is performed with this object.

The constructor for `ArcSystem` has an optional integer argument. If this argument is 1, then ARC will attach some functions to the signal handlers to report special messages in the case of a segmentation fault and a bus error. If 0, these signals will not be modified.

There are two ways to use the ARC API. All information regarding design specification can be passed using files or direct memory API manipulation. That is, the design can be given as an input file or the user may directly call the API to create and manipulate objects in the design. Most of the operations that can be performed through the shell can also be performed through the API functions. For example, if we declare and object of the ArcSystem `arc` in the C++ program. We can load a PDL+ model, and a design file as shown below:

```c++
arc.LoadPerfModelSpec(const char *file);
arc.LoadDesignSpec(const char *file);
```
The API also provides functions for low-level manipulations such as instantiating design objects and for providing the connectivity information. When a design file is loaded, the model is automatically compiled, whereas if the design components are provided through the API, the user must control when a design has been completely given by appropriately calling the `CompileModel` function shown below.

```c
arc.CompileModel(bool partial, bool memsave);
```

All objects in a design are represented using the `ArcObject`. This is a class declared in the API header file. The user instantiates design objects and a handle to this object is given to the user as the `ArcObject`. The user can create objects, add items to the composition sets, and even provide the connectivity information. All objects including modules, carriers, ports, and HNF components are referred to as the `ArcObject`.

The functions to create design objects, add them to a composition are shown below.

```c
int arc.CreateDesignObject(const char *name, const char *type, ArcObject &obj)
```

```c
int arc.AddItemToComposition(ArcObject &obj, const char *set, ArcObject &item)
```

The functions `AddPort`, and `ConnectPorts` are used to associated ports to the objects and connect them together. These function declarations are shown below.

```c
int arc.AddPort(ArcObject &obj, const char *name, const char *set, ArcObject &port)
```

```c
int arc.ConnectPorts(ArcObject &port1, ArcObject &port2)
```

The attribute values for the objects can be set using the `SetAttributeValue` functions and the evaluated attribute values can be queried using the `GetAttrData` function. These are shown below. The attribute values and the evaluated attributes are passed as string constants. The evaluated attributes are objects of the `ostrstream` class.

```c
int arc.GetAttrData(const char *obj, const char *name, const char *attrname, ostrstream &str, int spc, bool flat)
```

```c
int arc.SetAttributeValue(const char *obj, const char *name, const char *attrname, const char *expr)
```

There are other functions that enable us to execute the shell commands such as `evaluate`, `set partial evaluation`, and also to perform some error checking.

Some of the models described in the following chapters were integrated with other CAD tools through the API using a number of these functions.

A complete listing of the various API functions and their declarations can be found in the ARC API user’s guide [17].
3.3.3 Plotting the Evaluated Data

The ARC system provides an interface with visualization tools like gnuplot. This is useful to perform trade-off analysis and study the trends of the evaluated attributes. The following commands are used to generate plot files from within the ARC shell.

**vary**

This command is used to setup a variable within the command shell that can be used to assign values to attributes. The syntax for the command is:

```
Arc> vary <variable-name> <low-limit> <upper-limit> <step>
```

A variable is declared with the name given by `variable-name`. This variable can then be used in any `set value` command to set the value of an attribute. It is used primarily for the purposes of plotting. Thus, the value of the variable will increment from the `low-limit` to the `upper-limit` incremented by the value in `step`.

Any number of variables can be declared within the command shell using this command. However, the variable is not activated or incremented until `repeat` and `done` statements are executed. Thus, the `vary` command is used to declare all the variables that will be used for generating a plot.

**repeat**

This command initializes the plot generation sequence of the command shell. It causes all variables declared with the `vary` statement to be initialized to their `low-limit`.

Once a repeat statement is encountered the shell enters a special mode. Every command that comes after the `repeat` clause is stored in a queue. The commands are not actually executed. Execution of the commands does not begin until a `done` statement is encountered.

The number of repeats are determined by the number of `vary` statements that were defined. That is, if there are 3 variables, all statements are repeated until each variable has been incremented to its upper limit. Thus, this mechanism can be used to generate any dimension of data for plotting purposes.

**done**

This command will cause all statements appearing after the last `repeat` clause to be executed. Thus, it delimits the end of all the statements to be executed within a `repeat/done` block.
**set plotfile**

The *set plotfile* command is used to create plotfiles for visualization during evaluation of a model. This file is only used by command that generate plots (i.e., the repeat and done commands).

During evaluation a plot of values can be generated by the command shell. This plot is also stored in a file given by the name that is an argument to the *plotfile* command.

**show plotfile**

The *show plotfile* command is used to call Gnuplot and generate a graph of the data collected in a plotfile. The syntax for the command is:

```Arc> show value <plotfile-name>```

The *plotfile-name* is the name of a plot file created with the command *set plotfile*. It can be any plotfile previous generated with the command shell.

**watch**

This command is used to write data to the plotfile specified by the *set plotfile* command. The syntax for the command is:

```Arc> watch <obj-name> <attr-name> (, <obj-name><attr-name>)*```

When this command is executed, all the attribute *attr-name* for the object *obj-name* is printed to the plotfile. Once a value has been printed, a linefeed is printed to the plotfile. Thus, in order to make three columns of data in the plotfile, there must be three attributes listed on the watch line.

### 3.4 Evaluation

Section 3.1.2 provides an overview of the evaluator in the ARC system. There are two modes of evaluation that are supported by the ARC system. These are the default *full* evaluation mode and the *partial* evaluation mode. Consider the performance model shown in figure 3.5.

Prior to evaluation, each attribute is assigned a status value of *known* or *unknown*. Initially all attributes including primitive attributes are assigned the status of unknown. Then during the evaluation process, an attribute is assigned the status of known when its corresponding evaluation rule evaluates to a value [18].
3.4.1 Full Evaluation

In the default full evaluation mode if an attribute depends on other attributes whose value is not known at the time of evaluation, the attributes remain unknown. For example, if the time attribute for the inputs of the design module in figure 3.5 are not assigned a value, the attributes that depend on the primary input time attributes would all remain unknown at the end of evaluation.

Consider an input design shown in figure 3.6 [6].

Figure 3.7 shows the evaluation rules for each of these attributes.

3.4.2 Partial Evaluation

On the other hand in the partial evaluation mode, the attributes are evaluated to their corresponding expressions. For example, the rule \( a = b/d \) cannot be fully evaluated unless the attributes \( b \) and \( d \) are known. In the partial evaluation mode if say attribute \( d \) is not known at the time of evaluation and \( b \) is assigned to 5, then \( a \) is assigned the partially evaluated expression \( 5/d \). Figure 3.8 shows partially evaluated attribute rules [6].

There are several benefits of partial evaluation. These are reduced data collection times for visualization, performance model reduction, model transformation, and debugging the model. A detailed description of the partial evaluation features of the ARC system can be found in [6].

An interesting application of the symbolic evaluation feature of the ARC system is described in Radhakrishnan’s thesis [19].

All the models described in this thesis use the full evaluation mode. An in-depth description of the evaluation procedure and the algorithms implemented in the evaluator can be found in Walrath’s thesis [6].

3.5 Summary

This chapter provides the necessary background information required to understand the execution environment aspects of the ARC system. The process of creating an executable performance model from a PDL+ description and a design netlist was described.

The next chapter describes the first of the eight case studies. The model of a component library was developed using PDL+. The methodology and usage context of the model will be described in detail in the following chapter.
port ioport
attributes
time : real;
end port;

carriers wire
ports
inputs[] : ioport;
outputs[] : ioport;
attributes
primitive delay : real;
rules
outputs[]’time = max_real(foreach i : ioport in inputs{’time}) + delay;
end carrier;

module gate
ports
inputs[] : ioport;
output : ioport;
attributes
primitive delay : real;
rules
output’delay = max_real(foreach i : ioport in inputs{i’time}) + delay;
end module;

module design
ports
inputs[] : ioport;
outputs[] : ioport;
carriers
wires[] : wire;
modules
gates[] : gate;
attributes
max_delay = max_real(foreach o : ioport in outputs{o’time});
end module;

Figure 3.5: Sample PDL+ program
Figure 3.6: Attributes of the Components

\[ g_{1}'time = g_{1}'delay + \max(n_{1}'time, n_{2}'time) \]
\[ g_{2}'time = g_{2}'delay + \max(n_{3}'time, n_{4}'time) \]
\[ n_{5}'time = g_{1}'time + n_{5}'delay \]
\[ n_{6}'time = g_{2}'time + n_{6}'delay \]
\[ g_{3}'time = g_{3}'delay + \max(n_{3}'time, n_{4}'time) \]
\[ g_{4}'time = g_{4}'delay + \max(n_{5}'time, n_{6}'time) \]
\[ n_{7}'time = g_{3}'time + n_{7}'delay \]
\[ n_{8}'time = g_{4}'time + n_{8}'delay \]

Figure 3.7: Evaluation Rules for the Attributes

Figure 3.8: Partial Evaluation of a Performance Model
Chapter 4

Library Manager

In this chapter we describe the development of a model of a *library of components*. The model will be referred to as the Library Manager. The following sections describe the usage context and model development methodology of the library manager. Apart from describing the model, the chapter provides most of the background information that will also be useful for the remaining chapters.

The *Library Manager* model illustrates a simple usage of modules, attributes and their evaluation rules. It also shows how to define new data types in PDL+ and the use of functions for attribute evaluation. In the previous chapter we introduced different attribute types in PDL+, based on their evaluation semantics. These are static, dynamic, and qdynamic types. The Library Manager is an example of a static system and has static attributes.

Following the description of the model and its attributes, is a section that details how to interface a compiled PDL+ model with a C++ program using the Application Procedural Interface. The API is a part of the run-time environment provided by the ARC system. This acts as an interface that can be linked with an external application program.

The first two sections provide a brief introduction to library based synthesis and describe the usage context of the model. This is followed by a description of the PDL+ model.

### 4.1 Usage Context

A library of components is used in most hardware synthesis systems. Before we describe the relevance of a component library, we introduce traditional synthesis systems. *Synthesis* is the process of converting a design description from a higher level of abstraction to a lower level. The initial level of abstraction of the design specification differentiates the synthesis processes. For example, the translation of a *behavioral* level description to a *Register-Transfer Level* (RTL) design is called *High level synthesis* (HLS) [20]; while the translation of an *RTL design* to an optimized
gate level description is called logic synthesis [21].

The component library model developed in this chapter is relevant to high level synthesis systems. HLS tools automate the translation of behavioral descriptions to RTL implementations. The advantages of using these tools are manifold and can be found in the literature. Most of these tools perform library based synthesis, where a component library is input to the HLS tool. The output of the HLS process is an RTL design that consists of instantiations of components from the input component library.

The core of the synthesis process is however independent of the library of components that is available to it. But design decisions and performance of the overall design greatly depend on the performance data of these components. It is therefore useful to integrate the synthesis system with a component library performance estimator. This ensures that any change in the component library does not affect the core synthesis tool.

Figure 4.1 illustrates the usage context of the Library Manager. The performance metrics of interest are the areas and delays of each of the components. The synthesis tool supplies a component name and a bitwidth to the library manager which then returns the delay and estimated resource count. The library manager predicts the performance metrics of the components using data from pre-characterized components. This performance data depends on the target architecture of the components. The Library Manager models the performance of components targeted for the Xilinx 4000 series devices. Most of the models developed through the rest of the thesis are targeted to the Xilinx 4000 series devices. The following section briefly outlines the architecture of the XC4000 family of devices and introduces the basic terminology used in the context of these devices.

### 4.2 Programmable Architectures

A Field Programmable Gate Array (FPGA) device has a fixed amount of uncommitted logic and interconnect resources that are programmed to realize logic functions. Xilinx 4000 series, and Altera Flex 10K are examples of some FPGA devices. The two main performance metrics for components targeted to these devices are the area occupied and the delay through the component. Since the number of resources available on an FPGA are limited, it
is important for the designer (or synthesis tool) to know the area occupied by each of the components in order to predict the area of the entire RTL implementation. Based on this information, alternate designs may be explored by the HLS tool.

A basic programmable gate array (PGA) device is made of logic cells and interconnect resources that may be programmed to realize logic functions. The PGA architecture is characterized into two styles, cellular-based and island-based. Cellular based approach consists of simple logic cells (typically realizing 2 input functions) and an interconnect architecture that relies heavily on neighbor to neighbor communication. This is a fine grain approach to computation. Examples of these devices are XC6200 [22], and Atmel AT6000 [23].

The XC4000 architecture is island-based, consisting of an array of complex logic blocks allowing more computation per logic block. The interconnect structure is a set of general purpose wiring used to connect these arrays of logic blocks.

### 4.2.1 XC4000e Series Architecture

The basic logic blocks in a Xilinx FPGA are called Configurable Logic Blocks (CLBs). Each CLB consists of two four input look-up tables (4-LUTs) and a three input look-up table (3-LUT). Figure 4.2 shows the CLB architecture of an XC4000 series device. The two 4-LUTs can realize any 4 input function that may be passed through the 3-LUT or connected directly to the output of the CLB. The CLB outputs may also be registered using the two flip flops, shown in the figure. The CLB is configured by setting the memory bits that control the internal multiplexers and by programming the LUTs to perform the desired functions.

The Xilinx device has segmented interconnect architecture. Figure 4.3 shows a detailed view of the interconnect resources available on the PGA device. There is an array of switch boxes that either connect shorter segments to form longer segments or enable corner turns during routing. The interconnect resources are classified based on their length. Most of the segments either span one CLB pitch (single segment) or two CLB pitches (double segment). The double segments bypass a switch matrix and can be connected at alternate switch matrices. Another type of segment is the long line that traverses the entire CLB array. These long lines are sometimes halved into two segments if the entire length is not required. All the interconnect segments allow bidirectional signal flow.

The XC4000 family of devices have the same architecture but differ in the number of CLBs. For example, the XC4025 device has 1024 CLBs while the XC4013 device has 576 CLBs [9]. Also these devices are available in various speed grades. A 4025 -3 speed grade device is faster than a 4025 -5 device. The target architecture for the components is a Xilinx 4000e -3 series device. The performance data for a design on a Xilinx 4000 series device is measured in terms of the number of CLBs, 3-LUTS, 4-LUTS, flip flops used, and the delay through the design. Library characterization aims at obtaining the above mentioned data for each of the components. The delay of the library component refers to the maximum delay at the outputs of the macro. The following section describes the characterization procedure.
Figure 4.2: XC4000e Series CLB

Figure 4.3: XC4000e Series Interconnect
4.3 Library Characterization Procedure

*Macros* are pre-synthesized, register-transfer level components that are characterized for various metrics. An outline of the traditional synthesis process for FPGAs is presented before the description of the macro characterization procedure. Figure 4.4 shows the synthesis steps involved in converting an RTL description to an FPGA bitstream. The RTL components undergo logic synthesis followed by *architecture specific* layout synthesis. During *logic synthesis*, the RTL description is converted to a gate level description that is optimized and mapped onto the target device using device specific libraries. This optimized netlist is then input to the layout synthesis tool. The layout tool performs the physical synthesis steps of partitioning, placement, and routing on the mapped components. This process is dependent on the architecture of the target device. At each synthesis step, more information is added to the description and the design undergoes several optimizations. It is only after these synthesis steps that accurate data for resource utilization and component delays can be obtained. The macro characterization procedure obtains this data for all the components in the library.

The macro components are described structurally, in a hardware description language like VHDL. These descriptions are usually *parameterized* and are independent of specific bitwidths. The area and delay information depends on the bitwidths of the components. Hence the components are synthesized for specific bitwidths and the trend for area and
In the case of the library manager, the macros were synthesized for 2, 4, 8, 16, 32, and 64 bits. The macros of a given size were synthesized for various shapes. The shape information is also stored in the library. The library manager may also interact with a placement tool and estimate the performance for shape specific queries. A detailed description of the macro characterization is provided in Sundaraman’s thesis [24]. The exact number of CLBs, 3-LUTs, 4-LUTs, Flip Flops (ffs), and delay for each macro of the above bitwidths, for each shape was obtained after layout synthesis. The characterized data for an adder macro for different bitwidths is shown table 4.1. This data was obtained after logic and layout synthesis of the adder for each of the specified bitwidths. The Synopsys design compiler (version 1998.02) was used for logic synthesis and the Xilinx M1 tools (version M1.3.7), for layout synthesis.

Based on these trends, estimation rules are developed to obtain the resource and delay characteristics for bitwidths of macros that are not actually synthesized. The library manager is essentially a ”macro delay and resource estimator” that uses the characterized macro information and rules to estimate the resource and delay information for macros of any bitwidth. The following section describes the PDL+ model.

### 4.4 Modules and Attributes

The synthesis system queries the model for the performance information of a single component at a time. The PDL+ model thus contains one module *component* whose attributes and evaluation rules are shown in figure 4.5 A module definition has two parts, an attribute declaration part and an evaluation rule part. The attribute declaration part is preceded by the keyword *attributes* and the evaluation rules are specified after the keyword *rules*. The *num_inputs*, *comp_name*, *num_bits*, and *library* attributes are supplied to the model. Hence these are declared *primitive*. The first three are built-in data types, while the library attribute is of a user defined type, called *ComponentLibrary*. The library attribute contains the characterized data for all the macros in the library. It is also declared primitive so that the same model may be used for different sets of library data. The remaining five attributes are evaluated according to the rules specified in the model.

<table>
<thead>
<tr>
<th>#Bits</th>
<th>CLBs</th>
<th>4-LUTs</th>
<th>3-LUTs</th>
<th>FFs</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>7.799</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>12</td>
<td>3</td>
<td>0</td>
<td>22.726</td>
</tr>
<tr>
<td>16</td>
<td>22</td>
<td>24</td>
<td>7</td>
<td>0</td>
<td>65.536</td>
</tr>
<tr>
<td>32</td>
<td>46</td>
<td>48</td>
<td>15</td>
<td>0</td>
<td>114.03</td>
</tr>
<tr>
<td>64</td>
<td>94</td>
<td>96</td>
<td>31</td>
<td>0</td>
<td>220.68</td>
</tr>
</tbody>
</table>

Table 4.1: Characterized Data for the Adder Macro

delay is used to obtain the estimation functions.
module component

attributes
    primitive num_inputs : int;
    primitive comp_name : string;
    primitive num_bits : int;
    primitive library : ComponentLibrary;
    num_clbs : int;
    num_luts3 : int;
    num_luts4 : int;
    num_flops : int;
    delay : real;

rules
    let selector : int = 1;
    num_clbs = get_num_clbs(comp_name,num_bits,num_inputs,library,selector);
    num_luts3 = get_num_luts3(comp_name,num_bits,num_inputs,library,selector);
    num_luts4 = get_num_luts4(comp_name,num_bits,num_inputs,library,selector);
    num_flops = get_num_flops(comp_name,num_bits,num_inputs,library,selector);
    delay = get_delay(comp_name,num_bits,num_inputs,library,selector);
end module;

Figure 4.5: The Component Module

4.4.1 ComponentLibrary type

Table 4.1 again shows the typical data for an adder macro. For each macro of a given size, the number of CLBs, 4-LUTs, 3-LUTs, Flip flops and delay information is stored. All of the information shown in the table, is stored for each shape of the macro and for all macros in the library.

Records and lists in PDL+ are very useful to create user defined data types. Any user defined type must be defined between the keywords type and end type. A record is used to group a number of different data types together. The ComponentLibrary type is built from a number of user defined types. Each of these types are described below. Every

ShapeClbRec :record -- stores shape no., no. of clbs, delay without pad
    shape_num : int;
    num_clbs : int;
    num_flops : int;
    num_luts4 : int;
    num_luts3 : int;
    delay : real;
end record;

Figure 4.6: ShapeClbRec Record

shape of a component of a given size has the following performance data associated with it – shape number, number of clbs, number of flops, number of 3-LUTs, number of 4-LUTs, and the delay. Since the fields are of different
types (ints and real) these are stored as records. ShapeClbRec is the type associated with the record. The PDL+ declaration is shown in figure 4.6.

Each bitwidth of the macro consists of a list of the above information corresponding to each of its shapes. This is stored in the record BitsShapeClbRec as shown in figure 4.7. A list of these BitsShapeClbRecs stores the data for all bitwidths of a given macro. This list is of the type ClbData. Each component in the library consists of a name and the component data. This record is of the type LibraryComponent. A list of these constitutes the entire macro library, defined to be of type, ComponentLibrary. Figure 4.7 shows the type definitions for the record structures as defined in PDL+. The component module, shown in figure 4.5 has a primitive attribute called library, of the ComponentLibrary type. The figure 4.10 shows sample data stored in the record structure described in figure 4.7 for BitsShapeClbRec:

```plaintext
record
    num_bits : int;
    data[] :ShapeClbRec;
end record;
```

ClbData[] : BitsShapeClbRec;

LibraryComponent : record
    component_name : string;
    component_data :ClbData;
end record;

ComponentLibrary[] : LibraryComponent;

Figure 4.7: LibraryComponent Datatype

a library of two components, c_sub and c_adder. For each bitwidth is a list of performance characteristics for four different shapes. The data for each macro is stored in increasing order of bitwidths.

4.5 Estimation Function

The trend of the metrics over bitwidths for all components in the library was mostly linear. A piece wise linear approximation was used to estimate the performance metrics for bitwidths of macros that did not belong to the database. Using this method, area and delay estimates are obtained for any bitwidth bounded by the lowest and highest bitwidths for the macro in the library. The PDL+ implementation of the function is shown in figure 4.8. We access the corresponding component metric by performing a sequential search on the elements of the list of library components.

Every function definition in PDL+ must follow the key word function and the function body is enclosed within begin and end blocks. The functions are side-effect free and do not alter attributes within the function body. Hence, attributes are assigned the return value of functions. A default return value may be specified at the end of the function.
definition (in figure 4.8 :int:=-1 is the default return value of the get_num_clbs function). The default value is optional if the function has an explicit return statement.

If the component name exists in the library, the required bitwidth is compared with the corresponding list of bitwidths and interpolated linearly to obtain the number of clbs. Since each bitwidth has a number of shapes associated with it, the selector argument is used to select the desired shape. Here, it has been set to 1 as shown in figure 4.5. Hence the data associated with the first shape is used. The fields of a record are accessed using the "." operator. The built-in function \#, is used to return the size of lists and arrays. Iteration through homogeneous lists is done using the foreach statement as shown in figure 4.8. If the component name is not found in the library, or the bitwidths required are out of range of those in the library, the function returns a value of -1. Similar functions are used to obtain the delay, 3-LUT, 4-LUT, and flip flop counts of a macro.

The following section outlines the procedure to create an executable model of the library manager and evaluate it from the shell and through the procedural interface.

### 4.6 Executing the model

In order to create an executable model, a specific instance of the module is instantiated in a design file. The design file contains an instance of the module component and is shown in figure 4.9. This design file is specified in the flattened PDL netlist format or pnf. The synthesis tool sets the attributes of the comp1 instance of the component module to obtain the desired data.

The primitive attributes may be set from within the design file, or supplied before evaluation. Since the library attribute is a constant for a given architecture, it is a good idea to set this attribute from within the design file. Each library of components may have a particular design file associated with it. The num_inputs, bit_width, and component_name attributes are then changed for obtaining the relevant information. Figure 4.10 shows a sample design file with the library attribute. The attribute and module names must conform with those specified in the pdl model file. The procedure to create an executable model and query the evaluated model from within the ARC shell is outlined below.

- load program libman.pdl
- load design lib1.pnf
- set value comp1 comp_name "c_adder"
- set value comp1 num_bits 4
- set value comp1 num_inputs 2
- evaluate
- show value comp1 *
function get_num_clbs(component : string; bits : int; num_inputs : int;
   library : ComponentLibrary; selector : int): int := -1
begin
   clbdata : ClbData;
   clbs : int := -1;
   index : int := 1;
   while (index < #(library) and library[index].component_name!=component)
begin
      index := index + 1;
end;
if (library[index].component_name == component) then
begin
   clbdata := library[index].component_data;
   if (clbs == -1) then
begin
      foreach p : int in 1 to #(clbdata)-1
      {
         if ((clbdata[p].num_bits <= bits) and (clbdata[p+1].num_bits > bits)) then
begin
            prev : int;
            next : int;
            prev := clbdata[p].data[selector].num_clbs;
            next := clbdata[p+1].data[selector].num_clbs;
            clbs := prev + (next - prev)*
            (bits - clbdata[p].num_bits)/(clbdata[p+1].num_bits - clbdata[p].num_bits);
        end;
        endif;
      }
      if (clbdata[#clbdata].num_bits == bits) then clbs := clbdata[#clbdata].data[selector].num_clbs;
    endif;
end;
if (num_inputs > 2 and clbs !=-1) then clbs := clbs*num_inputs/2;
endif;
return clbs;
end;

Figure 4.8: The get_num_clbs Function
module component = comp1
end module;

Figure 4.9: Sample Design File

module component = comp1
attributes
  library =
  [<"c_sub",
   [<2,[<1, 2, 0, 3, 0, 2.0>],
    <2, 2, 0, 3, 0, 2.0>],
    <3, 2, 0, 3, 0, 2.0>],
    <4, 2, 0, 3, 0, 2.0>],
   <4,[<1, 4, 0, 6, 1, 9.181>],
    <2, 4, 0, 7, 1, 10.612>],
    <3, 4, 0, 7, 1, 11.182>],
    <4, 4, 0, 7, 1, 10.921>];
end module;

Figure 4.10: Sample Design File with library Data - lib1.pnf

The model file is loaded first, followed by the corresponding design file. The set value command is used to assign values to the primitive attributes. Once the primitive values are set, the compiled model is ready to be evaluated. After evaluation, all the attributes of comp1 are assigned using the rules specified in the model and the input primitive values. The show value command displays the appropriate attribute values. The * option shows all the attributes corresponding to the comp1 instance of the component module. The above commands may be entered at the shell prompt interactively, or stored in a file and executed in a batch mode. In the latter case, the outputs are redirected to a file. New queries are made to the model by again using the set value command to set the comp1 attributes, corresponding to the new query.

4.6.1 Through the API

The executable model may also be evaluated and queried from within a C++ program using the API functions. The synthesis tool may interact directly with the ARC system through the API, or through an interface that may be
linked with the synthesis engine like any other C++ library. In the second case, the synthesis tool is unaware of the internals of the library manager and is provided with the necessary interface functions to access the required data.

Figure 4.11 is the header file for the interface program. The LibraryManager class provides the necessary functions that may be called from within the synthesis engine. These are declared in the public part of the class. The LibraryManager class contains a pointer to the ArcSystem class. Using this, the various API functions are accessed.

```cpp
class LibraryManager {

public:
    LibraryManager();
    LibraryManager(char* libraryname, char* executablefilename);

    //compdetails[0] = bitwidth, numInputs is always taken as 2
    //nooffields gives the no.of entries in compdetails
    int getClbCount(char* compname, const int* compdetails, int nooffields);
    int get3LutCount(char* compname, const int* compdetails, int nooffields);
    int get4LutCount(char* compname, const int* compdetails, int nooffields);
    int getFlopCount(char* compname, const int* compdetails, int nooffields);
    float getDelay(char* compname, const int* compdetails, int nooffields);
    char* getLibraryName();

    ~LibraryManager();

private:
    ostrstream _libraryname;
    ostrstream _datafile;
    char* _compdata;
    int _attributetype;

    class ArcSystem* _arc;
    int _errorcode;

    void printErrors();
    void readAttributeData();
    void setAttributeValues(char* componentname, int bits, int inputs);
};
```

Figure 4.11: The Interface Header File

The PDL+ program along with a specific design file constitutes the executable model and is saved from within the shell. This executable model is loaded using the function `_arc->LoadExecutableModel("filename")`. All arguments are passed as strings to the ARC system. Objects of the ostrstream class provide convenient string handling features.
The primitive attributes are set using the function \texttt{arc}→\texttt{SetAttributeValue(“object”, ”attribute”, value)}. The value is also passed on as a string. All the commands that are executed from the shell may also be accessed through the API.

4.7 Results

We have included the characterized data for all components that were synthesized using the synopsys design compiler, and the Xilinx suite of M1 tools in the Appendix A.

4.8 Summary

The first case study was presented in this chapter. The usage context and development methodology were described. The Xilinx 4000 series device architecture was also described in the chapter. A library of Xilinx devices are the target architecture for the synthesis tools described in the following chapters. The library manager is used to obtain component area and delay estimates for the components and is integrated into two other models that are described in the following two chapters. Also the steps involved in interfacing an executable model with a C++ program were illustrated.

This chapter introduces the reader to the \textit{module} design component, illustrates the use of attributes and their evaluation rules. The library manager models the characterization data using the PDL+ record type. The following chapter introduces a register transfer level model that makes use of the estimation functions of the library manager.
Chapter 5

RTL Estimator

The library manager described in the previous chapter models a component library that is usually input to hardware synthesis systems. The register transfer level estimator predicts the area and clock period of designs that contain instantiations of components from a component library. An RTL netlist is input to the estimator that uses the estimation functions of the library manager to predict the overall area and clock speed of the post-layout design. The RTL netlist may be generated as an output of a high level synthesis tool.

The estimator is a hierarchical model and the evaluation rules require bottom-up propagation of the leaf level attributes. The evaluation rules make use of a variety of expressions and constructs that were described in chapter 2. We will describe these constructs in greater detail with the help of examples. We describe the PDL+ model of the estimator in section 5.2 and present the results obtained from the model in section 5.3.

5.1 Estimation Methodology

The target architecture for the designs is a Xilinx 4000e series FPGA device. We therefore use the functions and the characterized data of the components described in the previous chapter. The two metrics of interest are the area and the clock speed of the design. The area of the design is the sum of the areas occupied by each of the components. Using the estimator functions of the library manager, we obtain the clbs, luts_3, luts_4, ffs, and the delay for each of the components. The area of the design in clbs is given by

\[
clbs = \max(\#\text{luts}_4/2, \#\text{luts}_3)
\]

where, luts_4 and luts_3 refer to the total number of luts occupied by the design. This is because the RTL netlist is subject to technology mapping followed by layout synthesis. During the technology mapping phase, the functions are mapped to the CLBs. Since each CLB has two 4-input LUTs and one 3-input LUT, we estimate the optimization due to the mapping process as shown above.
We estimate three different clock periods for the design. These are the estimated clock, heuristic high, heuristic low clock periods. A typical RTL data path netlist that is generated by a high level synthesis tool is shown in figure 5.1.

![Typical RTL Netlist](image)

The estimated clock period is twice the sum of the worst case component delay, register delay, multiplexer delay, and a nominal interconnect delay. The interconnect delay is estimated to be the delay through 4 double wires and the programmable interconnect points.

On careful analysis of two designs we found that the worst case interconnect delay was between 45-150 ns. These values were used to fix the lower and upper bounds of the clock period. This is used to calculate the heuristic low and heuristic high clock period.

Before we describe the evaluation rules of the PDL+ components, we highlight some basic features of PDL+ that are used through the rest of this chapter.

**Some Features of PDL+**

We highlight three important design tips from the language point of view in this subsection.

The keyword *let* allows us to declare new attributes that are used locally within the design component. These are typically used to store temporary attributes or to store partially evaluated attribute results.

The *foreach* expression is useful for iterating through sets of objects, lists and arrays, or over a discrete range of values. It is a list constructor. A foreach expression has three parts: an iteration variable, the domain of iteration, and a body. A foreach expression returns a list of values. The type of the return value is a list of $Ts$ where $T$ is the type of each element in the list. Each element in the list is the result of evaluating the body of the foreach expression once for each value of the iterating variable in the iteration domain [13]. This list of type $T$, may be passed as an argument to functions. Figure 5.2 shows the foreach expression over a discrete range of values. The attribute small_list is assigned the integers 1 through 10 after evaluation.

It is a good idea to define some commonly used functions in a file and include the file whenever necessary. We define some basic functions like max_int, sum_int, sum_real, product_int, in the file functions.pdl. Each of the functions takes a list of integers, or reals and returns their maximum, sum, or product. The definition of the max_int function is shown in figure 5.3. Since the result of a foreach expression is a list of values, we can define a foreach expression as an argument of functions like max_int. The evaluation rules described in the following sections illustrate this.
module design
attributes
small_list[i]:int;
rules
small_list =
   foreach i: int in 1 to 10
   {i};
end module;

Figure 5.2: Foreach Expression Over Discrete Range

function max_int(ints[]): int: int := 0
begin
   temp: int := 0;
   if #(ints) == 0 then return 0; endif;
   temp := ints[1];
   foreach i:int in ints
      { if temp < i then temp := i; endif; }
   return temp;
end;

Figure 5.3: Max Function

We now describe the PDL+ implementation of the model.

5.2 PDL+ Implementation

The PDL model consists of two modules, the component module and the design module. We describe these modules, their attributes and evaluation rules in the following subsections.

5.2.1 Module : Component

Each component in the design is represented as a component module instance with the primitive attributes set accordingly. The clock_phase attribute is used to distinguish between sequential and combinational components. The delay estimation procedures differ on the basis of the clock_phase attribute. The component module, its attributes and evaluation rules are shown in figure 5.4. Unlike in the library manager, where the user specifies the library data, the characterized library data is internal to the RTL estimator. Consider the following library of components.

type
components : enum{adder,multiplier,subtractor,mux,divider,comparator,andgate,orgate,reg};
end type;
port ioport
attributes
delay : real;
end port;

module component
ports
inputs[] : ioport;
outputs[] : ioport;

attributes
primitive num_inputs : int;
primitive comp_name : components;
primitive num_bits : int;
primitive clock_phase : int; -- 0 for combinational, 1 for sequential
clbs : int;
flops : int;
luts3 : int;
luts4 : int;
delay : real;
seq_delay : real;

rules
clbs = get_clbs(comp_name,num_bits,num_inputs); -- library manager functions
flops = get_flops(comp_name,num_bits,num_inputs);
delay = get_delay(comp_name,num_bits,num_inputs);
luts3 = get_luts3(comp_name,num_bits,num_inputs);
luts4 = get_luts4(comp_name,num_bits,num_inputs);

outputs[] 'delay = if clock_phase == 0 then max(foreach b :ioport in inputs[b'delay'])+delay else 0.0 endif;

seq_delay = if clock_phase == 0 then 0.0 else max(foreach b :ioport in inputs[b'delay']) + delay endif;
end module;

Figure 5.4: Component Module

Figure 5.4 shows the PDL+ implementation of the component module.

The delay attribute of the ports is assigned to be the combinational delay of the design. In the case of combinational components the output delay is set to the sum of the maximum of the input port delays and the component delay. In the case of sequential components, feedback paths may exist from the output to the inputs. This would result in a cyclic dependency in the evaluation rule. Therefore, we set the output port delays to zero for sequential components and propagate the sum of the maximum input port delays and the flop delay through the seq_delay attribute. The seq_delay attribute is set to zero for the combinational components.
module design
ports
  inputs[ ] :ioport;
  outputs[ ] :ioport;
modules
  components[ ] :component;
attributes
  heuristilocw : real;
  heuristichigh : real;
  delay : real;
  clbs : int;
  flops : int;
rules
  let luts_3 : int = sum_int(foreach c:component in components f'c'luts3);
  let luts_4 : int = sum_int(foreach c:component in components f'c'luts4);
  clbs = max(luts_4/2,luts_3);
  flops = sum(foreach c:component in components f'c'flops);
inputs[ ]'delay = 0.0;
delay = 2 * max(max_real(foreach o:ioport in outputs o'delay),
  max_real(foreach c:component in components c'seq_delay)) + 2*get_delay(mux,2) +
interconnect_delay();
heuristilocw = 2 * (max_real(foreach c : component in components max_real c'seq_delay) + 45);
heuristichigh = 2 * (max_real(foreach c : component in components max_real c'seq_delay) + 150);
end module;

Figure 5.5: Design Module

5.2.2 Module : Design

The entire design is represented as a collection of component modules and primary input and output ports. Figure 5.5
shows the design module its attributes and evaluation rules. The luts3, and luts4 attributes of each of the component
modules are propagated bottom-up to the design module.

The delays of the primary input ports are all set to zero using the broadcast rule shown in figure 5.5. The maximum
between the output port delays and the maximum seq_delay attributes of the components dictates the period of the
clock. The evaluation rules for the clock periods are also shown in figure 5.5.

5.3 Results

Each of the benchmark circuits were specified in behavioral VHDL and were converted to equivalent Register Trans-
fer Level netlists using the HLS tool Asserta [8]. The RTL netlists were input to the estimator and were also subject
to logic and layout synthesis using the Synopsys Design Compiler and the Xilinx M1 tools. The estimated areas and
clock speeds were compared with those generated after layout synthesis. A comparison of the estimated and actual 
values is shown in this section.

Figure 5.6 shows the actual vs estimated clb counts for each of the benchmark designs. Figure 5.7 shows the actual 
vs estimated flip flop counts for each of the benchmark designs. Figure 5.8 shows the actual vs estimated clock 
speed in MHz for each of the benchmark designs. The first bar indicates the estimated value obtained from the 
estimator. The second bar represents the value obtained from M1 tools after layout synthesis. The third bar indicates 
the heuristic high clock speed, and the fourth bar indicates the heuristic low clock speed. The estimated and the 
actual values are reasonably close and in most cases these values were found to lie between the heuristic high and 
low values.

5.4 Summary

We described a register transfer level estimator that predicts the post logic and layout synthesis area of RTL netlists. 
The estimator uses the functions of the library manager described in the previous chapter. We also presented the 
results obtained from the estimator.

The foreach expression, use of the let statement, and the use of the estimation functions of the previous chapter 
were illustrated through the different evaluation rules. The attributes of the child operation modules are propagated 
bottom-up to the parent design module and are used in the evaluation rules of the design module.

The next chapter describes a hierarchical model called the behavioral estimator that predicts the number of resources
in the data path of the RTL design and also makes use of the estimation functions of the library manager.
Figure 5.8: Clock Speed
Chapter 6

Behavioral Estimator

The RTL estimator from the previous chapter models the performance of designs that are usually output from a high level synthesis (HLS) system. This chapter deals with the development of a performance model of a high-level synthesis system. The model will be referred to as the Behavioral Estimator. The model estimates the area and clock speed of the resultant design, during the synthesis process. The HLS tool can use these estimates to iteratively improve the quality of the design.

The first few sections give a brief introduction to traditional high-level synthesis and the usage context of the behavioral estimator. This is followed by a description of the estimation technique and a step-by-step development of the model in PDL+. Section 6.5 compares the results from the estimator with the actual values and suggests improvements to the current model.

The model is a static system where the attributes do not depend on their previous values and a single evaluation of the compiled model is sufficient. In the future sections we illustrate the aspects of hierarchical composition, bottom-up attribute propagation, and the interaction between different evaluation rules. The following section gives a brief introduction to the high-level synthesis system, whose performance model is the behavioral estimator.

6.1 High-level Synthesis: Estimator’s View

The translation of a behavioral specification to a register transfer level (RTL) design is the essence of high-level synthesis. Traditional HLS process consists of the following four tasks: scheduling, register optimization, interconnect generation, and data path-controller generation. Following this, target technology dependent synthesis steps are performed. For example, if the device onto which the design is to be mapped is a Field Programmable Gate Array (FPGA), the HLS phase is followed by architecture specific logic and layout synthesis. Each of these synthesis steps is time consuming and therefore it is essential to develop estimators that can make reasonably accurate predictions about the post-synthesis performance of the design. The following sections describe the four main tasks performed
during HLS, as seen by the estimator. More detailed descriptions of these steps are found in [25].

### 6.1.1 Scheduling

*Scheduling* is the task of assigning behavioral operations across time steps. Figure 6.1 shows the operation graph representation of the function \( C' = \sum a[i].b[i] \). Each node in the graph represents an operation in the specification and an edge between any two nodes denotes a dependency. In figure 6.1, each operation has a type (ex. plus ‘+’), multiply ‘\(*\)’) and a bit-width or size \((m, n, k)\) associated with it.

Figure 6.2 is a scheduled operation graph where each operation is assigned to a specific time step (or control step). The dotted lines in the figure represent time-step boundaries. The number on the left denotes the cstep number. Similar operations across time-steps may be shared or assigned to the same physical operator. Some synthesis systems allow operators of different bit-widths to be shared (operator folding), while some systems only allow operators of the same bit-width to be shared (do not support operator folding). If the system does not allow operator folding, the schedule in figure 6.2 requires 2 m-bit multipliers, 1 n-bit adder and 1 k-bit adder.

Thus in a scheduled operation graph, each operator has a cstep number assigned by the scheduler, in addition to its type and size information. This is used by the estimator to determine the number of resources required for the RTL implementation.
6.1.2 Register Optimization

When a physical resource is shared across time steps, its inputs and outputs over different time-steps need to be stored. This sharing of the physical operators over time-steps is achieved by introducing registers and multiplexers at the inputs and outputs of the operator. The registers store the input and output values across time step boundaries and are suitably passed on to the operators using multiplexers. During register optimization, variables with non-overlapping life spans may be assigned to the same physical register. The estimator predicts the number of registers that will be required by the design.

6.1.3 Interconnect Generation

Whenever a resource is shared, a multiplexer is introduced at the inputs and outputs of the resource. The resource could either be a functional unit or a register. The control inputs of the multiplexer select the appropriate inputs to the resource, for each cstep. The estimator accounts for the additional area occupied by these multiplexers that facilitate sharing of resources.

6.1.4 Datapath-Controller Generation

Finally, the controller that enables the appropriate inputs to the multiplexers and controls the flow of data between the physical resources is generated by the HLS system. The estimator however does not account for the controller
area.

6.2 Usage context of the Behavioral Estimator

Scheduling is the most crucial step during HLS and it greatly influences the performance of the output design. It is only after scheduling that reasonably accurate resource estimates can be made. Thus the input to our estimator is a scheduled operation graph.

The behavioral estimator is aware of the kind of optimizations to be performed by the HLS engine and has predictive rules for performance-estimation based on this information.

Thus the estimator takes a scheduled operation graph as input and predicts the clock speed, and the number of resources utilized by the executable design.

Figure 6.3 shows the interaction between the synthesis system and the estimator.

6.3 Building Blocks : Operation Graph

The behavioral estimator is developed incrementally based on the discussions in the previous sections. The first step is to represent the input operation graph in PDL+. A graph is a collection of nodes and edges and is modeled hierarchically. The behavior module shown in figure 6.4 is the core of the estimator. It is the top-level module and contains a set of operation modules. Each node in the graph is modeled by the operation module, and the entire graph is represented by the behavior module.

Figure 6.5 shows the operation module. The connectivity or dependency information between nodes is established through the set of input and output iports. The iport declaration is also shown in figure 6.5.

The scheduled graph whose performance needs to be estimated is represented by instantiating the operation modules in a design file and associating these operations to an instance of the behavior module. The exact connectivity information is also mentioned in the design file, and is established using ports of the operation module instances. Figure 6.7 shows the design file that corresponds to the graph shown in figure 6.6. Each operator shown in figure 6.6 has an equivalent operation module instance (op1, op2,..), in the design file. All these operation module instances are associated with their parent behavior module (beh). The ports are primitive objects and are visible outside the module definitions. The keyword net is used to specify the interconnections as shown in figure 6.7.

For now, no attributes have been attached to either the modules or the ports.
Figure 6.3: Interaction between HLS System and Behavioral Estimator
module behavior
  modules
    operations[ ]: operation;
  ports
    inputs[ ]: ioprt;
    outputs[ ]: ioprt;
end module;

Figure 6.4: PDL+ Representation of the Operation Graph

module operation
  ports
    inputs[ ]: ioprt;
    outputs[ ]: ioprt;
end port
end module;

Figure 6.5: PDL+ Representation of a node and ioprt Declaration

6.4 Estimation Algorithms

The behavioral estimator is developed for a HLS system called Asserta [8] whose synthesis flow is very similar to
the conventional HLS process described in the earlier section. The output of Asserta is an RTL description in the
form of a datapath-controller pair. The estimator predicts the resource and timing metrics of the datapath of the
design.

The RTL datapath comprises of the following resources: functional units, registers, and multiplexers. The total area
of the design is the sum of the areas occupied by resources of each type. The datapath components are instantiated
from the RTL component library. The area and clock speed of the resultant RTL design are the metrics to be evaluated
by the model. The algorithms implemented for obtaining the resource and timing estimates will be described in detail
in the following subsections. Also, the implementation of the algorithms in PDL+ will be shown.

Let the library contain the following basic components – adder (+), subtractor (-), divider (/), multiplier (*), multi-
plexer, and register. The total area occupied by the RTL implementation of the operation graph is the sum of the
areas occupied by each of these components. The actual area occupied depends on the target architecture for the
implementation. The target architecture for the designs is a Xilinx 4000 series FPGA.

6.4.1 Evaluation Rules : Operation Module

The estimator uses the functions described in the previous chapter to obtain the area and delay estimates for the
required operators. Hence given the bitwidth, and component type information, the area (CLBs, 3-LUTS, 4-LUTs,
and ffs) and delay information is obtained for the operators. Figure 6.8 shows the attributes and evaluation rules for
the operation module. Since the input is a scheduled operation graph, each operator in the graph has a \textit{cstep} number that is assigned by the scheduler. The operators are defined by their type and bitwidth. The primitive attributes \textit{opcode}, \textit{size}, and \textit{cstep} represent this information.

The \textit{get\_clbs}, \textit{get\_ffs}, and \textit{get\_delay} functions call the area and delay estimation functions described in the library manager. The \textit{clbs} attribute is evaluated as explained in the previous chapter. The attributes, \textit{in\_registers} and \textit{out\_registers} are used to estimate the register area of the design.

All of the estimation algorithms are implemented as attribute evaluation rules of the parent behavior module. The order in which these rules appear in the program is immaterial, as the compiler performs a dependency analysis on the attribute evaluation rules and determines the evaluation order. Some of the important attribute evaluation rules of the behavior module are presented in the following subsections.

### 6.4.2 Evaluation Rules: Behavior Module

Figure 6.9 shows the behavior module declaration, its attributes and evaluation rules. The attributes \textit{op\_clbs}, \textit{reg\_clbs}, \textit{mux\_clbs}, and \textit{reg\_ffs} that are used in these evaluation rules are local to the behavior module. Evaluation rules corresponding to these attributes are described in the future sections. The last five attributes are described in section 6.4.6.
module behavior = beh
    ports
        inputs = {i1,i2,i3,i4,i5,i6,i7,i8} ;
        outputs = {o1} ;
    modules
        operations = {op1,op2,op3,op4,op5,op6,op7} ;
end module ;

module operation = op1
    ports
        inputs = {i1,i2} ;
        outputs = {o1} ;
end module ;
module operation = op2
    ports
        inputs = {i1,i2} ;
        outputs = {o1} ;
end module ;
module operation = op3
    ports
        inputs = {i1,i2} ;
        outputs = {o1} ;
end module ;
module operation = op4
    ports
        inputs = {i1,i2} ;
        outputs = {o1} ;
end module ;
module operation = op5
    ports
        inputs = {i1,i2} ;
        outputs = {o1} ;
end module ;
module operation = op6
    ports
        inputs = {i1,i2} ;
        outputs = {o1} ;
end module ;
module operation = op7
    ports
        inputs = {i1,i2} ;
        outputs = {o1} ;
end module ;

net beh1.i1, op1.i1; net beh1.i2, op1.i2;
net beh1.i3, op2.i1; net beh1.i4, op2.i2;
net beh1.i5, op4.i1; net beh1.i6, op4.i2;
net beh1.i7, op5.i1; net beh1.i8, op5.i2;
net op1.o1, op3.i1; net op2.o1, op3.i2;
net op3.o1, op7.i1; net op4.o1, op6.i1;
net op5.o1, op6.i2; net op6.o1, op7.i2;
net op7.i2, beh1.o1;

Figure 6.7: Design File using Operation and Behavior Modules
module operation
ports
  inputs[]: ioport;
  outputs[]: ioport;
attributes
  primitive opcode: operation_type;
  primitive size: int;
  primitive cstep: int;
  clbs: int;
  ffs: int;
  delay: real;
  in_registers: int;
  out_registers: int;

rules
  clbs = get_clbs(opcode,size);  -- call functions of library manager
  ffs = get_ffs(opcode,size);  -- call functions of library manager
  delay = get_delay(opcode,size);  -- call functions of library manager
  in_registers = #inputs;
  out_registers = #outputs;
end module;

Figure 6.8: Attributes and Evaluation Rules of the Operation Module

6.4.3 Attributes: op_clbs

The algorithm described in this section is used to estimate the functional unit area of the design. Suppose the library contains the following four functional operators, plus, minus, mult, and divide. Figure 6.10 defines the allowed types for the operations. The functional unit area is the sum of the areas for each of these operators.

The evaluation rule for the op_clbs attribute is

\[
\text{let } \text{op}_1\text{clbs} = \text{plus}_1\text{clbs} + \text{minus}_1\text{clbs} + \text{mult}_1\text{clbs} + \text{div}_1\text{clbs};
\]

The main task for functional unit area estimation is to predict the operations that may be assigned to the same physical operator. The algorithm described here is applicable to HLS systems like Asserta, that do not allow operator folding. That is, same operators of different bitwidths cannot be shared.

The estimated number of operators of a given type and size required in a design, is the maximum number of these operators needed in any control step.

For example, in figure 6.6, the maximum number of “m bit” multipliers required in any cstep is 2. Therefore op1 and op2 can be assigned to the physical multipliers m1 and m2 in cstep1 and to op3 and op4 at the next cstep.

The following subsections describe the evaluation rules for the plus_clbs attribute. Similar rules are applied to obtain
module behavior
modules
operations{}: operation;
ports
inputs{}: ioprt;
outputs{}: ioprt;
attributes
clbs: int;
ffs: int;
csteps: int; -- number of control steps used in the partition
clock_period: real; -- estimated clock period
clock_frequency: real; -- estimated clock frequency.
heuristic_low_clock_period: real;
heuristic_high_clock_period: real;
rules
clbs = op_clbs + reg_clbs + mux_clbs;
ffs = reg_ffs;
end module;

Figure 6.9: Behavior Module and its Attributes

type
operation_type: enum {plus, minus, multiply, divide};
end type;

Figure 6.10: User Defined Type: op_type

minus_clbs, mult_clbs, and div_clbs.

Attribute: plus_clbs

Figure 6.11 shows the rule that estimates the number of plus_clbs. The inner most foreach expression creates a list of the number of clbs occupied by an adder in a given cstep and of a given size. If an adder of a given size occurs in the given cstep, we append the corresponding number of clbs, otherwise a zero is appended to the list. The else part of the expression is not optional because the result of the foreach expression is a list that should contain an entry for each of its iterations. This list is passed as an argument to the sum_int function.

The inner sum_int function finds the sum of the number of clbs of plus operators of a given size in a given cstep. The outer foreach statement that iterates over all csteps and creates a list of number of adder clbs required for each cstep.

The maximum number of clbs for the plus operator of a given size, over all csteps is found using the max_int function.

Finally, the sum of the clbs over all sizes of the plus operator gives the total plus_clbs.

The plus_sizes attribute that is used in the outer most foreach expression is evaluated by the rule shown in figure
6.12.

```
let plus_clbs: int =
  sum_int(foreach i:int in plus_sizes
    { max_int(foreach c:int in 1 to csteps
        { sum_int(foreach op:operation in operations
            { if op'opcode == plus and
              op'cstep == c and
              op'size == i
                then op'clbs else 0 endif ) }) )
  });
```

Figure 6.11: Number of Plus CLBs

Attribute: plus_sizes

```
let plus_sizes: int[] = begin
  a[]: int := {};
  foreach op:operation in operations
    { if op'opcode == plus then a := a union { op'size }; endif }
  return a;
end;
```

Figure 6.12: Set of Plus Sizes

The evaluation rules are specified within begin and end blocks. A series of statements are enclosed within these blocks. These are side-effect free and can only assign values to attributes using the return statement. Hence the set of integers \(a\), is used in the body and returned at the end of the block. Each operation in the behavior module is traversed using the **foreach** statement. A set of integers is used instead of a list because the built-in **union** function on sets, ensures that the plus_sizes attribute only contains unique entries for each of the sizes.

6.4.4 Attribute: reg_clbs, ffs

To estimate the area occupied by registers, the number of registers and their respective bitwidths are determined. The register optimization phase that results in the sharing of registers across control steps is accounted for by the following rule.

*The total number of registers is the maximum number of them required at any control step.*

The number of registers in each cstep is obtained by the following two rules:

- **regs in cstep:** number of registers in each control step is the sum of the registers required by all operations in
that control step.

- *regs per operator*: registers required by each operator is the maximum number of its inputs or outputs.

In the Asserta synthesis system, the inputs and outputs of all operators are registered and this is accounted for by steps *Regs in cstep*, and *Regs per operator*.

The maximum of *Regs in cstep* over all *csteps* is the estimated number of registers required by the design. The register area evaluation rule in PDL+ is shown in figure 6.13.

\[
\text{registers} = \max_{\text{int}}(\text{foreach} \ c : \text{int in 1 to csteps}
\sum_{\text{int}}(\text{foreach} \ op: \text{operation in operations}
\text{if} \ op\text{'}cstep == c
\text{then} \max_{\text{int}}([op\text{'out registers, op\text{'in registers]})
\text{else} \ 0
\text{endif}))));
\]

Figure 6.13: Register Evaluation Rule

The average size of the register, *ave_reg_size* is the ratio of the sum of operator sizes to the total number of operators in the design. The area occupied by the registers is the *registers* times the area occupied by a register of size *ave_reg_size*. The *get_reg_clbs* function uses the estimation functions of the previous chapter to obtain the clb count for a register of size, *ave_reg_size*.

\[
\text{let} \ \text{ave_reg_size: int} = \\
\sum_{\text{int}}(\text{foreach} \ op: \text{operation in operations} \{ op\text{'size} \} )/#\text{operations};
\]

\[
\text{let} \ \text{reg_clbs: int} = \text{registers} \times \text{get_reg_clbs(ave_reg_size)};
\]

Figure 6.14: Average Size and Register Area

Since the only sequential element in the library is the register, the flip flops occupied by the design is the same as those occupied by the registers. The *get_reg_ffs* function accesses the library manager estimator functions and obtains the corresponding flip flop count.

\[
\text{ffs} = \text{registers} \times \text{get_reg_ffs(ave_reg_size)};
\]

6.4.5 Attribute: *mux_clbs*

The multiplexer area depends on the number of multiplexers and the number of inputs to each multiplexer. The estimation method for an operator of type *optype* and size *opsize* is shown below. The sum of the multiplexer areas for all operators over all sizes gives the total area estimate due to multiplexers.
• **optype over csteps:** count the number of instances of operators of type optype and size opsize in all csteps.

• **max M in any cstep:** find maximum number of instances (M) of these operators in any one cstep. (M is the number of muxes required)

• **inputs to mux:** optype over csteps / M gives the number of inputs of the mux.

• **clbs for a single mux:** using estimation functions obtain clb counts for mux of size opsize, and inputs to mux number of inputs.

• **total mux area for operator:** multiply clbs for single mux by 2 to account for the two ports and by M (number of muxes) to get the total mux area for the operator.

This however, is an overly pessimistic estimate and we can assume that in half the cases, proper ALU allocation will eliminate these muxes. So, we divide the overall multiplexer clbs by 2. This still is an overestimate due to which we have ignored muxes introduced due to register sharing.

\[
\text{mux.clbs} = \frac{(\text{mux.clbs for plus} + \text{mux.clbs for minus} + \text{mux.clbs for mult} + \text{mux.clbs for div})}{2};
\]

Figure 6.15 shows the mux.clbs for plus attribute. Similar rules evaluate mux.clbs for minus, divide, and mult. The get_mux.clbs function refers to the estimation functions to obtain the clb counts described in the previous chapter.

### 6.4.6 Attribute: clock.period

Figure 6.16 shows a typical RTL design that is output from a high level synthesis tool. Every operator in the RTL implementation has a register-multiplexer pair at its input and outputs. The clock period is therefore twice the worst case register-register delay.

The sum of the worst case macro delay, register delay, and a nominal interconnect delay (multiplexer and wiring delay) gives us the worst case register-register delay. All the above delays except the interconnect delay is obtained from the characterized macro library data. The interconnect delay is estimated to be twice the multiplexer delay plus the delay through 4 double wires and the programmable interconnect points.

On careful analysis of two designs we found that the worst case interconnect delay was between 45-150 ns. These values were used to fix the upper and lower bounds of the clock frequency. This is used to calculate the heuristic.high and heuristic.low clock frequency.

\[
\text{clock.period} = (\maxreal(\text{foreach op:operation in operations} \{\text{op.delay}\}))
+ \text{interconnect.delay()} + \text{register_delay()})*2;
\]
let mux_clbs_for_plus: int =
  sum_int(foreach i:int in plus_sizes
    { get_mux_clbs(
      sum_int(foreach c:int in 1 to csteps
        { sum_int(foreach op:operation in operations
          { if op'opcode == plus and
            op'cstep == c and
            op'size == i
            then 1 else 0 endif})
        max_int(foreach c:int in 1 to csteps
          { sum_int(foreach op:operation in operations
            { if op'opcode == plus and
              op'cstep == c and
              op'size == i
              then 1 else 0 endif})
        )
      )
    )
  )
  )

Figure 6.15: Muxes for Plus Operator

6.5 Results

Figure 6.16: Typical Datapath Output of an HLS Tool

Each of the designs was modeled in behavioral VHDL and synthesized using Asserta. This was followed by logic synthesis (using the synopsys design compiler) and layout synthesis (using Xilinx M1 tools) [26] of the datapath of the design. The area and timing estimates were compared with those obtained after layout synthesis.

Figure 6.17 shows the actual vs estimated clb counts for each of the benchmark designs.

Figure 6.18 shows the actual vs estimated flip flop counts for each of the benchmark designs. Figure 6.19 shows the actual vs estimated clock speed in MHz for each of the benchmark designs. The first bar indicates the actual value obtained from the M1 tools after layout synthesis. The second bar indicates the heuristic high clock speed, the third bar indicates the heuristic low clock speed, and the last bar indicates the clock speed obtained using the interconnect delays. The estimated clock speed always lies between the heuristic high and heuristic low clock speeds.
6.6 Summary

A hierarchical performance model was described in this chapter. An introduction to traditional high level synthesis and the usage context of the estimator was also shown. The estimator predicts the area and timing metrics of the RTL implementation during high level synthesis. Each of the estimation methods were described and the PDL+ implementation was shown.

The following chapter models a spatial partitioner that targets a family of reconfigurable computer architectures. The model also makes use of parts of the behavioral estimator and the library manager.
Figure 6.18: **FF Counts**

![FF Counts Diagram]

Figure 6.19: **Clock Speed**

![Clock Speed Diagram]
Chapter 7

Partition Estimator

The behavioral estimator described in the previous chapter models the performance of a high level synthesis system. The target architecture for the designs in the previous model was a single FPGA. This chapter describes a partition estimator that is used in the context of multi-FPGA reconfigurable computing (RC) systems. The estimator models the performance of partitioning and synthesis tools for reconfigurable computers. RC systems typically consist of a host computer that is connected to a reconfigurable board. The performance of a design on a single-FPGA system depends mainly on the synthesis process. While in the case of multi-FPGA boards, design performance depends on both the board architecture and the synthesis process.

A typical reconfigurable board consists of a number of programmable elements (usually FPGAs) that are connected through an interconnection network. The operations in an application targeted to RC systems are mapped onto the programmable elements (PEs). The process of automatically mapping an application to the programmable elements is the main task of a spatial partitioner. The partitioner divides the application into segments to be mapped onto the PEs. These partitioned segments undergo traditional high level synthesis, followed by logic and layout synthesis. The board resources utilized by a given mapping, and the latency of the post-synthesis design are the metrics that are estimated by the model. The partitioner can use these estimates to iteratively improve the quality of the mapping.

Section 7.2 describes the process of mapping an application onto the target architecture and the usage context of the estimator. The remaining sections describe the model development methodology with the annotated PDL+ code. Section 7.4 shows the different components of the partition estimator model. Each of these components are then described along with their evaluation rules. The results from the estimator are presented in section 7.6.

The model illustrates the use of inherited evaluation rules and demonstrates the top-down propagation of attributes. Some attributes are also propagated laterally through ports and carriers. The following section outlines the partitioning framework.
7.1 Partitioning for RCs: Estimator’s View

The resources on a typical RC board are the FPGAs, memories, and interconnect for inter-FPGA and memory communication. Examples of some RC boards are the WildFire family of boards, BORG architecture, and the RACE board. These boards differ in the type and quantity of PEs, provide varying interconnect capability between PEs, between memories and PEs, and so on. A detailed survey of these architectures can be found in Radhakrishnan’s thesis [19].

Figure 7.1 shows a typical application, target RC architecture, and the task of an RC compiler. The target architecture consists of four PEs, four memories, and an interconnection network. Here adjacent PEs communicate through the direct interconnect, and the memories are local to the respective PEs. The figure also shows a sample application represented as a data flow graph (DFG). The arrows in the figure represent the binding information produced by an RC compiler. The main tasks of the compiler are:

- bind all operations in the DFG to PEs on the RC board
- allocate memories or on-board registers for data
- assign interconnect or memory resources for data communication between operators in different PEs
- allocate interconnect for PE synchronization

The board architecture imposes constraints to the partitioner on the possible binding of operations to PEs. Typical constraints are the area available on the PEs, interconnect available for the data transfers and memory resources for the data. For example, if in figure 7.1 the direct interconnect bandwidth between PE2 and PE3 is only 10 bits, the above mapping would not be valid. The mapping should be such that the board-level constraints are met. If any, or a
combination of the constraints are not satisfied, it results in a violation. Hence the performance of a mapping of the design on a target RC system depends on the architecture.

A number of RC architectures are currently available and new architectures are constantly being developed. Therefore it is important to develop retargetable RC compilers, that would ideally be independent of the target architecture. In order to achieve this retargetability, most of the architecture specific information is abstracted away from the compiler and incorporated in an external estimator. Different estimators corresponding to new architectures may be developed and interfaced with the partitioner. Also the synthesis process affects the performance of the resultant implementation. The synthesis related information is also abstracted away from the compiler so that it may be retargeted to different design flows.

The partition estimator computes the cost of a given binding using the architecture specific information and the possible synthesis optimizations. Using this cost the compiler tries to find a constraint satisfying solution that also minimizes the latency of the implementation. This process is a combinatorial search problem and a number of heuristics have been developed to solve it effectively.

7.2 Usage Context

The partition estimator interacts with a generic RC compiler that generates possible bindings for the application. Figure 7.2 shows the usage context and interaction between the compiler and the partition estimator. The compiler assigns operators in the DFG to PEs, primary inputs and outputs to memories, and specifies whether an interconnect or a memory resource is to be used for data transfers between PEs. The compiler generates new bindings till a constraint satisfying binding is found. The best binding found by the compiler is then input to a high level synthesis tool like Asserta. The HLS tool converts the DFG descriptions to multiple datapath-controller pairs corresponding to each PE.

The cost of a binding is the sum of interconnect, area, and memory violations. The latency of the design is the sum the input, compute, data transfer, and output latencies. The evaluation rules for the resource violations and latencies are presented in the future sections.

The following subsection describes the target architecture onto which the applications are mapped.

7.2.1 Target Architecture

The target architecture for the designs is the WildFire family [27] of reconfigurable boards. They consist of a set of user-programmable FPGAs and one controller FPGA. The devices are arranged as a linear array with a 36-bit wide direct line between adjacent FPGAs. In addition to the direct lines between FPGAs, there is a programmable partial crossbar interconnection network. Each FPGA has a physical memory that is connected to it. These memories are
accessible to other PEs through the programmable interconnect network. The partitioner views the memory as a "single shared memory" and all memory accesses are serialized.

The four members in the family of these boards are – WildOne (1 FPGA), WildForce (4 FPGAs), WildChild (8 FPGAs), and WildFire (16 FPGAs). The FPGA devices on these boards are the Xilinx 4000 series chips. Since the underlying architecture of these devices is the same, instances of these boards are represented using different design files. Figure 7.3 shows the WildForce architecture.

The following section details the estimation methodology.
7.3 Estimation Methodology

Figure 7.4 shows the estimator framework and the inputs and outputs of the estimator. The architecture, application, and the evaluator are the three main components of the model. The areas of the PEs, the channel size between PEs, and the two PEs that are connected by a channel (pe₁, pe₂) are the architecture specific inputs to the model.

The RC compiler that binds operations to the PEs assigns a pe number to each of the operations in the application. The compiler also assigns sizes to each of the flows or dependency edges between operations. Each flow f has a source_pe and a list of sink_pes. The source and sink pes for each flow are inferred internally by the estimator from the operator-pe binding information. The compiler also assigns a boolean attribute wired to each of the flows, which when set to false refers to a memory transfer, and when set to true refers to a wired transfer. This is useful to estimate the latency of a transfer. The outputs of the estimator are also shown in the figure. They are the area_penalty, interconnect_penalty, and the latency of a binding.

In the following subsections we describe the estimation algorithms for each of the outputs of the estimator.

7.3.1 Area Penalty

The area_penalty of a binding is given by

\[ \sum_{\text{pe}} \left( \text{used_area} - \text{area} \right) / \text{area} \]

We estimate the used_area attribute for each pe. The used_area in a pe is the sum of the operator_area and the

Figure 7.4: Estimator Framework
**mux_area** for the pe. The operation graph is scheduled using the as soon as possible (ASAP) schedule that assigns **tstep** numbers to each operation. The operator_area on a pe is the sum of the areas occupied by operators of all sizes and types. The area of an operator of a given type and size is given by –

*The maximum number of operators of a given type and size required in any tstep on a given pe.*

The **mux_area** is estimated to be the *sum of the sizes of all operators on a given pe.*

### 7.3.2 Interconnect Penalty

The interconnect penalty is given by –

\[
\sum_{c=1}^{\#chs} \left( \text{used} \text{ bits} - \text{size} \right) / \text{size}
\]

The **used_bits** in a channel c, is the sum of the bits required by flows that are cut by the channel c. A flow is cut by a channel if the source and any destination operator are assigned to pes connected by channel c and the **wired** attribute of the flow f is set to true by the compiler. The used_bits for a particular channel c is given by,

- **used_bits** : If source_pe of f is the same pe (pe_1) connected by channel c, and any one of the sink_pes of f is assigned to pe_2 (pe_1), and the flow is wired used_bits = size of flow.

- **total_used_bits** : sum of **used_bits** over all flows is the total used_bits for a given channel c.

### 7.3.3 Latency

The latency of a binding is the sum of the primary input, compute, data transfer, and primary output latencies. The compute, input, and output latencies are evaluated within the dfg module. The data transfer latency depends on whether the transfer is through memory, or through the interconnect.

**Primary Input/Output Latency**

The **PLLatency** is *PrimaryInputs * MemoryReadTime*.

The **POLatency** is *PrimaryOutputs * MemoryWriteTime*.

**Compute Latency**

The **computeLatency** of a DFG is the length of the schedule. This is the maximum tstep assigned to an operator in the input dfg.
**Data Transfer Latency**

The data transfer latency is the sum of `memory.transfer.latency` and the `wire.transfer.latency`. If there is a data dependency through memory between operators in different PEs, additional states are introduced for the corresponding memory write and memory read operations. These additional timesteps are modeled as the memory transfer latencies. The `memory.transfer.latency` for a particular flow `f`, is computed as follows –

- **Foreach pe `p`, if any of the sink.pes of the flow `f` is `p`, and the source.pe is not `p`, and `f` is not wired, then add memory read time + memory write time.**

- **The sum of the above step over all pes gives the memory transfer latency for a given flow.**

The sum over all flows gives the total memory transfer latency.

When there is a data dependency through the board interconnect resources, two additional states are introduced by the synthesis tool. These additional states are modeled as the wire transfer latencies. These additional states are introduced between every pair of PEs that require a data transfer between them using the board interconnect in a given timestep. The `wire.transfer.latency` is estimated as follows –

- **For every cstep `cs`, for every pair of distinct pes `p1`, `p2`, for each operator `o` assigned to cstep `cs`, if a wired flow `f` that connects operator `o` from `p1` to `p2` exists, we add two cycles at the end of cstep `cs`**

- **The above step over all csteps gives the total number of cycles for the wired transfer.**

The following section describes the different components in the PDL+ model of the estimator.

### 7.4 PDL+ Implementation

The PDL+ model of the estimator is composed of three main components. They are the architecture model, application model, and the evaluate model. The architecture model represents the target architecture and the board-level attributes required to estimate the performance of a binding. The application model represents the design whose performance is to be estimated. The application model also has information about the mapping and its evaluation rules account for the synthesis optimizations.

Most of the evaluation rules require the application and architecture models to interact with each other. In PDL+, the scope of attributes of a component is restricted to within the component definition, and to the parent component that contains these components. The architecture and application model are however independent of each other and their components do not have a parent-child relationship. The evaluate model therefore contains a single module that contains components from both these models. These design components interact from within the evaluate module.
We describe these models in the following subsections. We also illustrate the interaction between components through the evaluate model.

### 7.4.1 Architecture Model: rc.pdl

The cost of a binding is the *normalized sum* of its area, interconnect, and memory violations. Through the rest of the model we assume that sufficient memory is always available and hence do not model memory violations. Since the partitioner views the memories as a global shared memory, synchronization signals need to be exchanged between PEs for the memory access. The estimator does not account for the interconnect resources required for the synchronization. This is offset by ignoring the programmable interconnect resources on the board.

The declarations of the `pe` module and the `channel` carrier are shown in figure 7.5. The `pe` module represents the programmable elements of the board and the channel models the direct interconnect between the PEs. The `plus_area` attribute has been defined to illustrate the interaction between the architecture and application models from within the evaluate model.

```pdl
module pe
attributes
  plus_area : int;
end module;

carrier channel
end carrier;
```

Figure 7.5: **Architecture Model**

The attributes and evaluation rules of these components are described in the section 7.5.

### 7.4.2 Application Model: dfg.pdl

The input to the application model is an operation graph. The operators are represented by the `op` module and the entire graph is represented by the `dfg` module. Their declarations are very similar to the `op` and behavior modules of the previous chapter. The edges of the graph are however represented by the `carrier flow`. The binding information provided by the RC compiler is represented as primitive attributes of the `op` and `flow` components. Each operator has a `pe_id` attribute and the flows have attributes that specify their mapping information (memory or interconnect). Figure 7.6 shows the components in the application model and sample attributes to illustrate the lateral propagation of attribute values between modules and carriers using ports.

The ports `opports` are used to connect the operator modules and the flow carriers. They are used to propagate attributes laterally between the modules and carriers. Consider a case when an attribute `pe_id` has to be propagated...
port opport
attributes
  pe_id : int;
end port;

module op
ports
  inputs[ ] : opport;
  output : opport;
attributes
  primitive pe_id : int;
  area : int;
rules
  output'pe_id = pe_id;
end module;

carrier flow
ports
  input: opport;
  outputs[ ] : opport;
attributes
  pe_id : int;
rules
  pe_id = input'pe_id;
end carrier;

module dfg
ports
  inputs[ ] : opport;
  outputs[ ] : opport;
carriers
  flows[ ] : flow;
modules
  ops[ ] : op;
end module;

Figure 7.6: Application Model
from an instance \textit{op1} of the op module, to an instance \textit{flow1} of the flow carrier. If the output port of \textit{op1} is connected to the input of \textit{flow1} in the design file. This is specified in the design file as, \texttt{net op1.output, flow1.input;}

The evaluation rules in figure 7.6 illustrate the lateral propagation of the \textit{pe.jd} attribute.

### 7.4.3 Evaluate Model : bind.pdl

Most of the evaluation rules require the application model components to interact with the architecture model components. Consider a situation where an attribute \textit{plus.area} of the \textit{pe} module of the architecture model needs to access the \textit{area} attribute of the \textit{op} module in the application model. This is not possible through the application or architecture models. In order to specify such rules, we define the \textit{evaluate} model, that contains a single \textit{evaluate} module and the required design components from the architecture and application models. The \textit{plus.area} attribute of the \textit{pe} module and the \textit{area} attribute of the \textit{op} module are visible in the scope of the evaluate module. The \textit{plus.area} attribute can be propagated \textit{top-down} to the corresponding \textit{pe} as shown by the rule in figure 7.7

```pdl
module evaluate
  modules
    pes[] : pe;
    ops[] : op;
    dfg[] : dfg;
  carriers
    channels[] : channel;
    flows[] : flow;
  rules
    pes[] +=
      sum(int(foreach o : op in ops
        {o.area}));
end module;
```

Figure 7.7: Evaluate Model

We now describe the evaluation rules associated with each of these components and illustrate the interaction between these rules.

### 7.5 Evaluation Rules

The cost of a binding is the sum of the \textit{interconnect} and \textit{area} penalties foreach \textit{pe}. In this section, we describe the evaluation rules that estimate the used area on the \textit{pes}, and the \textit{channels} required between each \textit{pe}. The estimated resource count and the available resources are used to obtain the corresponding penalties.

The latency of a binding is the sum of the primary input, compute, data transfer, and primary output latencies. The compute, input, and output latencies are evaluated within the \textit{dfg} module. The data transfer latency depends on
whether the transfer is through memory, or through the interconnect. The evaluation rules for these latencies are described in the following subsections.

We now describe the attributes and evaluation rules of each of the components. The estimation algorithms presented in these sections were developed in conjunction with Srinivasan [28]. A detailed description of the algorithms can be found in Srinivasan’s thesis.

### 7.5.1 Module: pe

Each *pe* module has an *area*-*penalty* attribute associated with it and each *channel* has an *interconnect*-*penalty* attribute. The module *pe*, its attributes, and evaluation rules are shown in figure 7.8.

```plaintext
module pe
    attributes
        primitive id: int;      -- unique integer identification
        primitive area: int;   -- CLBs
        used_area: int;
        area_penalty: real;
        plus_area : int;
        minus_area : int;
        multiply_area : int;
        mux_area : int;
    rules
        used_area = plus_area+minus_area+multiply_area+mux_area;
        area_penalty = if used_area <= area then 0
                        else ((used_area - area)/area)
        endif;
end module;
```

Figure 7.8: Module: pe

The *used_area* attribute is the estimated area occupied by the operators on a particular *pe*. A given binding is synthesized using Asserta, followed by logic and layout synthesis. The area estimation rules are specified in the evaluate module and are propagated top-down to the corresponding *pe* modules.

### 7.5.2 Carrier: channel

The primitive attributes specify a unique id, bandwidth, and the two PEs that are connected by the channel. Here too the evaluation rule for the *used_bits* attribute is not specified in the channel declaration and is inherited from the evaluate module. Figure 7.9 shows the definition of the *channel* carrier component.

Figure 7.10 shows a sample design file that specifies the wildforce architecture in the hierarchical netlist format.
carrier channel

attributes

primitive id: int;
primitive size: int;       --  bitwidth of the channel
primitive pe_1: int;       --  one side of the channel
primitive pe_2: int;       --  other side of the channel
used_bits: int;
interconnect_penalty: real;

rules

interconnect_penalty =
  if used_bits + <= size
    then 0
  else ((used_bits - size)/size)
endif;

end carrier;

Figure 7.9: **Carrier: channel**

(HNF). A cell definition in the HNF file represents a parametrized declaration of a component instance. The design components are then explicitly instantiated with appropriate attribute values. The cell definition is useful for instantiating components with default parameters and can be reused in other design files. The Wildforce board instantiated in the design file has four XC4025 FPGAs (1024 CLBs). A detailed description of the HNF format can be found in the HNF user guide [15].

### 7.5.3 Module: op

The op module represents an operation in the input DFG. Each operation is assigned to a PE by the compiler and this is represented by the primitive pe attribute. The rest of the attributes are identical to those described in the previous chapter. The partitioned DFG undergoes high level synthesis and the schedule information tstep is required to account for the sharing of operators that would result due to the HLS process. Unlike the previous chapter this is not a primitive attribute and is computed as the as soon as possible (ASAP) value for the operator. The evaluation rule is shown in figure 7.11.

The pe_id, tstep, and op_id attributes need to be propagated between the op and flow components. This is achieved through the port opport. The declaration of opport is shown in figure 7.12 Each of the ports has a tstep attribute associated with it. The evaluation proceeds from the primary input ports whose tstep numbers are set to 0. The operators perform single cycle operations and are scheduled to the earliest possible time step (one tstep after the max tstep of the input ports). Thus the tstep numbers are propagated laterally.
cell module pe = pe_cell(peid:int=-1, pearea:int=-1)
attributes
id = peid;
area = pearea;
end module;

cell carrier channel = channel_cell(cid:int=-1, csize:int=-1, cpe_1:peid=-1, cpe_2:peid=-1)
attributes
id = cid;
size = csize;
pe_1 = cpe_1;
pe_2 = cpe_2;
end carrier;

-- RC Model
pe1 : pe_cell(peid = 1, pearea = 1024);
pe2 : pe_cell(peid = 2, pearea = 1024);
pe3 : pe_cell(peid = 3, pearea = 1024);
pe4 : pe_cell(peid = 4, pearea = 1024);

c_1,2 : channel_cell(cid = 1, csize = 36, cpe_1 = 1, cpe_2 = 2);
c_2,3 : channel_cell(cid = 2, csize = 36, cpe_1 = 2, cpe_2 = 3);
c_3,4 : channel_cell(cid = 3, csize = 36, cpe_1 = 3, cpe_2 = 4);

Figure 7.10: The Wildforce Board Architecture

7.5.4 Carrier: Flow

The edges in the DFG are modeled using the carrier flows and the PDL+ description is shown in figure 7.13. Each flow has a size and a wired (mapping information) attribute associated with it. The source and destination pe numbers are propagated between the operators and the flows through the ports. This information is used to determine if a flow component was cut by the given binding. If it is cut, the wired attribute is used to determine if the data transfer is through memory or interconnect.

An edge in the dfg has a single source and may have multiple sinks. In the flow carrier, the source pe number is stored in the source.pe attribute and the sink pe numbers are stored in the list sink.pes[]. This is used to calculate the interconnect.penalty of the channel as shown in subsection 7.5.6.

The tstep numbers for the outputs is the same as the input tstep numbers. The latencies due to the transfer of data are modeled in the dfg and evaluate modules.

7.5.5 Module: dfg

Figure 7.14 shows the attributes and evaluation rules of the dfg module. The memory read operation on the board require 3 clocks and a memory write operation requires 1 clock. Therefore the PI_latency and PO_latency rules are
module op
  ports
    inputs{ } : opport;
    output : opport;

  attributes
    primitive id : opid;
    primitive opcode : opcodes;
    primitive size : int;
    primitive pe : peid;
    area : int; -- clbs
    tstep : int;

  rules
    area = get_area(opcode, size); -- library manager estimation functions
    inputs{ }'op = id;
    output'op = id;
    inputs{ }'pe = pe;
    output'pe = pe;
    -- ASAP computation
    tstep = max_int(foreach i:opport in inputs {i'tstep}) + 1;
    output'tstep = tstep;

end module;

Figure 7.11: Module: op
port opportun
attributes
  pe : peid;
  op : opid;
  tstep : int;
end port;

Figure 7.12: Port: opportun

carrier flow
ports
  input: opportun;
  outputs{ }; opportun;
attributes
  primitive size: int; -- bitwidth
  primitive wired: boolean; -- true: bind to channel if cut
  -- false: bind to memory if cut; false is the default.
source_pe: peid;
sink_pes[ ]; peid;
rules
  source_pe = input’pe;
  sink_pes = foreach x:opportun in outputs{ x’pe };
  outputs{ }’tstep = input’tstep;
end carrier;

Figure 7.13: Carrier: flow

defined as shown in the figure. The sizes attribute for each of the operator types is required to determine the sharing of resources on a given PE. This is propagated bottom-up to the evaluate module. The transfer latency attribute rule is defined in the evaluate module.

7.5.6 Module: evaluate

The evaluate module and its attributes and evaluation rules are shown in figure 7.15. The evaluate module has access to all attributes of its children. The rules for the attributes of the child components that are propagated top-down from the evaluate module are described in the remainder of this section.

Attribute plus_area (pe module)

The used_area attribute of the PE module is defined as shown in figure 7.8. The rules for each of the component areas (plus_area, minus_area...) are defined in the evaluate module. The evaluation rule for the plus_area attribute is
module dfg
  ports  
    inputs\{\} : opport;  
    outputs\{\} : opport;  
  carriers  
    flows\{\} : flow;  
  modules  
    ops\{\} : op;  

  attributes  
    ASAP\_length : int;  
    PI\_latency : int;  
    PO\_latency : int;  
    transfer\_latency : int;  
    plus\_sizes\{\} : int;  
    minus\_sizes\{\} : int;  
    multiply\_sizes\{\} : int;  
    divide\_sizes\{\} : int;  

  rules  
    -- ASAP Computation  
      inputs\{\}'tstep = 0;  
    -- Latency  
      ASAP\_length = max\_int(foreach o:opport in outputs \{o'tstep\});  
      PL\_latency = #inputs*3;  
      PO\_latency = #outputs*1;  
      plus\_sizes =  
        begin  
          a\{\} : int := \{\};  
          foreach op:operation in operations  
            { if op'opcode == plus then a := a union \{op'size\}; endif; }  
          return a;  
        end;  
    end module;

Figure 7.14: Module: dfg
module evaluate
modules
  pes[]: pe;
  ops[]: op;
  dfg: dfg;
carriers
  channels[]: channel;
  flows[]: flow;
attributes
  area_penalty: real;
  interconnect_penalty: real;
  total_penalty: real;
  latency: int;
rules
  area_penalty = sum_real(foreach p:pe in pes {p'area_penalty});
  interconnect_penalty = sum_real(foreach c:channel in channels {c'interconnect_penalty});
  total_penalty = area_penalty + interconnect_penalty;
  latency = dfg'transfer_latency + dfg'ASAP_length + dfg'PL latency + dfg'PO_latency;
end module;

Figure 7.15: The Evaluate Module

shown in figure 7.16. Similar rules are defined for the other component areas as well.

The estimated number of operators of a given type and size required in a design, is the maximum number of these operators required in any tstep on a given pe.

The construct pes[p]'plus_area applies the evaluation rule for the plus_area attribute forall pes in the evaluate module. Where p, refers to the particular pe in the iteration. The attributes of the pe in the current iteration can be accessed through the p’ notation.

-- area penalty computation
-- to find sharing with NO operator folding

pes[p]'plus_area = sum_int( foreach i:int in dfg'plus_sizes
  { max_int( foreach c:int in 1 to dfg'ASAP_length
    { sum_int( foreach op:op in ops
      { if op'opcode == plus and op'tstep == c and op'pe == p'id and op'size == i
        then op'area else 0 endif }) }) });

Figure 7.16: Attribute: plus_area in each pe
pes[p]'mux_area = sum_int(\texttt{foreach} op:op in ops
    \{ if op'pe == p'id then (op'size) else 0 endif \});

--- interconnect penalty computation

channels[c]'used_bits =
    \text{sum_int(\texttt{foreach} e:flow in flows
        \{ if ((c'pe_1 == e'source_pe and member(c'pe_2, e'sink_pes)) or
            (c'pe_2 == e'source_pe and member(c'pe_1, e'sink_pes))) and e'wired
            then e'size
                else 0
                endif \});}

--- Attribute: \text{used\_bits in channels}

Attribute: \text{mux\_area (in pe)}

Figure 7.17 shows the rule for the \textit{mux\_area} attribute. The \textit{mux\_area} is approximated to be the sum of the sizes of all operators on a given pe.

Attribute \textit{used\_bits (in channel)}

The \textit{used\_bits} for a channel \(c\) is obtained by the rule shown in figure 7.18. Each channel \(c\), has attributes \text{pe_1} and \text{pe_2} that represent the pes that are connected through \(c\) and the flows have the peids of their source (source\_pe) and sinks (sink\_pes). For each flow \(e\), if the source\_pe is the same as \text{pe_1}(\text{pe_2}) and any one of the sink\_pes is in \text{pe_2}(\text{pe_1}) and \(f\) has been mapped to the interconnect (wired=true), then bits required by flow \(e\) in channel \(c\) is the size of \(e\).

The sum over all flows gives the \textit{used\_bits} attribute for the channel.

The function \textit{member} is shown in figure 7.19

\begin{verbatim}
function member(i: int; l[]:int):boolean := false
begin
foreach j:int in 1 to #l
    \{ if l[j] == i then return true; endif \}
return false;
end;
\end{verbatim}

--- Function: member

89
let wire_transfer_latency: int =
begin
wl:int;
k:int;
wl := 0;

foreach i:int in 1 to dfg’ASAP_length-1 { -- look at every level
    foreach p1:peid in 1 to #(pes){ -- for each pair of PEs
        foreach p2:peid in 1 to #(pes){
            k := 0;
            foreach o:op in ops{
                if o’tstep == i
                then foreach f:flow in flows {
                    if f’wired and (f’source_op == o’id) -- if it is wired connection
                    then
                        if (not(p1==p2) and f’source_pe == p1 and member(p2, f’sink_pes))
                        then k := 2; -- then we add two wait cycles between those levels.
                    endif;
                endif;
            }
        }
        wl := wl + k;
    }
}
return wl;
end;

Figure 7.20: Attribute: wire_transfer_latency

Attribute: transfer_latency (in dfg)

The dfg transfer_latency attribute is the sum of all wire transfer latencies and memory transfer latencies.
dfg’transfer_latency = wire_transfer_latency + memory_transfer_latency;

Attribute: wire_transfer_latency

Figure 7.20 shows the evaluation rule for wire transfer latencies.
let memory_transfer_latency: int =
sum_int( foreach f:flow in flows { -- look at each flow 
begin 
    k: int;
    k := 0;
    foreach j:int in 1 to #(pes) -- look at each PE
        { if member(j, f'sink_pes) and (j != f'source_pe) -- if the PE is a sink but not the source
            then if not(f'wired) -- if f is memoried
                then k := k + 3 + 1; -- then add 3 cycles to read + 1 cycle to write
            endif;
        };
    return k;
end
});

Figure 7.21: Attribute: memory_transfer_latency

<table>
<thead>
<tr>
<th>Design</th>
<th>CLBs</th>
<th>Latency</th>
<th>Nodes</th>
<th>Flows</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT1D</td>
<td>736</td>
<td>74</td>
<td>40</td>
<td>16</td>
</tr>
<tr>
<td>FFT2D</td>
<td>1652</td>
<td>80</td>
<td>88</td>
<td>112</td>
</tr>
<tr>
<td>Ellip</td>
<td>1176</td>
<td>48</td>
<td>36</td>
<td>49</td>
</tr>
<tr>
<td>Reverb</td>
<td>1424</td>
<td>34</td>
<td>22</td>
<td>24</td>
</tr>
<tr>
<td>FIR-16tap</td>
<td>1116</td>
<td>78</td>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>DCT4x4</td>
<td>4108</td>
<td>210</td>
<td>224</td>
<td>256</td>
</tr>
<tr>
<td>DCT8x8</td>
<td>14032</td>
<td>835</td>
<td>1920</td>
<td>2304</td>
</tr>
</tbody>
</table>

Table 7.1: Latency and Area for Single FPGA

Attribute: memory_transfer_latency

Figure 7.21 shows the evaluation rule for the memory_transfer_latency attribute.

Based on the rules, the compiler generates the evaluation sequence and the executable model can be queried for the evaluated attributes. We present some of the results obtained from the estimator in the following section.

7.6 Results

We estimated the latency and resources for each of the circuits for a single FPGA device. These values are shown in table 7.1. The latency estimate is a lower bound for a multi-FPGA device.
A genetic algorithm [29] based compiler was integrated with the partition estimator. The board and its attributes are instantiated in the design file. The application is represented as an instance of the dfg module. The model and design file are loaded through the API in the partitioner. The partitioner then sets the other primitive attributes of the operations (like pe_id) and the model is evaluated. Based on the estimates for latency and resource violations, the compiler generates new populations that are evaluated by the estimator. The compiler assigns an initial random binding to all operators in the data flow graph. Using the operations of cross-over and mutation the compiler generates new bindings. Srinivasan’s thesis [28] has a detailed description of the GA based compiler. The best binding found by the compiler was then input to the HLS tool, Asserta. The HLS tool converts the DFG descriptions to multiple datapath-controller pairs corresponding to each PE.

The following figures show the trend of decreasing latencies with new generations as obtained from the estimator. The line in the graph represents the lower bound value for the latencies. In figure 7.22 and 7.23 the target architecture for the designs is a wildforce board with for XC4013 devices (576 CLBs). The designs are FFT1D and FIR16tap.

![FFT1D Latency Vs Generations](image)

**Figure 7.22: FFT1D Wildforce XC4013**

Figures 7.24 and 7.25 show the trend of the latencies over generations for the DCT4x4 design on the Wildchild board with 8 XC4036 (1296 CLBs) devices and on the wildfire board with 16 XC4025 (1024 CLBs) devices.

Table 7.2 compares the results between the estimated and actual values.
Table 7.2: Estimated Vs Actual WildForce board

<table>
<thead>
<tr>
<th>Example</th>
<th>Area(Est)</th>
<th>Area(Actual)</th>
<th>Latency(Est)</th>
<th>Latency(Actual)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector N=15,E=14</td>
<td>139(P1),119(P2)</td>
<td>158(P1),135(P2)</td>
<td>64</td>
<td>65</td>
</tr>
<tr>
<td>Stats N=23,E=22</td>
<td>141(P1),93(P2)</td>
<td>165(P1),102(P2)</td>
<td>69</td>
<td>72</td>
</tr>
</tbody>
</table>

Figure 7.23: FIR16 Wildforce XC4013

7.7 Summary

In this chapter we presented a methodology to model a target architecture, an application, and a mapping process for the designs. In the partition estimator the application and architecture models interact through the evaluate model. The rules that required information about the binding and the architecture were defined in the evaluate model. Using the partition estimator model, we can predict the performance of a design on a family of board architectures.

The model illustrates top-down, bottom-up, and lateral propagation mechanisms of attributes between components. The design component carrier was used in the model and the use of ports to propagate attributes between design components was described.

The partition and behavioral estimator model the performance attributes at a higher level of abstraction and are used earlier in the design cycle. In the next chapter we present a delay estimator model that is used to predict the delays of a design at a later stage in the design flow.
Figure 7.24: DCT4x4 Wildchild (XC4036)

Figure 7.25: DCT4x4 WildFire (XC4025)
Chapter 8

FPGA Delay Estimator

PDL+ descriptions of systems are independent of the level of abstraction. The models described in the previous chapters are examples of modeling at a relatively higher level of abstraction. These performance models correspond to the behavioral level in a design process. The FPGA delay estimator models performance at the physical or layout level, which is at a lower level of abstraction. This chapter deals with the usage context of the model, its development methodology, and we present the results obtained from the estimator.

From the language and modeling methodology point of view, the estimator illustrates lateral attribute propagation and the use of arrays to represent containment of design components. PDL+ provides two methods of attribute propagation between components - hierarchical and lateral. So far, the main method of attribute propagation has been hierarchical, where the attributes of the parent module are passed on to its constituent modules (top-down) or vice versa (bottom-up). The delay estimator makes use of a combination of both hierarchical and lateral attribute propagation. In the latter case, the attributes’ evaluation depends on the exact connectivity information of the design. Though hierarchical propagation is useful in most cases, lateral propagation provides a convenient method to model attributes like delays, longest path of a design, etc. We now describe the usage context of the delay estimator.

8.1 Usage Context

The estimator models the delay through netlists for the Xilinx 4000 series FPGAs. A description of the CLBs and interconnect elements of the XC4000 series FPGAs was provided in chapter 4. The usage context of the delay estimator is shown in figure 8.1. A placement tool provides routed netlist information to the estimator and it predicts the delays through the nets, using delay models of the Xilinx architecture. Based on the estimated delay and the constraints provided to the routing tool, the current placement and routing information may either be accepted, or changed and input again to the estimator.
8.2 Estimation Methodology

The architecture of the XC4000 devices has been described in section 4.2.1. Figure 4.2 shows the internals of an xc4000e device CLB. There are five different paths from the inputs of the clbs to the outputs in a XC4000e series device. These are Tilo, Tiho, Thh0o, Thh10, and Thh20. Each of these paths have different delays and the delay values can be found in the databook [9]. The delay through the clb depends on the path that is activated. Table 8.1 shows the delays through different paths within the CLB for a XC4000e -3 device. This data was obtained from the Xilinx Data book.

<table>
<thead>
<tr>
<th>Type</th>
<th>Net Name</th>
<th>Delay(ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F/G-X/Y</td>
<td>Tilo</td>
<td>2.0</td>
</tr>
<tr>
<td>F/G-H-X/Y</td>
<td>Tiho</td>
<td>4.3</td>
</tr>
<tr>
<td>C1-H-X/Y</td>
<td>Thh0o</td>
<td>3.6</td>
</tr>
<tr>
<td>C2-H-X/Y</td>
<td>Thh1o</td>
<td>3.6</td>
</tr>
<tr>
<td>C3-H-X/Y</td>
<td>Thh2o</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Table 8.1: Delays Through the CLB

We model the intra-clb routing delays and the inter-clb net delays in the estimator.

There are three basic routing resources that are available in a XC4000e device. There are : single lines (sl), double lines (dl), and long lines (ll). Any two segments are joined using programmable interconnect points (pips).

Xu and Kurdahi [10] modeled the delays through CLBs and nets for the Xilinx 4000 series device. They obtained CLB delays from the databooks and approximated the net delays in terms of the segment delays and pips. Based on the placement information, they predict the routing resources required. They also modeled the affect of loading on a net as a heuristic fanout factor. Using the heuristic fanout factor and the delays through the segments they estimate the delay through the nets. The individual segment delay values is not available from the data books. Therefore they performed experiments and reverse engineered these values using the Xilinx Xdelay tool. The values that they obtained were for a Xilinx 4000 series FPGA.

The FPGA delay estimator uses a similar approach to predict the delays for the XC4000e series devices. We reverse
<table>
<thead>
<tr>
<th>Type</th>
<th>Delay(ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single line (sl)</td>
<td>0.3</td>
</tr>
<tr>
<td>Double line (dl)</td>
<td>0.3</td>
</tr>
<tr>
<td>Long line (ll)</td>
<td>0.36</td>
</tr>
<tr>
<td>PIP</td>
<td>0.4</td>
</tr>
<tr>
<td>Fanout factor</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Table 8.2: **Element Delay Values**

engineered the basic element values using the *Epic* layout editor tool. Epic provides a graphical view of a design after placement and routing. The layout of a number of nets were studied using the layout editor and using this we obtained the segment delay information. We also studied the affect of fanout on the delay and obtained a fanout factor. These values are shown in table 8.2.

We illustrate the net delay estimation method for nets with no loading affects in figure 8.2. This method was developed by Xu and Kurdahi. The delay for the net shown in the figure is $3 \cdot \text{dl} + 2 \cdot \text{pip} + Tilo$.

Figure 8.2: **Sample net delay**

Figure 8.3 shows the affect of fanout on the delay of the nets. Let Dad, Dac and Dab be the delays from the point A to points D,C, and B respectively, without considering the fanout factor. The effective delay (Dadeff) from A to D considering loading due to the fanouts is $Dad + (Dab + Dac) / \text{fanout factor}$. Therefore, the effective delay is the sum of the delays without the fanout affect between the two points and the sum of all delays (without fanout effects) for the other fanout points divided by the fanout factor. This heuristic gives a relatively accurate measure for the delays through the nets. More information about this method can be found in [10].

We estimate the clock speed of the design. The clock speed is the maximum between the worst case flipflop to flipflop delay, or the worst combinational path delay.

We model the elements in the FPGA architecture in the estimator and the following sections describe the PDL+
\[ \text{Dad}_{\text{eff}} = \text{Dad} + (\text{Dab}+\text{Dac}) / \text{fanout factor} \]
\[ \text{Dac}_{\text{eff}} = \text{Dac} + (\text{Dab}+\text{Dad}) / \text{fanout factor} \]
\[ \text{Dab}_{\text{eff}} = \text{Dab} + (\text{Dad}+\text{Dac}) / \text{fanout factor} \]

Figure 8.3: *Delay with fanout*

8.3 The design module

The model is built hierarchically from its constituent modules. The design module is at the top of the hierarchy and is composed of clbs and wires. The clbs represent the configurable logic blocks and the wires model the nets in the design. Given a netlist, the estimator predicts the maximum delay. This is defined as the maximum between the flipflop to flipflop delays and the maximum combinational delay.

Figure 8.4 shows the design module and its evaluation rules. We describe the clb and wire components in the following subsections.

8.3.1 CLB module

As seen from the architecture of the device, the CLB consists of function generators, flipflops and multiplexers that activate the desired internal connections. A set of *internal wires* model the intra-clb connections. Figure 8.5 shows the clb module declaration, its attributes, and evaluation rules. The maximum delay at the flipflop is propagated bottom-up to the design module from each of the flipflop modules in a clb.

Figure 8.6 shows the PDL+ declarations and evaluation rules for the components that constitute the clb module. All internal wires in the design are assigned to anyone of the allowed types (Tilo, Tiho, ...). This is stored in the primitive attribute *internal wire type*. The function calculate_wire_delay returns the corresponding delay for the path shown in table 8.1. The ports have an attribute delay that is laterally propagated between modules and carriers. The delay at a port represents the combinational delay of the port. The delay at the output ports depend on the delays at the input ports. However, in the case of flipflops it is possible that a feedback path exists between the outputs and inputs.
module design
ports
    inputs[] : io_port;
    outputs[] : io_port;
modules
    clbs[] : clb;
    wires[] : wire;
attributes
    max_comb_delay : real;
    max_delay : real;
    max_ffdelay : real;
rules
    inputs[] ’delay = 0.0;
    max_comb_delay = max_real(foreach o : io_port in outputs{ o’delay });
    max_ffdelay = max_real(foreach clb : clb in clbs{ clb’delayff });
    max_delay = max(max_comb_delay,max_ffdelay);
end module;

Figure 8.4: The Toplevel design Module

These evaluation rules can not be applied in such cases because they would result in a cyclic dependency and PDL+ does not an attribute to depend on itself. Therefore we represent the delays through flip flops as shown in figure 8.6.

We set the output delay at the flipflop to zero and propagate the input delay of the flipflop through the \texttt{ffdelay} attribute. The maximum \texttt{ffdelay} for all flipflops in a clb is propagated as the \texttt{delayff} attribute of the clb module. This way we obtain the maximum delay between any two flipflops.

The user defined types in the model are shown below.

type
delay_types : \texttt{enum}\{dl,sl,ll,pip\};
internal_wire_types : \texttt{enum}\{Tilo, Tiho, Thh0o, Thh1o, Thh2o\};
components[] : delay_types;
complist[] : components;

8.3.2 Carrier : wire

The nets connecting CLBs are modeled by the carrier \texttt{wire}. These wires are multi terminal nets and are modeled as a single input and an \texttt{array} of outputs. Arrays can be used to define containment of design components. Here the number of output nodes for a net is set to three. We use arrays instead of the usual sets because the evaluation rules
module clb
ports
inputs[ ] : io_port;
outputs{ } : io_port;
carriers
internal_wires{ } : internal_wire;
modules
function_generators[ ] : function_generator;
flip_flops{ } : flip_flop;
attributes
delayff : real ;
rules
delayff = max_real(foreach ff : flip_flop in flip_flops { ff`ffdelay });
end module;

Figure 8.5: The clb Module

for each of the output ports of the wire are different. We illustrate the evaluation rules of the carrier wire in figure 8.7. According to this definition a wire can have a fanout of atmost three. This number can be set to any integer value.

The delay_components attribute is a list of lists of segment types. The inner list represents the segment types from the input to each output and the list of these lists for all outputs of the wire is represented by the delay_components attribute. This is a primitive attribute.

From figure 8.3 we can see that the delay at an output port depends on the delays at other ports connected by a given wire. This is shown in figure 8.7

We now describe the functions that are used in the evaluation rules.

8.3.3 Functions Used

The calculate_delay function uses the segment delay types and the element values as shown in table 8.2 and returns the delay of the wire.

In the following section we present a comparison of the results obtained from the estimator and the EPIC layout tools. The epic tool is a part of the suite of M1 tools for Xilinx FPGA synthesis.
module function_generator
ports
inputs{} : io_port;
output : io_port;

rules
output’delay = max(real(foreach i : io_port in inputs {i’delay}));
end module;

carrier internal_wire
ports
input :io_port;
output :io_port;
attributes
primitive internal_wire_type : internal_wire_types;
internal_wire_delay : real;

rules
internal_wire_delay = calculate_wire_delay(internal_wire_type);
output’delay = input’delay + internal_wire_delay;
end carrier;

module flip_flop
ports
input :io_port;
output :io_port;

attributes
primitive delay : real;
ffdelay : real;

rules
output’delay = 0.0;
ffdelay = input’delay + delay;
end module;

Figure 8.6: Components in the CLB Module
port io_port
attributes
delay : real;
end port;

carrier wire
ports
input : io_port;
attributes
primitive delay_components: complist;
rules
let fanout_factor : real = 3.5;
let delaylist[] : real =
begin
  a[] : real;
  foreach i:int in 1 to #delay_components
  {
    a := append(a,calculate_delay(delay_components[i]));
  }
  return a;
end;
outputs[x]'delay = input'delay +
  sum_real(foreach i : int in 1 to #delaylist { if i!=x then delaylist[i] else 0 endif })/fanout_factor +
delaylist[x];
end carrier;

Figure 8.7: Carrier wire
function calculate_delay (delays[] : delay_types) : real := 0.0
begin
    temp : real := 0.0;
    foreach p : int in #(delays) downto 1
    {
        case delays[p] of
            dl : temp := temp+0.3;
            sl : temp := temp+0.3;
            ll : temp := temp+0.36;
            pw : temp := temp+0.0;
            pip : temp := temp+0.4;
            ps : temp := temp+0.0;
            others : temp := temp+0.0;
        end case;
    }
    return temp;
end;

Figure 8.8: Function Definitions

8.4 Results

The Epic layout editor shows the routing and CLB resources used by a particular design. We studied the layout of different designs and represented randomly chosen nets from the layout in a design file. Figure 8.9 shows a comparison of the results for delays obtained for some small nets that only use single and double line resources.

We represented the critical path of the datapath of the shuffle exchanger and the traffic light controller designs in a pnf file and compared the results with those obtained from the Xilinx tools. These are shown in figures 8.10 and 8.11.

8.5 Summary

In this chapter we presented a methodology to model the delays in a Xilinx 4000e series device. The model describes the architecture of the device at a lower level of abstraction. We laterally propagate the delay values through ports. We presented a method to model feedback paths due to flip flops such that there is no cyclic dependency in the evaluation rules.

The delay estimator is a static system and all attributes are evaluated once. In the following chapter we present a model that uses dynamic rules and attributes.
Figure 8.9: Small Nets

Figure 8.10: Shuffle Exchanger: Critical Path
Figure 8.11: TLC: Critical Path
Chapter 9

Striped FPGA Estimator

This chapter deals with an estimator that models the performance of designs on a striped [11] reconfigurable processor. Striped FPGAs have a unique architecture that exploit hardware virtualization through pipeline reconfiguration [30]. We model the architecture and emulate the hardware virtualization to estimate the execution time, throughput, and the configuration schedule for designs mapped to the reconfigurable processor.

Typical striped architectures consist of a configuration controller that generates the configuration schedule. This is described in section 9.1. This is followed by a description of the framework of the model. We then describe the model development methodology. In section 9.5 we present the results from the estimator for different instances of the striped FPGA family of devices. The CMU PipeRench [11] is an example of a striped FPGA device.

The configurations are temporal in nature and are modeled in PDL+ using dynamic attributes. The evaluation of dynamic attributes is also illustrated through this case study. The dynamic attributes are evaluated in cycles and the number of cycles may be controlled by the designer. In this chapter we present one such method of controlling the number of evaluation cycles. We now describe the reconfigurable processor architecture.

9.1 Striped FPGA Architecture

Figure 9.1 shows an overview of the striped processor architecture. This architecture description is based on the piperench architecture model. The FPGA fabric is divided into many identical physical blocks. These blocks are known as physical stripes and is the basic unit of reconfiguration in the architecture. Each of these physical stripes represents a stage in the pipeline. A compiler for the device, breaks the entire design into pipeline stages such that each stage can be implemented on the physical stripe. The data controller stores and loads the appropriate data from the memory. The configuration words for the entire design is stored in the configuration memory. The configuration words in the memory are referred to as virtual stripes. There is a wide data-bus between the configuration memory and the FPGA fabric so that one memory read configures one pipeline stage at a time. If the number of virtual stripes
are greater than the physical stripes available on the board, there is a need for *hardware virtualization*.

**Figure 9.1: Striped FPGA Architecture**

The configuration controller schedules the configurations to be read from the memory and loaded onto the physical fabric. Typical applications targeted to reconfigurable architectures require the same operation to be performed on large amounts of data. The schedule depends on the number of physical and virtual stripes, and the amount of data to be processed. If the number of virtual stripes are lesser than the physical stripes, all virtual stripes are *alive* in the fabric till all input data is processed. If the virtual stripes are greater than the physical stripes, there is need for *virtualization*. The physical stripes are either *active*, or being *configured*. Consider a design that has four pipeline stages (or virtual stripes) to be implemented on a reconfigurable fabric of 3 physical stripes and the amount of data to be processed is 4. Figure 9.2 traces the movement the virtual stripes to the fabric. At any given time, one stripe is always being configured (C). The stripe that has stayed the longest on the fabric is swapped out of the fabric and a new stripe is loaded onto the fabric. This process continues till the entire data has been processed. The scheduler controls the movement of the virtual stripes from the configuration memory to the fabric. The data movement is controlled by the data controller. The estimator models the movement of stripes and predicts the throughput and execution time for the designs for a family of pipeline reconfigurable FPGAs. The devices in a family of pipeline reconfigurable FPGAs have the *same* basic physical stripe but differ in the *number* of stripes.

**Figure 9.2: Hardware Virtualization**

107
## 9.2 Estimator Framework

The PDL+ models of the physical and virtual stripes and the evaluator and a specific instance of the family of pipeline reconfigurable FPGAs are input to the PDL+ compiler. The compiled model interacts with the ARC evaluator (through the shell) and evaluates the execution time, throughput rate, and the configuration schedule for the design. Figure 9.3 shows the framework of the estimator.

![Estimator Framework Diagram](image)

We now describe the components and attributes of the estimator model.

## 9.3 The Striped FPGA Estimator

The estimator is composed of three modules. These are the *virtual stripe*, the *physical stripe*, and the *eval* modules.

The module declarations for the virtual and physical stripes are shown in figure 9.4.

```plaintext
module virtualstripe
attributes
  primitive vid : int;
end module;

module physicalstripe
attributes
  primitive pid : int;
  dynamic vid : int;
  dynamic state : stripestate := idle;
end module;
```

![Physical and Virtual Stripe Modules](image)

Each *virtualstripe* module has a unique id, *vid* attribute. This attribute is used to bind different virtualstripes to the
physicalstripes. The module `physicalstripe` has a unique `pid`. The `vid` attribute refers to the virtual stripe that is loaded onto the physical stripe. This is a `dynamic` attribute because the virtual stripe loaded on the physical stripe changes over time. The `state` attribute is defined to be of any of the following types. A physical stripe at any given time is either active, being configured, or is idle. The `stripestate` type is defined to be one of these three states.

```plaintext
type
    stripestate : enum{ active, configure, idle };
end type;
```

The evaluation rules for the `vid`, and `state` attributes are specified in the `evaluate` module. The `eval` module is similar to the `evaluate` module described in the partition estimator 7. The physical and virtual stripe modules interact with each other through the `eval` module. The module and its attributes are shown in figure 9.5. The `numdata` attribute is the number of data elements that have to be processed by the design. All dynamic attributes have to be assigned some initial values. These values are used in the first evaluation cycle. The attribute `done` is used to control the number of evaluations. The `dataprocessed` attribute keeps track of the current data element being processed. The evaluation rules for these attributes are described in the future subsections.

```plaintext
module eval
modules
    ps{} : physicalstripe;
    vs{} : virtualstripe;
attributes
    primitive numdata : int;
    dynamic vstripeonboard : int :=0;
    dynamic dataprocessed : int :=0;
    dynamic done : boolean := false;
end
```

Figure 9.5: Eval Module

Every cycle, a new virtual stripe is configured on the board and `vstripeonboard` attribute is a counter that keeps track of the number of times a new configuration is loaded on to the board. New configurations need to be loaded on the board till all data is processed through all the virtual stripes. The number of physical and virtual stripes are used to determine when a particular configuration needs to be loaded.

When a dynamic attribute evaluation rule depends on itself, the key word `curr` is used to refer to the value in the previous iteration. Therefore, it does not lead to a cycle in the evaluation.

Initially all physical stripes are assigned to be in the `idle` state (default value for the attribute). The physical stripe that has to be configured in a given cycle is evaluated by the function `mod_int` shown in figure 9.6.

For example, if there are 4 virtual stripes and 2 physical stripes. If the first virtual stripe is to be loaded in cycle 1. `vstripeonboard` is equal to 1 and `pes` is equal to 2. The function `mod_int` returns the value 1, and the physical stripe
function mod\_int(dividend, divisor : int) : int := 0

begin
    temp : int;
    temp := dividend;
    while (temp > divisor)
        begin
            temp := temp - divisor;
        end;
    if (dividend <= divisor) then temp := dividend; endif;
    return temp;
end;

Figure 9.6: **Function mod\_int**

whose id is 1 is configured. At a time, *only one* of the physical stripes gets configured and the rest continue to remain either active or idle.

The vstripe\_onboard attribute is incremented by 1 for every cycle and corresponds to the *total execution time* for all data through all stripes.

The vid associated with a physical stripe changes only for the stripe being configured. The rest of the stripes continue to hold the same virtual stripe of the previous cycle. The virtual stripe that is being loaded onto the board is obtained by *mod\_int* function.

The rules of the three attributes are shown in figure 9.7. The state attribute depends on the vstripe\_onboard attribute, and the vid attribute depends on the vstripe\_onboard and state attributes. Hence in every evaluation cycle, the vstripe\_onboard is evaluated before the ps state attributes after which the ps vid attributes are evaluated.

vstripe\_onboard = curr vstripe\_onboard + 1; – – total time

ps[ x ]\{state = if (x’pid == mod\_int(vstripe\_onboard, #ps))
    then configure else if (curr x’state != idle) then active else idle endif endif;

ps[ x ]’vid = if (x’pid == mod\_int(vstripe\_onboard, #ps)) then mod\_int(vstripe\_onboard, #vs)
    else curr x’vid endif;

Figure 9.7: **Evaluation Rules**

The dataprocessed attribute traces the movement of data through the stripe fabric and keeps track of the current data that is being processed.

The boolean attribute done is used to control the number of evaluations. If the last virtual stripe executes on the board and all data is processed, we stop evaluation. We iterate through all physical stripes and set the value of done to true if the condition is satisfied. Figure 9.8 shows the PDL+ implementation of these evaluation rules.
dataprocessed =
begin
k : int :=0;
temp : int :=0;
foreach p : physicalstripe in ps
{
    if (p'vid == 1 and p'state == active and curr dataprocessed < numdata)
        then k:= curr dataprocessed+1;
    endif;
}
if (curr dataprocessed < k ) then temp := k; else temp := curr dataprocessed;
endif;
return temp;
end;
done =
begin
result : boolean :=false;
foreach p : physicalstripe in ps
{
    if (p'vid == #vs and p'state == active and dataprocessed == numdata)
        then result := true;
    endif;
}
return result;
end;

Figure 9.8: Evaluation Rules
cell module physicalstripe = p_cell(id:int=-1)
  attributes
  pid = id;
end module;

cell module virtualstripe = v_cell(id:int=-1)
  attributes
  vid = id;
end module;

cell module eval = eval_cell(pstripes:int=0,vstripes:int=0,data:int=2)
composition
  foreach i in 1 to pstripes {
    p[i] : p_cell(id=i);
    ps += p[i];
  }
  foreach i in 1 to vstripes {
    v[i] : v_cell(id=i);
    vs += v[i];
  }
  attributes
  numdata = data;
end module;

eval1 : eval_cell(pstripes=2,vstripes=4,data=5);

Figure 9.9: Design File: pipe.hnf

In the next section we show the execution of the model through the shell.

9.4 Execution of the Model

A sample design file (hnf) is shown in figure 9.9. The architecture consists of 2 physical stripe and we trace the execution of the model for a design of 4 virtual stripes and for 5 data elements. The hierarchical file format is useful in this case because we can instantiate different members of the FPGA family, or application, or number of data elements, through the arguments. changing the number of physical and virtual stripes.

The steps required to execute the model are shown below:

- load program evaluator.pdl
- load design pipe.hnf
- set control eval1’done !=false
- evaluate

The set control command is used to control the number of evaluations. Evaluation proceeds till the done attribute
is not false. The output is shown below. The \texttt{vstripeonboard} shows the total execution time for the design on the architecture. The \texttt{state} attribute shows the state of the physical stripe at each evaluation cycle. At a given time one stripe is always being configured as shown below.

\begin{verbatim}
eval1’numdata = 5;
eval1’vstripeonboard = 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19, 20,21;
eval1’dataprocessed = 0,0,1,1,2,2,2,3,3,3,4,4,4,4,4,5,5,5,5,5,5,5,5;
eval1’done = false,false,false,false,false,false,false,false,false, false,false,false,false,false,false,false,false,false,false,false,
false,true;
eval1$p[1]’pid = 1;
eval1$p[1]’vid = 0,1,1,3,1,1,3,3,1,1,3,3,1,1,3,3,1,1,3,3,1;
eval1$p[1]’state = idle,configure,active,configure,active,configure, active,configure,active,configure,active,configure,active,configure, active,configure,active,configure,active,configure,active,configure;
eval1$p[2]’pid = 2;
eval1$p[2]’vid = 0,0,2,2,4,4,2,2,4,4,2,2,4,4,2,2,4,4,2,2,4,4; 
eval1$p[2]’state = idle,idle,configure,active,configure,active,configure, active,configure,active,configure,active,configure,active,configure, active,configure,active,configure,active,configure,active,configure, active, configure,active,configure,active,configure,active,configure,active;
eval1$v[1]’vid = 1;
eval1$v[2]’vid = 2;
eval1$v[3]’vid = 3;
eval1$v[4]’vid = 4;
\end{verbatim}

In the next section we present the results obtained from executing the model for different cases.

\section*{9.5 Results}

The trends for the execution times for different architectures (physical stripes), applications (virtual stripes), and number of input data to be processed is presented in this section.

Design files with different number of physical stripes were instantiated and the trend of execution time versus number of physical stripes for an application with 100 virtual stripes and 120 input data is shown in figure 9.10.

The trend of execution time over number of virtual stripes on a given architecture is shown in figure 9.11. Here the number of physical stripes is 20 and 200 data elements are to be processed.
9.6 Summary

In this chapter we presented the architecture of a striped FPGA and developed an estimator model that simulates the mapping of an application on the architecture. The throughput, execution time, and configuration schedule are predicted by the estimator.

The use of \textit{dynamic} attributes to specify performance of a system was demonstrated. The evaluation semantics of dynamic attributes was also described. The trends for the performance for different applications and architectures were also studied.

The following chapter describes a \textit{power estimator} that predicts the power dissipated on a Xilinx 4000 series FPGA. The power model is another example of a dynamic system.
Figure 9.11: Execution Time vs Virtual Stripes

- No. of Physical Stripes = 20
- Data to be processed = 200
Chapter 10

FPGA Power Estimator

This chapter deals with the development of an estimator for the power consumption of a Xilinx 4000 series device. The model predicts the total energy and power consumed by a design on the FPGA device. A design netlist that has been mapped to the hardware resources on the FPGA device is input to the estimator. We predict the overall power dissipated by the design for a sequence of inputs to the design.

The estimation methodology is presented in section 10.2 and is an adaptation of the method described in Kusse’s thesis [12]. We describe the PDL+ model of the power estimator and compare results obtained from the estimator with those of Kusse’s model in section 10.4.

Power estimation requires the knowledge of previous state of the system and is modeled using dynamic attributes. The model is hierarchical and uses bottom-up propagation of attributes. We now present the estimation methodology.

10.1 Power Estimation

The power consumed by a design on a CMOS device is the sum of the static, dynamic, and leakage power. The XC4003A device is fabricated on a 0.6 μm, 2-layer metal process. The power model estimates the power dissipated for designs mapped onto this device. The dynamic power constitutes more than 90% of the total power and so we model the dynamic power consumed. Consider a design shown in figure 10.1.

The total power consumed by the design is the sum of the power consumed at each node. The power per node is a function of the supply voltage, frequency, and activity level of the node. The supply voltage and frequencies are a constant for a design and we estimate the activity level of the nodes for a given set of inputs.

\[
\text{The power/node} = Vd^2 \times C \times Activity\ Level \times Frequency \\
= Vd^2 \times C \times Effective\ Frequency \\
= Vd^2 \times Total\ Switched\ Capacitance\ at\ the\ node
\]
Total Power = \( \sum \text{power/node} \)

The capacitance values for different components of the XC4003A -PC84C device is shown in table 10.1. These values were obtained from Kusse’s thesis [12].

<table>
<thead>
<tr>
<th>Component</th>
<th>Estimated cap(pF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLB FG</td>
<td>1.10</td>
</tr>
<tr>
<td>I/O input</td>
<td>5.6</td>
</tr>
<tr>
<td>CLB input</td>
<td>2.4</td>
</tr>
<tr>
<td>CLB output</td>
<td>1.64</td>
</tr>
<tr>
<td>10 CLB longline</td>
<td>4.4</td>
</tr>
<tr>
<td>5 CLB longline</td>
<td>2.2</td>
</tr>
<tr>
<td>Double line</td>
<td>5.35</td>
</tr>
<tr>
<td>Single line</td>
<td>4.4</td>
</tr>
<tr>
<td>Clock</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Table 10.1: **Capacitance Values of XC4003A Device**

We use these values in our power estimation methodology. The power dissipated in a design depends on the exact design, input vectors, and the way in which these capacitance values are embedded in their estimation method. We do not have sufficient information about Kusse’s methodology to implement the same. We however use the capacitance values in table 10.1 in our model.
10.2 Estimation Methodology

We view the FPGA architecture as shown in figure 10.2. We model the I/O capacitance, the output capacitance of a CLB as a lumped capacitance, the net capacitance, and the clock capacitance.

The capacitance at a node contributes to the power dissipated at the node in a given cycle, if the node switches its values. This is referred to as *switched capacitance*. Using these capacitance values, we model the following:

- Functionality of the CLB and determine the switched capacitance at each CLB output node
- Switched capacitance at each primary input (depends on input vectors)
- Switched capacitance at the clock line
- Energy of the design = Total switched capacitance \( \times Vd\,d\,t^2 \)
- Power = Energy\*Frequency

10.3 The Power Model

Figure 10.3 shows the design components of the power model. These are the *ioprt* ports, *connect* carriers, *gate* modules (to specify the functionality of the CLBs), *clb* modules, and the *fpga* module.

10.3.1 Port: ioprt

In order to determine the *activity* of a node, we represent the value of a node by the boolean *value* attribute. Since this value may change at every cycle, it is declared dynamic. Only those nodes that *switch* during a cycle contribute to the dynamic power dissipated. The boolean attribute *event* compares the *value* between two consecutive cycles.
port ioport
end port;

carrier connect
ports
input : ioport;
outputs[] : ioport;
end carrier;

module gate
ports
inputs[] : ioport;
outputs[] : ioport;
end module;

module clb
ports
inputs[] : ioport;
outputs[] : ioport;
modules
gates[] : gate;
end module;

module fpga
ports
inputs[] : ioport;
outputs[] : ioport;
carriers
connects[] : connect;
modules
clbs[] : clb;
end module;

Figure 10.3: Design Components of the Power Model
#include "functions.pdl"

type
GateType: enum {and_gate, or_gate, not_gate, xor_gate, xnor_gate, nand_gate, nor_gate, ff, other_gate};
WireType: enum {single, double, h5long, h10long, v5long, v10long};
end type;

port ioport
attributes
  clbcap: real; -- for ioports connecting to i/o of clbs, this is the
  -- interface cap due to the clb.

dynamic value: boolean := false;
load capacitance: real resolve sum_real;
total capacitance: real;
dynamic event: boolean := false;
rules
  total capacitance = load capacitance + clbcap;
  load capacitance = 0.0;
  
  event = if (value == true) and (curr value == false)
    then true
    else false
  endif;

end port:

Figure 10.4: Port ioport

and is set to true if the value in the previous and present cycle are different. Figure 10.4 shows the port ioport and its attributes.

The capacitance at each node is required to calculate the switched capacitance at the node. The total capacitance of a port is due to the nets and the clbs. This is represented as the sum of the load and clb capacitances at the port. The clbcap attribute represents the capacitance due to the clbs and the load capacitance attribute models the capacitance due to the nets. The evaluator combines all ports that are connected in the design file to one logical port. If each of the connected ports have different evaluation rules for the same attribute, there is a need for resolution functions. We illustrate the use of resolution functions through the load capacitance attribute. The sum_real function is used to resolve the load capacitance attribute of a port. We describe the working of the resolution function in the following two subsections.

10.3.2 Carrier : connect

The carrier is used to model the interconnect capacitances. The capacitance of a wire depends on the segments that constitute the wire. These segments are supplied to the model through the attribute segments. The input load
carrier connect

ports
  input: ioport;
  outputs[]: ioport;
attributes
  primitive segments[]: WireType;
  capacitance: real;
rules
  capacitance = sum_real(foreach s:WireType in segments { wire_cap_table(s) });
  input'load capacitance = capacitance +
    sum_real(foreach output:ioport in outputs
      { output'clbcap } );
  outputs[]’value = input’value;
end carrier;

Figure 10.5: Carrier connect

capacitance is the sum of the wire capacitance and the load capacitances at the outputs. This incorporates the loading due to fanout. Figure 10.5 shows the connect carrier.

The carrier is used to connect two clb modules in the design file. The output port of a clb is the input port for a carrier. The evaluation rule for the load capacitance is applied to the input port of every wire. Since the clb output port and the wire input are merged into one logical port, the evaluation rule for the clb output port is also applied to the same logical port. Since there are multiple drivers for the same attribute, the resolution function is executed. Here the sum_real function is used.

10.3.3 Module : gate

The gate modules represent the functionality of the clbs. The gate_function function returns the output value for the given inputs and gate functionality.

module gate

ports
  inputs[]: ioport;
  outputs[]: ioport;
attributes
  primitive gate_type: GateType;
  dynamic input_values[]: boolean := [];
  dynamic value: boolean := false;
rules
  input_values = foreach x:ioport in inputs { x'value };
value = gate_function(gate_type, input_values);
outputs[]'value = value;
end module;

A clb consists of a set of gates and is modeled as shown in the following subsection.

**10.3.4 Module : clb**

The input and output capacitances of the clbs are obtained from the table 10.1. The clbcap attribute of the input and output ports are set to the input and output capacitances of the clb. The load capacitance due to the clb is set to zero, since this attribute depends on the net capacitances. The sumreal function resolves the load capacitance attribute.

We calculate the switched capacitance at each output of the clb. The switched capacitance in current cycle is the sum of the capacitances of the output ports that switch in a given cycle. Using this we calculate the energy in current cycle. The total switched capacitance is the cumulative sum of the previous cycles and the current cycle. The clb module and its attribute evaluation rules are shown in figure 10.6.

**10.3.5 Module: fpga**

The top-level module in the power estimator is the fpga module. The fpga module, its attributes, and some evaluation rules are shown in figure 10.7. The event count, and number of nodes attributes are used to calculate the percentage nodes switched attribute. This gives us an idea about the activity level of the nodes in the circuit, for a given set of inputs.

The total switched capacitance in a given cycle is the capacitance due to the input ports, clb ports, and the clock. The clb module computes the switched capacitance at the clb ports. There are two transitions on the clock per cycle. The clock energy is modeled as shown in figure 10.7. The primary input switched capacitance is evaluated as shown below.

---

PI power is consumed by the sum of I/O pad cap and the
---

input cap offered by the clb to which I/O is connected.

\[ PI_{\text{switched capacitance in current cycle}} = \sum_{\text{foreach p:ioport in inputs}} \begin{cases} p'\text{event} & (p'\text{total capacitance} + IO_{\text{capacitance}}) \\ 0.0 & \text{endif} \end{cases} \]

\[ PI_{\text{energy in current cycle}} = (\text{voltage}^{*2.0} \times PI_{\text{switched capacitance in current cycle}}); \]
module clb
ports
inputs{}: ioport;
outputs{}: ioport;
modules
ports
inputs{}: ioport;
outputs{}: ioport;
modules
modules
gates{}: gate;
attributes
input_cap: real;
output_cap: real;
voltage: real;
number_of_nodes: int;
dynamic
event_count: int := 0;
dynamic
switched_capacitance_in_current_cycle: real := 0;
dynamic
total_switched_capacitance: real := 0.0;
dynamic
energy_in_current_cycle: real := 0.0;
dynamic
total_energy: real := 0.0;
rules
voltage = 5.0;
input_cap = 2.4e-12;  -- input interface cap;
output_cap = 1.64e-12 + 3*1.10e-12;  -- output interface cap + 3 fungen
inputs{}’clbcap = input_cap;
outputs{}’clbcap = output_cap;
outputs{}’load_capacitance = 0.0;

event_count =
sum_int(foreach p:ioport in outputs
{|if p’event then 1
else 0 endif|});

number_of_nodes = #outputs;

switched_capacitance_in_current_cycle =
sum_real(foreach p:ioport in outputs
{|if p’event then p’total_capacitance
else 0.0 endif|});

energy_in_current_cycle = (voltage ** 2.0 * switched_capacitance_in_current_cycle);
total_switched_capacitance = total_switched_capacitance + switched_capacitance_in_current_cycle;

total_energy = (voltage**2.0 * total_switched_capacitance);
end module;

Figure 10.6: Module clb
module fpga
modules
clbs{} : clb;
carriers
connects{} : connect;
ports
inputs{} : ioprt;
outputs{} : ioprt;
attributes
primitive frequency: real;
voltage: real;
number_of_nodes: int;
dynamic event_count: int := 0;
dynamic percentage_nodes_switched: int := 0;
dynamic cycles: int := 0;
dynamic PI_switched_capacitance_in_current_cycle: real := 0;
dynamic PI_energy_in_current_cycle: real := 0;
dynamic PI_total_energy: real := 0;
dynamic energy_in_current_cycle: real := 0.0;
dynamic average_energy_per_cycle: real := 0.0;
dynamic power: real := 0.0;
rules
voltage = 5.0;
let IO_capacitance: real = 5.6e-12; -- I/O input path capacitance.
let clock_cap: real = 1.5e-12;
let clock_energy_per_cycle: real = voltage**2*clock_cap*2; -- two clock events per cycle
number_of_nodes = sum_int(foreach c: clb in clbs {c'number_of_nodes}) + #inputs;
event_count = sum_int(foreach c: clb in clbs {c'event_count}) +
    sum_int(foreach i: ioprt in inputs {if i'event then 1 else 0 endif});
percentage_nodes_switched = (event_count/number_of_nodes)*100;
cycles = cycles + 1;

Figure 10.7: Module fpga
\[ \text{PI}_{\text{total\_energy}} = \text{PI}_{\text{total\_energy}} + \text{PI}_{\text{energy\_in\_current\_cycle}}; \]

The energy in the current cycle is the sum of energies due to clbs, primary inputs and clocks. The *power* is represented as the energy per clock cycle in milliWatts.

\[
\text{energy\_in\_current\_cycle} = (\text{sum}\_\text{real}(\text{foreach } c : \text{clb in clbs } \{ c'\text{\_energy\_in\_current\_cycle} \}) + \\
\text{PI}_{\text{energy\_in\_current\_cycle}} + \\
\text{clock\_energy\_per\_cycle}) \times 1e9; \quad \text{-- } nJ; \; \text{two clock events per cycle}
\]

\[
\text{total\_energy} = (\text{sum}\_\text{real}(\text{foreach } c : \text{clb in clbs } \{ c'\text{\_total\_energy} \}) \\
+ \text{PI}_{\text{total\_energy}} + \text{clock\_energy\_per\_cycle}\times \text{cycles}) \times 1e9; \quad \text{-- } nJ
\]

\[
\text{average\_energy\_per\_cycle} = \text{total\_energy}/\text{cycles}; \quad \text{-- } nJ
\]

\[
\text{power} = \text{average\_energy\_per\_cycle} \times \text{frequency} \times 1e-6; \quad \text{-- this is based on power = energy per clock cycle in mW;}
\]

### 10.3.6 Functions Used

We show the definitions of the functions used in the attribute evaluation rules of the previous sections in this subsection. The function *gate\_function* is used to determine the value at the output of a gate of a given type. The function has three different cases based on the number of inputs on the gates. The *values* parameter determines the number of inputs to the gate. If the number of inputs is greater than 1, we perform a bitwise operation on the inputs to evaluate the corresponding output value.

**function gate\_function(gate\_type: \GateType; values[]: \Boolean): \Boolean := false**

**begin**

\[
\text{temp: } \Boolean := \text{true};
\]

if \#(values) == 0 then return false; endif;

if gate\_type == \text{not}\_gate then return \(\text{not} \text{values}[1] \); endif;

if gate\_type == \text{ff} then return values[1]; endif;

if \#(values) == 1

then

\[
\text{case gate\_type of}
\]
and_gate: return values[1];
or_gate: return values[1];
xor_gate: return false;
xnor_gate: return true;
nand_gate: return (not values[1]);
nor_gate: return (not values[1]);
others: return values[1];
end case;
endif;

if #(values) > 1 then
begin
temp := values[1];
foreach i:int in 2 to #values
{
    case gate_type of
        and_gate: temp := temp and values[i];
or_gate: temp := temp or values[i];
xor_gate: temp := temp xor values[i];
xnor_gate: temp := temp xnor values[i];
nand_gate: temp := temp nand values[i];
nor_gate: temp := temp nor values[i];
others: return values[1];
    end case;
}
return temp;
end;
endif;
end;

function wire_capable(s: WireType): real := 0.0 /* page. 28, Kusse MS Thesis */
begin
c : real := 0.0;
c :=
    case s of
        single: 4.44e-12;
double: 5.35e-12;
h5long: 1.1e-12;
v5long: 2.2e-12;
The `wire_cap_table` function returns the corresponding capacitance values for segments of a given type. These capacitance values are the same as shown in table 10.1.

### 10.4 Results

We present a comparison between the results obtained from Kusse’s thesis and our model. We represented single bit implementations of the adder, comparator, and shifter circuits within a CLB. We then cascaded these clbs to obtain results for the 4 and 8 bit implementation of these designs. The input vectors were provided such that the activity level of the nodes was around 50% for every cycle. We also assigned some nominal interconnect resources to connect the clbs.

We however do not have sufficient information about the exact designs that were used by Kusse et al. We have used the following designs as inputs to the estimator. A ripple carry implementation for the adder, a serial shifter, and a bit-wise comparator circuit. Figures 10.8, 10.9, and 10.10 show the comparison of the results obtained from our estimator and Kusse’s power estimation tool.

The design files and the input values of each of the models is included in the accompanying diskette.

### 10.5 Summary

We presented a methodology to model the power dissipated in a Xilinx 4000 series FPGA. We used dynamic attributes to represent the temporal functionality of the nodes in the circuit. We illustrated the evaluation cycles in dynamic attributes. We also discussed the use of resolution functions during attribute evaluation.

In the next chapter we present the use of qdynamic variables and their evaluation semantics through a model of a hypothetical FPGA.
Adders

Figure 10.8: **Power: adder**

Comparators

Figure 10.9: **Power: comparator**
Figure 10.10: Power: shifter
Chapter 11

Reconfiguration Modeling

This chapter presents an RC modeling methodology for different FPGA architectures. We illustrate the use of \textit{qdynamic} attributes and evaluation rules. PDL+ can be used to specify abstract models of a system’s functionality and performance [31]. Section 11.1 discusses the modeling of various elements of an RC using PDL+. We develop a model of the FPGA architecture from these elements. We describe a \textit{scalable} model of an RC co-processor that contains the FPGA model. This chapter illustrates various features of PDL+ and the ARC system in developing \textit{abstract} RC models. The model described in this chapter was a part of a paper authored by Vemuri, and Walrath in SPIE’98 [31].

11.1 Modeling Reconfiguration

A reconfigurable element has several distinct \textit{modes} or \textit{configurations} of operation. We consider four types of elements in RCs: computing, interconnect, memory and I/O. An abstract functional model of an element identifies the element’s modes of operation, its mode transition – reconfiguration – rules and the element’s behavior in each mode. An abstract performance model of an element identifies its performance parameters and how they should be computed in each mode. In PDL+, both functional and performance models can be written using attributes and attribute evaluation rules.

Consider the look-up table based configurable logic block in Figure 11.1. The look-up table consists of several bits each of which can be configured into one of the two possible modes: 0 or 1. In PDL+, this is modeled as follows:

```pdl+
module lut_bit: boolean := false
end module;
```

In this model, module \texttt{lut\_bit} is specified as having a Boolean type configuration mode. Its initial mode is specified
Figure 11.1: **Configurable Logic Block**

to be false. Any discrete type can be used to specify the configuration modes; each element of the type then identifies a unique configuration. Any component in PDL+ can have a configuration mode associated to it.

An I/O element can be declared in PDL+ using the port declaration construct as shown below.

```pdl
port iport
end port;
```

Carrier declaration construct in PDL+ is used to introduce interconnect objects:

```pdl
carrier long_wire
ports
  inputs{}: ioprot;
  outputs{}: ioprot;
end carrier;
```

This declares long_wire interconnect objects as having a set of input ports and a set of output ports, both of type ioprot declared earlier.

Memory elements are no different from computing elements. A memory element is viewed as having various configurations corresponding to its content. Consider the following example:

```pdl
type
  word[]: boolean;
  table[]: word;
end type;
```

```pdl
module memory: table
```

Spinell
type
  mode_list[]: boolean;
clb_mode:
  record
    fg_mode : mode_list;
    mux_mode : boolean;
  end record;
end type;

module lut_bit : boolean := 0
attributes
  primitive id: int;
end module;

module function_generator : mode_list := []
modules
  lut{} : lut_bit;
ports
  inputs{} : ioport;
  output : ioport;
rules
  lut{x}'trigger = trigger;
  lut{x}'mode = mode[x'id];
end module;

module clb : clb_mode := <[], 0>>
modules
  fg : function_generator;
  ff : flip_flop;
  mux: multiplexer;
ports
  inputs{} : ioport;
  output : ioport;
attributes
  primitive id: int;
rules
  fg'trigger = trigger;
  mux'trigger = trigger;
  fg'mode = mode.fg_mode;
  mux'mode = mode.mux_mode;
end module;

Figure 11.2: Reconfigurable Model of the Logic Block

ports
  address, data: ioport;
  enable, rw: ioport;
end module;

In this example, a word is declared as a list of Boolean objects and a table as a list of words. Then, the memory module is declared; its modes are denoted by the table type. That is, each distinct value of table denotes a distinct mode of the memory object.

11.1.1 Modeling Using Modes

Configuration of an element is a special attribute attached to the element. During model evaluation, the configuration of an element can be changed by executing a rule that changes the current configuration to a new configuration. Figure 11.2 shows the model of the reconfigurable logic block shown in Figure 11.1:

The clb component in this model has a record type configuration with two fields. The first field is a list which defines
the configurations for each of the look-up table bits and the second field defines the configuration of the multiplexor. The initial values of these two fields are declared to be the null list and '0’ (false) respectively. Each configurable element in PDL+ has a built-in Boolean attribute called trigger which must be set to true in order for the element to be reconfigured. The new configuration itself is determined by an assignment to the mode which is another built-in attribute of every reconfigurable element. A set of attribute evaluation rules are attached to each element to determine how these and other attributes change. The first two rules in clb module specify that the triggers of its children modules fg and mux assume the same value as the trigger of the parent clb. Where clb is the top-level module, its trigger must be set to true or false by the environment during evaluation. The new modes of fg and mux are determined by the next two evaluation rules in clb which propagate the appropriate fields in the clb’s mode. The first evaluation rule in the function generator sets the triggers of each lut.bit in it and the next evaluation rule sets their modes by indexing into the clb’s mode list using the integer ids of the lut bits. These two rules are examples of broadcast rules in PDL+ where the attributes of a set of contained objects are set by a single rule in the parent object.

11.2 Performance Model of the FPGA

Figure 11.4 shows a simple delay model of a logic block and a clock-period model of an FPGA design. This model determines the maximum flip-flop to flip-flop delay in a given configuration of the FPGA following the scheme shown in Figure 11.3. In this simple model, the interconnect is non-programmable and its delay is ignored.

In this model, the FPGA contains a set of logic blocks; each block derives its configuration by indexing into the configuration list fpga.mode. A time attribute is attached to each I/O port. In this model, the delay of the function generator module is computed as the product of the size of the look-up table and a delay factor per each look-up table bit which is declared as a primitive attribute.

The flip-flop to flip-flop path delay is computed by setting the time at the output of the flip-flop to zero and propagating it through a chain of function-generators until reaching the input of a flip-flop, accumulating the delay through the function-generators and multiplexors. This is similar to the approach described in the delay estimator in chapter.
type
   fpgamode[]: clb_mode;
end type;

port ioport
attributes
time: real;
end port;

module multiplexer : boolean := 0
ports
   input1, input2, output: ioport;
attributes
   primitive delay: real;
   select: boolean;
rules
   output'time = if mode
      then input1'time + delay
      else input2'time + delay
   endif;
end module;

module flip_flop
ports
   input, output: ioport;
attributes
   primitive delay: real;
   time: real;
rules
   output'time = 0.0;
   time = input'time + delay;
end module;

module function_generator : mode_list := []
modules
   lut{} : lut_bit;
ports
   inputs{} : ioport;
   output : ioport;
attributes
   primitive delay_per_lut_bit: real;
   delay: real;
rules
   lut{x}’trigger = trigger;
   lut{x}’mode = mode[x’id];
   delay = #lut * delay_per_lut_bit;
   output’time = delay +
      max_real(foreach x:ioport in inputs {x’time})
end module;

module clb : clb_mode := <[], 0>
modules
   fg : function_generator;
   ff : flip_flop;
   mux: multiplexer;
ports
   inputs{}: ioport;
   output : ioport;
attributes
   primitive id: int;
   time: real;
rules
   ff’trigger = trigger;
   mux’trigger = trigger;
   fg’mode = mode.fg_mode;
   mux’mode = mode.ff_mode;
   time = ff’time;
end module;

module fpga : fpga_mode := []
modules
   clbs{} : clb;
ports
   inputs{} : ioport;
   outputs{} : ioport;
rules
   clbs{x}’mode = mode[x’id];
   clbs{}’trigger = trigger;
attributes
   primitive id: int;
   time: real;
   qdynamic clock_period: real := 0.0;
rules
   inputs{}’time = 0;
   time = max_real(foreach x:clb in clbs {x’time});
   clock_period = max_real({time, curr clock_period});
end module;

Figure 11.4: Performance Model of FPGA Designs
8. The *time* attribute of the *fpga* is computed to be the maximum of these combinational path delays in the current *fpga* configuration. Clock period is the maximum of these maximum combinational path delays across all the configurations under which the *fpga* is exercised during evaluation.

At the beginning, all static attributes are evaluated and all dynamic and quasi-dynamic attributes are set to their initial values. Then in each cycle of evaluation where no reconfiguration takes place – that is, no element has its trigger set to true – all dynamic attributes are evaluated and their new values replace their old values. If a trigger is set during any cycle then the performance model is reconfigured by evaluating and updating the mode of each module whose trigger is set. All static attributes are evaluated again since primitive attribute data might have changed. All quasi-dynamic attributes are also evaluated and updated. Following this, cycles of dynamic attribute evaluation commence as before. Since mode changes occur only when triggers are set, mode is considered a built-in quasi-dynamic attribute.

### 11.2.1 Evaluation

In summary, static attributes are used to compute and store data that is static for the duration of a configuration and doesn’t depend upon the previous configuration. Dynamic attributes assume streams of values, one per each evaluation cycle, throughout the course of model evaluation as illustrated by the models in the previous two chapters. Quasi-dynamic attributes, like dynamic attributes assume streams of values, but only one per each configuration that the model goes through. This is because quasi-dynamic attributes are reevaluated *only* when reconfiguration takes place. Thus, these attributes are typically used to carry data from one configuration to the next and remain stable through the configuration. Since quasi-dynamic attributes also assume streams of values, it is possible to refer to their current values using the *curr* keyword, while computing a new value for these attributes. Thus, the rule for clock-period in Figure 11.4 declares that the new clock-period is the maximum of the maximum combinational path time in the current configuration and the current value of the clock-period determined over all the previous configurations.

### 11.2.2 Usage Context

As an illustration, consider the performance model shown in Figure 11.4. A compiler may be interested in determining the clock-period of the *fpga* over a sequence of proposed configurations. The configuration data, along with the primitive attribute data, is supplied to the evaluator which then executes the performance model and returns the computed clock-period.
11.3 Reconfigurable Coprocessor Model

We now describe an abstract model of a reconfigurable coprocessor. Figure 11.5 shows a reconfigurable, multi-FPGA based coprocessor communicating with a host to provide a hardware-software coexecution environment. We will develop a simple model of this environment to help determine the latency of a task graph after binding each task to software or hardware in a cosynthesis system. Figure 11.6 shows the PDL+ model of a non-pipelined, cache-free processor. In this model the execution time of a task modeled in turn as a sequence of instructions is determined by first computing the total number of processor cycles required to execute the task.

Figure 11.7 shows the performance model of a multi-FPGA RC board. In this model, the clock speed of the board is computed as the minimum clock speed at which any FPGA on the board can operate. The list of all FPGA configurations defines the coprocessor configuration. FPGA clock-period model was discussed earlier in this paper. It is assumed that the cosynthesis tool synthesizes the task on the coprocessor architecture, determines the FPGA configurations from the task description and invokes this performance model to determine the speed at which the board can operate to execute that task. Note that it is entirely possible to develop more abstract or more concrete models of coprocessor performance depending upon the accuracy and speed requirements of the cosynthesis tool. Also note that, this PDL+ model illustrates how irrelevant details of the board design can be ignored when writing abstract models so that only the necessary aspects can be modeled.

Figure 11.8 shows example tradeoff data examined by a cosynthesis tool by interacting with ARC system evaluating this coprocessor model. Through this plot, the cosynthesis tool is exploring the impact of processor speed grade and scheduling options for a coprocessor bound task named t9 on the overall execution latency of the task graph being synthesized for coexecution.

11.4 Programmable Interconnect

Figure 11.9 shows a programmable switch box and Figure 11.10 shows its performance model. The switch box consists of switches. The switch is ON if its mode is true; OFF otherwise. The switch box has inputs and outputs on all its four sides and contains twelve switches, three per each input. By programming the switches, the left input may be connected to the right, top or bottom outputs. Similarly any input may be connected to any of the outputs except the output on the same side as that input.

In this model, the time attribute is dynamic; a new value for the attribute is computed in each evaluation cycle. Also,
type
opcode: enum [ADDR, ADDM, SUB, MOVR, MOVI, FDIV...];
program[]: opcode;
end type;

module processor : program := []
attributes
  primitive clock_speed: int; -- MHz
total_cycles: int;
execution_time: real; -- ns
rules
total_cycles = sum_int(foreach i:opcode in curr mode {cycles(i)});
execution_time = (1/clock_speed)*total_cycles*1000;
end module;

function cycles(op: opcode):int := 0
begin
  return (case op of
    ADDR: 1;
    ADDM: 2;
    MOVR: 1;
    FDIV: 4;
    ...
    others: 1;
  end case);
end:

Figure 11.6: Performance Model of the Processor

type
rc_mode[]: fpga_mode;
end type;

module reboard : rc_mode := []
modules
fpgas[]: fpga;
rules
fpgas[x] `mode = mode[x'idx];
fpgas[] `trigger = trigger;
attributes
  qdynamic clock_period: real := 0.0;
rules
clock_period = max_real(foreach x:fpgas [x'clock_period]);
end module;

Figure 11.7: Performance Model of the Coprocessor

137
Figure 11.8: Codesign Performance Analysis Data

Figure 11.9: Programmable Switch Box
in this model, time is a resolved attribute. A resolution function, max_real in this case, is used to determine the effective value of an attribute with multiple definitions. If a signal is passing to the output of the switch through multiple paths, the delay should be the maximum of these path delays. Resolution using max_real accomplishes this. When the switch is ON it reproduces its input signal at its output with a delay. By programming the switches in various ways, different permutations of the input are achieved.

Figure 11.11 shows the architecture of programmable interconnect device using the programmable switch boxes and Figure 11.10 shows its scalable PDL+ model. A device with $n \times n$ switch boxes has $2 \times n$ inputs and $2 \times n$ outputs on each side and contains $12 \times n^2$ switches. By configuring the switch boxes, various routes from inputs to outputs can be implemented. The above model can be used by an RC compiler to determine, for example, the worst-case propagation delay through the interconnect device in a given configuration.

### 11.5 Time-Multiplexed FPGAs

Figure 11.12 shows a configurable logic block in a time-multiplexed or context-switchable FPGA [32, 33]. The logic block contains an active look-up table and several passive configurations called planes. During any cycle any passive configuration can loaded into the look-up table or a passive configuration can be updated by external input.
The latter operation can also be done concurrently with the normal operation of the logic block. By selectively switching the configuration in some logic blocks, partial reconfiguration of the FPGA is accomplished. Figure 11.13 shows a PDL+ model of this logic block.

In contrast to the previous examples, this model uses dynamic attributes exclusively. Modes, which are quasi-dynamic attributes, are not used. This is due to the fact that computation, configuration loading and context-switching can all be interleaved in this FPGA. To model all of these in each cycle, all attributes are modeled as dynamic attributes. The PDL+ program includes all attributes necessary to model context-switching. Additional attributes, as illustrated by previous examples, can be introduced to model performance as desired.

### 11.6 Summary

This chapter presented a methodology for abstract performance modeling of a FPGA architectures in PDL+. We illustrated the use of qdynamic attributes and compared the evaluation semantics between static, dynamic, and qdynamic attribute types.

In the following chapter we present some guidelines listed as thumb rules for future model developers in PDL+.
Figure 11.12: Time-Multiplexed Logic Block
type
lut\_mode\[]): boolean;
end type;

port iport
der end port;

module function\_generator
ports
inputs\[]\): iport;
output : iport;
attributes
— active LUT mode
dynamic config: lut\_mode := [];
end module;

module flip\_flop
ports
input, output: iport;
der end module;

module multiplexer
ports
input1, input2, output: iport;
attributes
dynamic select: boolean := 0;
end module;

module clb
modules
fg : function\_generator;
ff : flip\_flop;
mux: multiplexer;
ports
inputs\[]\): iport;
output : iport;
attributes
dynamic planes\[]\): lut\_mode := [];

primitive dynamic load\_mode: boolean := 0;
primitive dynamic target\_plane: int := 0;
primitive dynamic new\_mode: lut\_mode := [];

primitive dynamic switch\_plane: int := 0;
— 0: don’t switch planes; 1: switch planes
rules
planes =
if load\_mode
then insert(curr planes, target\_plane, new\_mode)
else curr planes
endif;

let new\_plane: lut\_mode = if switch\_plane == 0
then []
else planes[switch\_plane]
endif;

let current\_fgconfig: lut\_mode = curr fg’config;
let current\_muxconfig: boolean = curr mux’select;

fg’config = if switch\_plane == 0
then current\_fgconfig
else delete(new\_plane, 1)
endif;

mux’select = if switch\_plane == 0
then current\_muxconfig
else new\_plane[1]
endif;
end module;

Figure 11.13: Model of a Time-Multiplexed Logic Block
Chapter 12

Conclusions

This thesis describes performance modeling methodologies using PDL+ and the ARC environment. The modeling methods were demonstrated with the help of case studies that model performance of systems at different levels of abstraction. The models were documented such that the development procedure may guide future model developers.

The models belong to different levels of abstraction in the design flow. For example, the behavioral estimator and partition estimator, model systems at a higher level of abstraction, while the power estimator and the FPGA delay estimator model systems at the physical level. We presented estimation methodologies for the performance metrics for each of these systems along with the corresponding PDL implementations. The annotated code for each of these models, sample design files are provided in the accompanying diskette.

In the following section we summarize the model development procedure as guidelines that may be used by future model developers.

12.1 Modeling Guidelines

- Model the system as a collection of modules, carriers, and ports.
- Assign attributes to these components.
- Develop evaluation rules for the attributes.
- If the rule for an attribute depends only on synthesized attributes, define the rule within the component. 6.4.1
- If the rule depends on attributes of its parent module, define the rule in the parent component 6.4.2
- If the rule requires attributes from an independent component, create a binding module that instantiates these components, and define the rule in the binding component. 7.4.3
• Define the rules such that they are easily understandable and modifiable.

• Use local attributes to simplify the definitions of complicated rules. 7.5.65.1

• If attributes are propagated hierarchically, use broadcast definitions for the child attributes from within the parent module.

• If the attributes of each child have to be set to a different value, use an id attribute to distinguish the children or use arrays instead of sets for containment. 7.4.2

• If an attribute is to be propagated laterally between modules and carriers, declare a port component with the attribute and assign the attribute from within the modules and carriers. 7.4.2

• If the same logical port may have more than one driver, define appropriate resolution functions for the attribute. 10.3.1

• Define commonly used functions in a separate file and include them wherever necessary.

• If the model interacts with other automation tools, define the necessary interface functions and access the compiled model through the procedural interface else use the shell to interact with the model. 4.6.1

• Use the hierarchical netlist format if very large, regular, or parameterized designs are to be instantiated. 7.5.2

In the following subsection we present some directions for future work.

### 12.2 Future Work

The models developed during the course of this thesis were one of the first few real life models developed in the ARC system. We’ve listed possible suggestions for future work below.

• Identify more systems and model them in ARC.

• Explore more applications that require qdynamic attributes to describe performance.

• Identify applications that would appropriately exploit the newer features like projections that were recently added to the language.

• Most of the computer aided design tools in VLSI use a class of heuristic algorithms that use different cost functions. These cost functions can be modeled as evaluation rules in PDL, as the rules are easy to understand and modify.
Appendix A

Characterized Component Data

Tables A.1, A.2, A.3, A.4, A.5, A.6 show the characterized data for all components in the library. These components are instantiated by the high level synthesis tools in designs used in the future chapters.
<table>
<thead>
<tr>
<th>comp_name</th>
<th>bitwidth</th>
<th>shape_num</th>
<th>clb</th>
<th>3-LUT</th>
<th>4-LUT</th>
<th>FF</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_sub</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>9.181</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>10.612</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>11.182</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>10.921</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>12</td>
<td>3</td>
<td>22.643</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>15</td>
<td>3</td>
<td>29.203</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>15</td>
<td>3</td>
<td>29.469</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>15</td>
<td>3</td>
<td>29.469</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1</td>
<td>22</td>
<td>0</td>
<td>24</td>
<td>7</td>
<td>56.584</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16</td>
<td>0</td>
<td>31</td>
<td>7</td>
<td>69.386</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>16</td>
<td>0</td>
<td>31</td>
<td>7</td>
<td>66.607</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>0</td>
<td>31</td>
<td>7</td>
<td>74.012</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>46</td>
<td>0</td>
<td>4</td>
<td>15</td>
<td>142.747</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>32</td>
<td>0</td>
<td>63</td>
<td>15</td>
<td>142.301</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>32</td>
<td>0</td>
<td>63</td>
<td>15</td>
<td>148.307</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>63</td>
<td>15</td>
<td>147.374</td>
<td></td>
</tr>
<tr>
<td>c_adder</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>7.799</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>10.298</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>9.948</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>9.948</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>12</td>
<td>3</td>
<td>22.726</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>15</td>
<td>3</td>
<td>30.486</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>15</td>
<td>3</td>
<td>16.72</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>15</td>
<td>3</td>
<td>30.78</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1</td>
<td>22</td>
<td>0</td>
<td>24</td>
<td>7</td>
<td>65.536</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16</td>
<td>0</td>
<td>31</td>
<td>7</td>
<td>74.266</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>16</td>
<td>0</td>
<td>31</td>
<td>7</td>
<td>64.301</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>0</td>
<td>31</td>
<td>7</td>
<td>74.266</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>46</td>
<td>0</td>
<td>48</td>
<td>15</td>
<td>114.9030</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>32</td>
<td>0</td>
<td>63</td>
<td>15</td>
<td>129.327</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>32</td>
<td>0</td>
<td>63</td>
<td>15</td>
<td>136.312</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>63</td>
<td>15</td>
<td>142.513</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1</td>
<td>94</td>
<td>0</td>
<td>96</td>
<td>31</td>
<td>206.75</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>64</td>
<td>0</td>
<td>127</td>
<td>31</td>
<td>281.68</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>64</td>
<td>0</td>
<td>127</td>
<td>31</td>
<td>280.855</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>64</td>
<td>0</td>
<td>127</td>
<td>31</td>
<td>319.644</td>
<td></td>
</tr>
</tbody>
</table>

Table A.1: Characterized data - I
<table>
<thead>
<tr>
<th>comp_name</th>
<th>bitwidth</th>
<th>shape_num</th>
<th>clb</th>
<th>3-LUT</th>
<th>4-LUT</th>
<th>FF</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_compar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>13.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>17.048</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>15.639</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>15.651</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>17</td>
<td>2</td>
<td>28.56</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10</td>
<td>0</td>
<td>19</td>
<td>2</td>
<td>39.391</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>10</td>
<td>0</td>
<td>19</td>
<td>2</td>
<td>32.428</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>9</td>
<td>0</td>
<td>18</td>
<td>2</td>
<td>29.485</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>24</td>
<td>0</td>
<td>44</td>
<td>5</td>
<td>50.512</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>23</td>
<td>0</td>
<td>41</td>
<td>5</td>
<td>58.962</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>23</td>
<td>0</td>
<td>41</td>
<td>5</td>
<td>71.255</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>23</td>
<td>0</td>
<td>41</td>
<td>5</td>
<td>57.394</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>52</td>
<td>0</td>
<td>95</td>
<td>5</td>
<td>105.34</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>49</td>
<td>0</td>
<td>92</td>
<td>5</td>
<td>131.761</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>49</td>
<td>0</td>
<td>92</td>
<td>5</td>
<td>155.774</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>49</td>
<td>0</td>
<td>92</td>
<td>5</td>
<td>126.879</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>107</td>
<td>0</td>
<td>191</td>
<td>7</td>
<td>193.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>98</td>
<td>0</td>
<td>191</td>
<td>7</td>
<td>470.66</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>98</td>
<td>0</td>
<td>191</td>
<td>7</td>
<td>394.332</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>98</td>
<td>0</td>
<td>191</td>
<td>7</td>
<td>376.597</td>
<td></td>
</tr>
<tr>
<td>c_mult</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>27</td>
<td>6</td>
<td>22.858</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>15</td>
<td>0</td>
<td>29</td>
<td>6</td>
<td>39.217</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>15</td>
<td>0</td>
<td>29</td>
<td>6</td>
<td>32.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>29</td>
<td>6</td>
<td>30.969</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>63</td>
<td>0</td>
<td>120</td>
<td>14</td>
<td>48.442</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>65</td>
<td>0</td>
<td>115</td>
<td>14</td>
<td>90.569</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>65</td>
<td>0</td>
<td>115</td>
<td>14</td>
<td>112.448</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>65</td>
<td>0</td>
<td>115</td>
<td>14</td>
<td>108.0499</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>255</td>
<td>0</td>
<td>500</td>
<td>40</td>
<td>117.796</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>260</td>
<td>0</td>
<td>481</td>
<td>40</td>
<td>553.359</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>1023</td>
<td>0</td>
<td>2027</td>
<td>304</td>
<td>448.894</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1023</td>
<td>0</td>
<td>2027</td>
<td>304</td>
<td>448.894</td>
<td></td>
</tr>
</tbody>
</table>

Table A.2: Characterized data - II
<table>
<thead>
<tr>
<th>comp_name</th>
<th>bitwidth</th>
<th>shape_num</th>
<th>clb</th>
<th>3-LUT</th>
<th>4-LUT</th>
<th>FF</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_and</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1</td>
<td>64</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td>c_or</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1</td>
<td>64</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Table A.3: Characterized data - III
<table>
<thead>
<tr>
<th>comp_name</th>
<th>bitwidth</th>
<th>shape_num</th>
<th>clb</th>
<th>3-LUT</th>
<th>4-LUT</th>
<th>FF</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_nand</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1</td>
<td>64</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td>c_nor</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1</td>
<td>64</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Table A.4: Characterized data - IV
<table>
<thead>
<tr>
<th>comp_name</th>
<th>bitwidth</th>
<th>shape_num</th>
<th>clb</th>
<th>3-LUT</th>
<th>4-LUT</th>
<th>FF</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_xnor</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1</td>
<td>64</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>c_xor</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1</td>
<td>64</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>2</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Table A.5: Characterized data - V
<table>
<thead>
<tr>
<th>comp_name</th>
<th>bitwidth</th>
<th>shape_num</th>
<th>clb</th>
<th>3-LUT</th>
<th>4-LUT</th>
<th>FF</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_register</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>4</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>4</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>4</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>4</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>16</td>
<td>0</td>
<td>8</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>8</td>
<td>16</td>
<td>0</td>
<td>8</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>8</td>
<td>16</td>
<td>0</td>
<td>8</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>0</td>
<td>8</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>32</td>
<td>0</td>
<td>16</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>16</td>
<td>32</td>
<td>0</td>
<td>16</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>16</td>
<td>32</td>
<td>0</td>
<td>16</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>16</td>
<td>32</td>
<td>0</td>
<td>16</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>32</td>
<td>64</td>
<td>0</td>
<td>32</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>32</td>
<td>64</td>
<td>0</td>
<td>32</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>32</td>
<td>64</td>
<td>0</td>
<td>32</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>32</td>
<td>64</td>
<td>0</td>
<td>32</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1</td>
<td>64</td>
<td>128</td>
<td>0</td>
<td>64</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>64</td>
<td>128</td>
<td>0</td>
<td>64</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>64</td>
<td>128</td>
<td>0</td>
<td>64</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>64</td>
<td>128</td>
<td>0</td>
<td>64</td>
<td>2.5</td>
</tr>
<tr>
<td>c_mux</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>16</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Table A.6: Characterized data - VI
Bibliography


[22] Xilinx, Inc. The XC6200 Datasheet. Xilinx Inc.


