

PHYSICAL SYNTHESIS TOOLKIT FOR AREA AND POWER OPTIMIZATION ON FPGAs

by

Tomasz Sebastian Czajkowski

A thesis submitted in conformity with the requirements
For the degree of Doctor of Philosophy,
Edward S. Rogers Sr. Graduate Department of
Electrical and Computer Engineering
University of Toronto, Toronto, Ontario, Canada

© Copyright by Tomasz Sebastian Czajkowski 2008

Abstract

Physical Synthesis Toolkit for Area and Power Optimization on FPGAs

Tomasz Sebastian Czajkowski

Doctor of Philosophy

Edward S. Rogers Sr. Graduate Department of Electrical and Computer Engineering

University of Toronto, Toronto, Canada, 2008

A Field-Programmable Gate Array (FPGA) is a configurable platform for implementing a variety of logic circuits. It implements a circuit by the means of logic elements, usually Lookup Tables, connected by a programmable routing network. To utilize an FPGA effectively Computer Aided Design (CAD) tools have been developed. These tools implement circuits by using a traditional CAD flow, where the circuit is analyzed, synthesized, technology mapped, and finally placed and routed on the FPGA fabric. This flow, while generally effective, can produce sub-optimal results because once a stage of the flow is completed it is not revisited.

This problem is addressed by an enhanced flow known Physical Synthesis, which consists of a set of iterations of the traditional flow with one key difference: the result of each iteration directly affects the result of the following iteration. An optimization can therefore be evaluated and then adjusted as needed in the following iterations, resulting in an overall better implementation. This CAD flow is challenging to work with because for a given FPGA researchers require access to each stage of the flow in an iterative fashion. This is particularly challenging when targeting modern commercial FPGAs, which are far more complex than a simple Lookup Table and Flip-Flop model generally used by the academic community.

This dissertation describes a unified framework, called the *Physical Synthesis Toolkit* (PST), for research and development of optimizations for modern FPGA devices. PST provides access to modern FPGA devices and CAD tool flow to facilitate research. At the same time the amount of effort required to adapt the framework to a new FPGA device is kept to a minimum.

To demonstrate that PST is an effective research platform, this dissertation describes

optimization and modeling techniques that were implemented inside of it. The optimizations include: an area reduction technique for XOR-based logic circuits implemented on a 4-LUT based FPGA (25.3% area reduction), and a dynamic power reduction technique that reduces glitches in a circuit implemented on an Altera Stratix II FPGA (7% dynamic power reduction). The modeling technique is a novel toggle rate estimation approach based on the XOR-based decomposition, which reduces the estimate error by 37% as compared to the latest release of the Altera Quartus II CAD tool.

Acknowledgments

I would like to thank my thesis supervisor, Professor Stephen Dean Brown, for his guidance and teachings over the years. His extensive knowledge has given me a solid foundation to build my work on, while his keen eye for detail and unwavering dedication to the scientific process has improved the quality of my research. On a personal level, I am grateful for the patience and dedication you have shown in the effort to further my education. You have gone beyond the call of duty to help me become a better researcher and a better teacher. I consider you to be my mentor.

To Professor Zvonko George Vranesic I extend my gratitude for the guidance he offered during the course of my studies. As a member of my advisory committee he has always provided sound advice and shared his experience and wisdom both with myself and other members of our research group. I am grateful for his contribution to my education.

I also take this opportunity to thank Professor Jianwen Zhu for his contribution as a member of my advisory committee. His feedback was invaluable in improving the quality of my research.

To my family, Dr. Grzegorz Czajkowski, Tatiana Czajkowska, and Przemysław Czajkowski, I extend my most heartfelt gratitude for their support throughout my studies.

I would also like to thank my friends and colleagues from the University of Toronto. In particular, I would like to acknowledge: Andrew Ling, Franjo Plavec, Mark Bourgeault, and Henry Jo.

Finally, I would like to thank my academic supervisor, the University of Toronto and Altera Corporation for funding this research.

Table of Contents

Abstract	iii
Acknowledgments	v
Table of Contents	vii
List of Figures	xi
List of Tables	xv
Introduction	1
Background	7
II.1 Basics of Field-Programmable Gate Arrays	7
II.2 Commercial FPGA Devices	9
II.2.1 Altera Stratix	9
II.2.2 Altera Stratix II	10
II.3 Physical Synthesis	12
II.4 Existing Academic CAD Tools	14
II.5 Summary	15
Physical Synthesis Toolkit	17
III.1 Introduction	17
III.1.1 The PST Flow	17
III.1.2 Features	18
III.2 Graphical User Interface	19
III.3 CAD Tool Interface	25

III.4	Commercial FPGA Support	26
III.5	Flexible Data Structures	27
III.6	Physical Synthesis Flow Support	28
III.7	Other features	29
III.8	Summary	30
	Functionally Linear Decomposition and Synthesis	31
IV.1	Introduction	33
IV.2	Background	35
	IV.2.1 Logic Synthesis	35
	IV.2.2 Linear Algebra	37
	IV.2.3 Notation	39
IV.3	Functionally Linear Decomposition and Synthesis	40
IV.4	Heuristic Variable Partitioning	43
IV.5	Basis and Selector Optimization	45
IV.6	Multi-Output Synthesis with FLDS	47
	IV.6.1 Multi-Level Decomposition	48
	IV.6.2 Multi-Output Synthesis	48
	IV.6.3 Multi-output Synthesis Algorithm	51
IV.7	Performance Considerations	53
IV.8	FPGA-Specific Considerations	54
IV.9	Contrast with Prior Work	55
IV.10	Experimental Results	59
	IV.10.1 Methodology	59
	IV.10.2 Results	60
	IV.10.3 Discussion of Individual Circuits	63
IV.11	Conclusion and Future Work	65

Dynamic Power Reduction	67
V.1 Introduction	68
V.2 Background and Terminology	69
V.2.1 Probability Theory	69
V.2.2 FPGA Power	70
V.3 Power Models	71
V.3.1 Average Net Toggle Rate Computation	72
V.3.1.1 Example 1 - Glitch-Free Inputs	74
V.3.1.2 Example 2 - Inputs with glitches	78
V.3.1.3 Estimation Error	81
V.3.1.4 Comments	82
V.3.2 Net Capacitance Model	83
V.3.3 LUT Power Model	84
V.4 Glitch Reduction	85
V.4.1 Negative-Edge-Triggered Flip-Flop Insertion Example	85
V.4.2 Negative-Edge-Triggered Flip-Flop Alternatives	87
V.4.2.1 Gated D Latch	88
V.4.2.2 Gated LUT	88
V.5 Optimization Algorithm	89
V.5.1 The Algorithm	90
V.5.2 The Cost Function	90
V.6 Experimental Results	92
V.6.1 Setup	92
V.6.2 Methodology	93
V.6.3 Results	94
V.6.4 Discussion	95
V.6.5 Related Works	96
V.7 Conclusion	97

- Estimation of Signal Toggle Rate 99
 - VI.1 Introduction 100
 - VI.2 Background 101
 - VI.2.1 Power Dissipation and Toggle Rate Analysis Review 101
 - VI.2.2 Existing Toggle Rate Estimation Techniques 102
 - VI.2.3 Important Aspects of Toggle Rate Computation 104
 - VI.3 Computing Toggle Rate 105
 - VI.3.1 Basic Approach 106
 - VI.3.2 Temporal Correlation 107
 - VI.3.3 Spatial Correlation 108
 - VI.4 Computing Spatial Correlation 110
 - VI.5 Experimental Results 112
 - VI.5.1 Procedure and Results 113
 - VI.5.2 Discussion 114
 - VI.6 Conclusion and Future Work 117

- Conclusion and Future Work 119

- Detailed Synthesis Algorithms 121
 - A.1 Heuristic Variable Partitioning Algorithm 121
 - A.2 Basis and Selector Optimization Algorithm 123
 - A.3 Multi-Output Synthesis Algorithm 124

- References 127

List of Figures

Figure I-1: Traditional CAD Flow	2
Figure I-2: Physical Synthesis CAD Flow	3
Figure II-1: Basic FPGA Architecture	8
Figure II-2: Stratix LE in normal mode [Altera05a]	9
Figure II-3: Stratix LE in dynamic arithmetic mode [Altera05b]	10
Figure II-4: High level diagram of the Stratix II ALM [Altera05b]	11
Figure II-5: ALM in arithmetic mode	11
Figure II-6: ALM in shared arithmetic mode	12
Figure III-1: PST Main Window	20
Figure III-2: New Project Window	20
Figure III-3: PST with a loaded project for Altera Stratix II device	21
Figure III-4: Node Properties Window	22
Figure III-5: Cell Properties Window	23
Figure III-6: Optimization Target Window	24
Figure III-7: CAD Tool Interface Flow Chart	25
Figure III-8: Relationship between nodes, nets and pins	28
Figure III-9: Applying optimizations	29
Figure IV-1: Example of synthesis of a logic function a) without using XOR gates, and b) with the use of XOR gates.	32
Figure IV-2: Column multiplicity example	36
Figure IV-3: Truth Table for Example 1	41
Figure IV-4: Gaussian Elimination applied to Example 1	42
Figure IV-5: Circuit synthesized for Example 1	43
Figure IV-6: Truth Table for Example 2	43
Figure IV-7: Heuristic variable partitioning algorithm	44
Figure IV-8: Truth Table for Example 3	46

Figure IV-9: Basis-Selector optimization algorithm	47
Figure IV-10: Multi-output synthesis example	49
Figure IV-11: Synthesized functions f and g	49
Figure IV-12: 2-bit ripple carry adder example	50
Figure IV-13: Full adder synthesized using FLDS	51
Figure IV-14: Multi-output Synthesis algorithm	52
Figure IV-15: Gauss-Jordan Elimination applied to Example 1	54
Figure IV-16: XOR gate replacement example	56
Figure IV-17: Example of XOR gate replacement	56
Figure IV-18: Example of the differences between FLDS and <i>Factor</i>	59
Figure IV-19: Distribution of area savings of FLDS combined with ABC versus ABC alone	65
Figure V-1: Examples of logic signals and properties	73
Figure V-2: Logic circuit and signal properties for Example 1	74
Figure V-3: Logic circuit and signal properties for Example 2	78
Figure V-4: Example of overlapping glitches	79
Figure V-5: Transition density estimate error	81
Figure V-6: Net capacitance estimates for nets with fanout 1 (top-left), 2 (top-right), 3 (bottom-left) and 4 (bottom-right)	83
Figure V-7: LUT Power Dissipation Model	84
Figure V-8: A three level LUT network	86
Figure V-9: Timing diagram showing the output behaviour of LUT A given a sample input ..	86
Figure V-10: A three level LUT network with a negative-edge-triggered FF inserted at the output of LUT A	87
Figure V-11: Activity of signal f after a negative-edge-triggered FF is inserted at the output of LUT A	87
Figure V-12: A three level LUT network with an inserted gated D latch	88
Figure V-13: Examples of Gated LUTs	89
Figure V-14: Power dissipation of a gated D latch (top), negative-edge-triggered FF (middle), and	

a gated LUT (bottom)	91
Figure VI-1: Correlated logic cones g and h are used as inputs to an AND gate to form function f	108
Figure VI-2: Truth tables for functions g and h	109
Figure VI-3: Correlation between $\mathbf{X}_1\mathbf{S}_1$ and h	110
Figure VI-4: Absolute Estimation Error histogram	116
Figure A-1: Heuristic variable partitioning algorithm	122
Figure A-2: Basis and Selector Optimization Algorithm	124
Figure A-3: Multi-output Synthesis algorithm	125

List of Tables

Table IV-1: Area Results Comparison Table for XOR-based Circuits	61
Table IV-2: Area Results Comparison Table for non-XOR based logic circuits	61
Table V-1: Signal Properties	72
Table V-2: Output transition table for function $f=a \cdot b$	77
Table V-3: Set of benchmark circuits	93
Table V-4: Final Results	94
Table VI-1: Toggle Rate Estimation Results in Comparison to Quartus II 7.1	113
Table VI-2: Toggle Rate Estimation Results in Comparison to approach in Chapter V	114

Chapter I

Introduction

Field-Programmable Gate Array devices, known as FPGAs, are programmable devices capable of implementing any digital logic circuit. They offer a designer the flexibility of creating a wide array of logic circuits at a low cost, because it is not necessary to manufacture a new custom made integrated circuit each time. However, the FPGA devices are bigger and consume more power than their Application Specific Integrated Circuit (ASIC) counterparts [Kuon07]. As a result FPGAs have been found to be a practical platform for medium and low volume applications.

The traditional approach to implementing logic circuits on FPGAs is to start with a description of a circuit in a Hardware Description Language (HDL) and go through a series of steps to produce an output bit stream that is used to configure an FPGA. The first step is to take a logic circuit described in an HDL and convert it into a graph of logic gates. This graph is then optimized using various algorithms in a step called Logic Synthesis. Following Logic Synthesis, the Technology Mapping step takes an optimized graph of gates and represents it as a graph of resources available on an FPGA. These resources are usually Lookup Tables, Memory and Input/Output pins. General logic functions are usually implemented as Lookup Tables, data storage modules are assigned to memory blocks and external connections are facilitated by I/O pins. A graph of hardware resources can then be placed on an FPGA, and the links between each node of the graph are realized

by routing the physical connections between logic components using a programmable routing network. This step is called Placement and Routing.

Finally, the resulting implementation is analyzed using a timing analyzer. A Timing Analyzer is a tool that computes the worst case delay information and determines the maximum clock frequency at which the circuit can operate. Results from the Timing Analyzer complete the timing analysis step, which is then followed by generation of a programming bit stream. The above steps are depicted in Figure I-1.

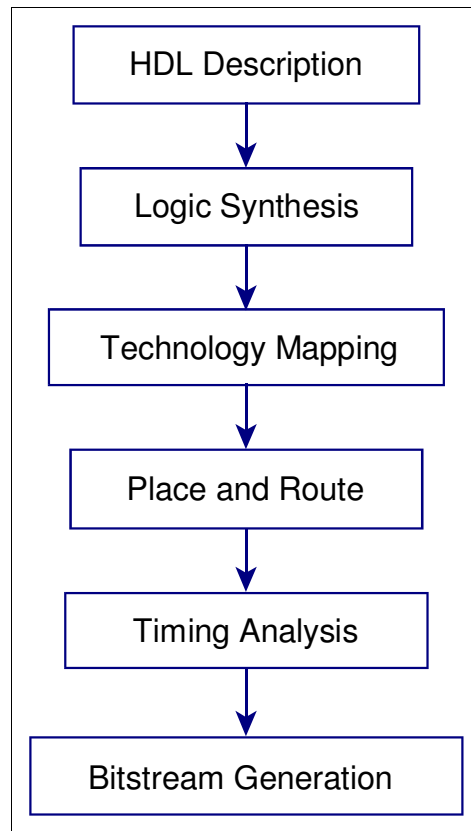


Figure I-1: Traditional CAD Flow

The traditional CAD flow is a straightforward approach to the implementation of logic circuits. As shown in Figure I-1, the CAD flow proceeds linearly and decisions made in one stage are not modified in the following stages. For example, a synthesis optimization during logic synthesis does not account for actual delays in a circuit. The delays are not known until the circuit is placed and routed. Thus, an optimization that was originally promising may turn out to be sub-optimal.

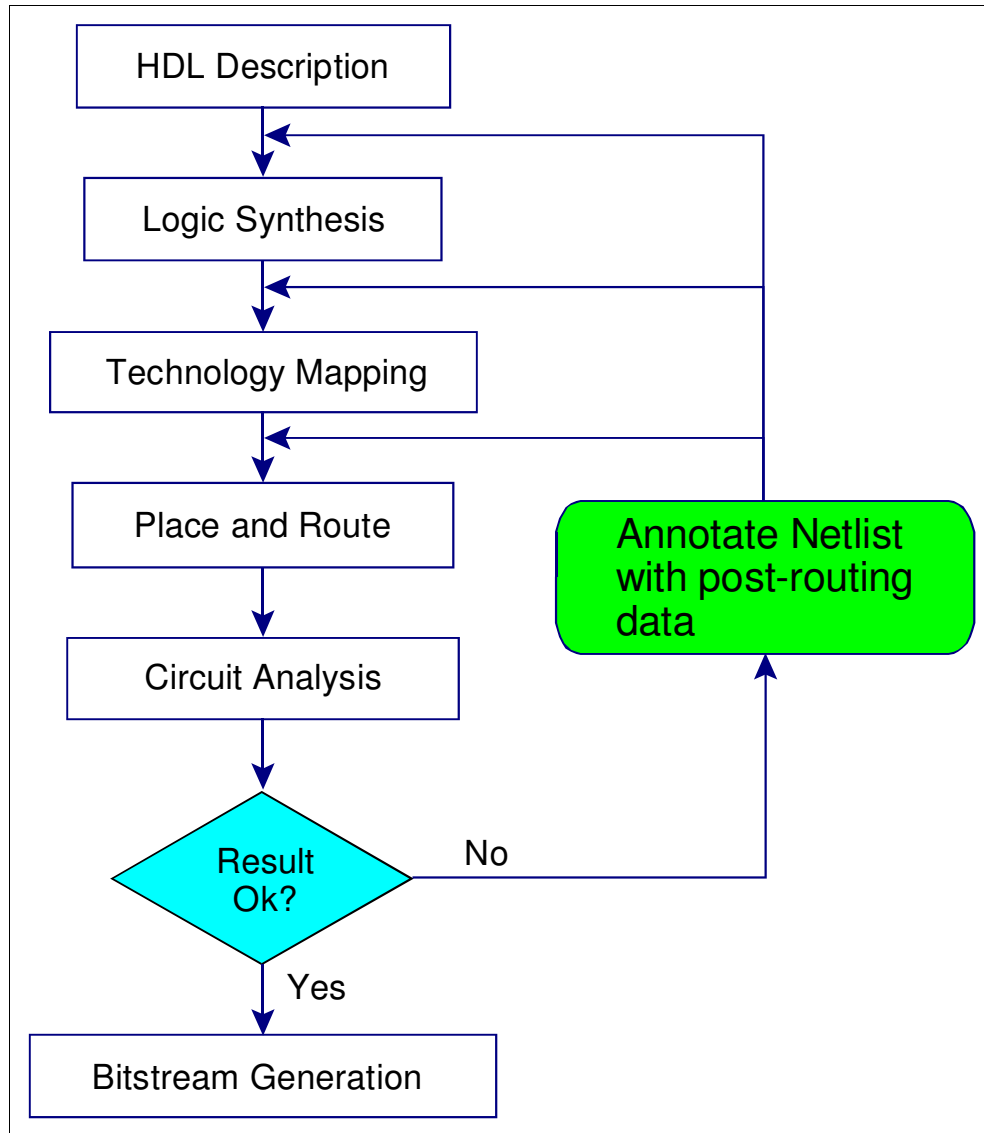


Figure I-2: Physical Synthesis CAD Flow

An enhanced approach is to employ a flow known as *Physical Synthesis*, shown in Figure I-2. Physical Synthesis is a CAD flow that can evaluate its own effectiveness and adjust its algorithms to iteratively improve its results [Singh05]. This is done by using the results of timing analysis, power analysis and others, to improve the results produced by each stage of the CAD flow. By iteratively using the results produced by the flow it is possible for Logic Synthesis, Technology Mapping and Place and Route stages to make better optimization decisions. This results in an overall better logic circuit implementation.

The Physical Synthesis approach is a relatively recent development in FPGA CAD research.

Its benefits have mostly been shown [Singh05] on commercial FPGAs, because they facilitate a wider array of implementations for logic functions. These commercial FPGAs contain dedicated circuitry to improve the performance of arithmetic circuits as well as more complex LUT based structures that can further improve the implementation of common logic functions. In addition to a more elaborate design of logic structures, dedicated blocks that perform arithmetic multiplication, multiplication and addition, as well as those for data storage are included in commercial FPGAs. In contrast, most researchers use an academic model of an FPGA, which contains only simple logic resources, due to a significant amount of effort required to create a tool that supports both an academic model and a commercial one.

This dissertation presents a unified framework, called the *Physical Synthesis Toolkit* (PST), to facilitate future research of FPGA algorithms. It is a platform capable of cooperating with current industry tools in order to implement logic circuits on commercial devices. The design of the Physical Synthesis Toolkit allows researchers to address any aspect of the CAD flow given that such support is also given by industrial CAD tools. While it provides access to commercial FPGAs, it does not forbid research on academic style FPGA architectures, as well as new architectures. It can be used in tandem with tools such as Versatile Place and Route (VPR) [Betz99] to perform such research.

A key contribution of the proposed framework is its ability to facilitate three key features of the Physical Synthesis flow. They are: the ability to perform synthesis (and/or re-synthesis), implement incremental changes to an existing design, and finally to facilitate circuit models to estimate circuit parameters. This dissertation presents three contributions, each designed to demonstrate each of the three key features of the Physical Synthesis flow. The first feature is demonstrated using a novel logic synthesis technique called Functionally Linear Decomposition and Synthesis. The second feature is presented using a dynamic power reduction technique, where negative-edge-triggered flip-flops are strategically inserted into a fully placed and routed logic circuit. The circuit changes are incremental and do not break the logical functionality of the circuit, while reducing its dynamic power dissipation. Finally, the ability to model circuit parameters is demonstrated via a toggle rate estimation technique. The toggle rate estimation technique determines a statistical probability a wire will toggle its logic value under a real-delay model, accounting for the presence of glitches in a logic circuit. This technique uses the Functionally Linear Decomposition

and Synthesis technique to account for spatial and temporal correlation between logic signals.

This dissertation is organized as follows: Chapter II presents the background information on current FPGA architectures and CAD tools. Chapter III discusses the design and functionality of the Physical Synthesis Toolkit. Chapter IV presents a novel logic synthesis technique designed for optimization of XOR-based logic circuits, while Chapter V discusses a physical synthesis approach to dynamic power reduction. Chapter VI shows how an approach to logic synthesis discussed in Chapter IV can be utilized to efficiently compute toggle rate of signals in FPGA circuits, including spatial and temporal correlation. The dissertation concludes with Chapter VII, where the avenues for future work are discussed.

Chapter II

Background

This chapter contains background information regarding the modern FPGA architectures as well as background information on physical synthesis. It highlights the differences between the academic and commercial FPGAs and helps visualize why implementing logic circuits on commercial FPGAs is more challenging than for a standard academic architecture. Then, key research in Physical Synthesis for FPGAs is presented to introduce the context in which the Physical Synthesis Toolkit will work. Finally, a review of existing academic CAD tools is given.

II.1 Basics of Field-Programmable Gate Arrays

Field-Programmable Gate Array (FPGA) devices can be thought of as rectangular arrays of logic cells (LC) connected by a programmable routing network. Each logic cell is responsible for implementing a logic function and/or a flip-flop. A logic circuit is formed by programming each logic cell to implement a particular logic function and connect them using the programmable routing network. To connect to the outside world, an FPGA uses Input/Output pins, located on its perimeter. The diagram representing an FPGA is shown in Figure II-1.

Each logic cell is connected to the vertical and the horizontal routing tracks. The logic cell

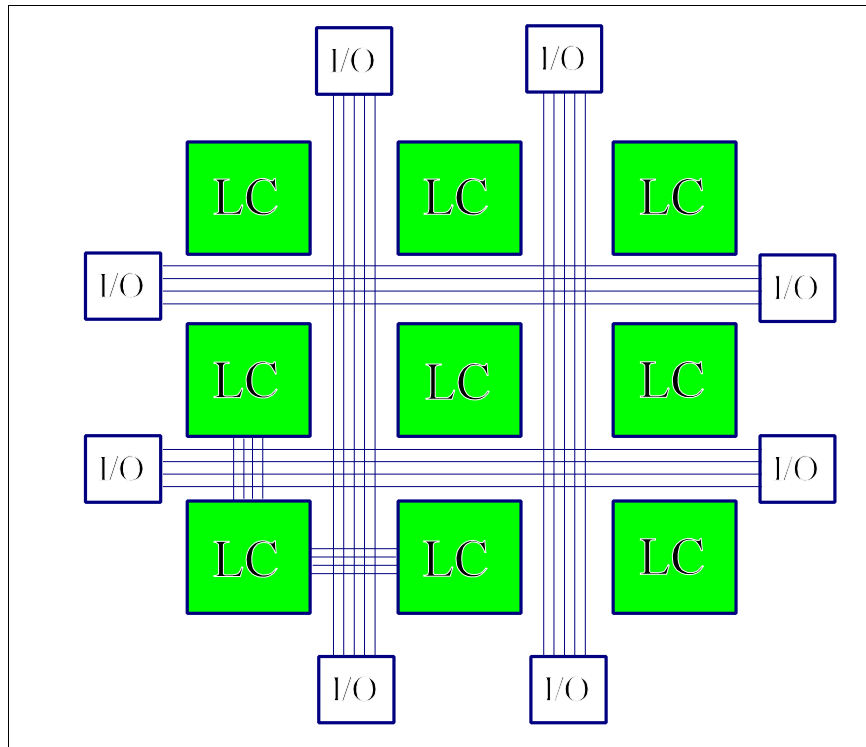


Figure II-1: Basic FPGA Architecture

contains a four input Lookup Table (LUT) and a flip-flop (FF). A 4-LUT can implement any logic function of 4 inputs, while a flip-flop is used to synchronize the operation of the circuit. A circuit composed of such logic cells can be implemented on an FPGA by placing logic cells onto the FPGA fabric and connecting them using the programmable routing network.

The programmable routing network is a set of wires and switches that allow a logic signal to travel across the FPGA. It consists of horizontal wires that allow a signal to travel left or right on the chip, vertical wires to travel up or down the chip, and finally local wires that connect a logic cell to the vertical and horizontal wires.

The above description provides a general understanding of how an FPGA device works. It is worth noting however, that in most cases an FPGA does not consist of isolated logic cells. Instead, logic cells are grouped into clusters, allowing a cluster to share local wires to speed up data transfer between logic cells inside of it. This and other changes to the basic FPGA design are described on the example of several commercial devices, considered to be state-of-the-art during the period of 2004 through 2007.

II.2 Commercial FPGA Devices

This section presents a few commercial FPGA devices.

II.2.1 Altera Stratix

The Altera Stratix is an FPGA device that consists of five major components: Logic Array Blocks (LABs) to implement arbitrary logic functions, memory blocks to store data, Digital Signal Processing blocks to speed up multiply and accumulate operations, Phase Locked Loop modules to alter the phase and the frequency of the input clock, and I/O pads to access the outside world. All of these components are interconnected by a programmable routing network [Altera05a].

Each LAB consists of 10 Logic Elements. Logic Elements (LEs) operate in the normal or the dynamic arithmetic mode. In the normal mode, the LE is configured as a single 4-input lookup table (LUT) and a register. The output of an LE is either the output of the LUT or the output of the register whose data input comes from the LUT. This mode is useful when implementing arbitrary logic functions. The functional schematic of an LE in this mode is shown in Figure II-2.

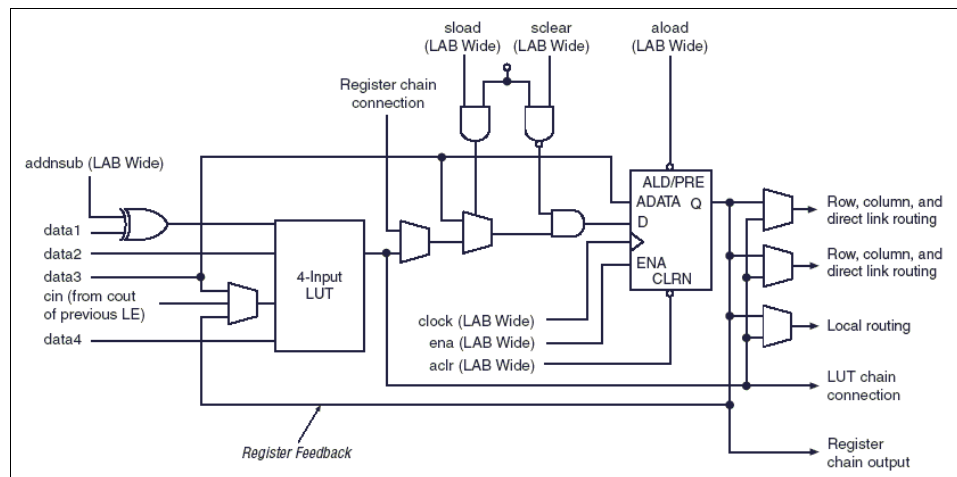


Figure II-2: Stratix LE in normal mode [Altera05a]

To efficiently implement arithmetic operations, the LE can be configured into the dynamic arithmetic mode. In the dynamic arithmetic mode the LE produces three outputs: the sum and two carry-out signals, where the carry-out signals connect to the adjacent LE via dedicated routing. The sum output is generated by one of two 2-input LUTs depending on the value of the carry-in input, where each 2-LUT computes the sum for a possible carry-in of either 0 or 1. The two carry-out

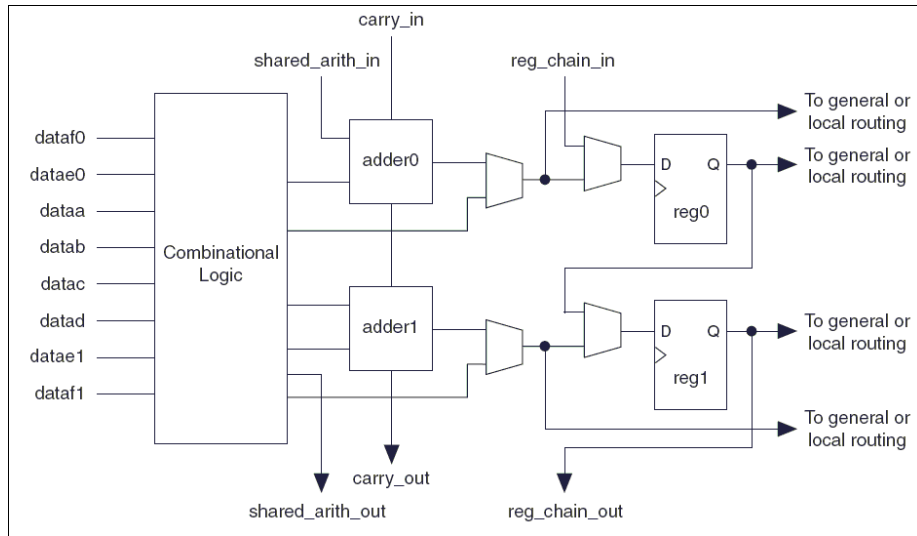


Figure II-4: High level diagram of the Stratix II ALM [Altera05b]

mode. The carry-out signal connects to the carry-in signal input of the adjacent ALM via dedicated routing, allowing for fast signal propagation. The ALM configuration in this mode is shown in Figure II-5.

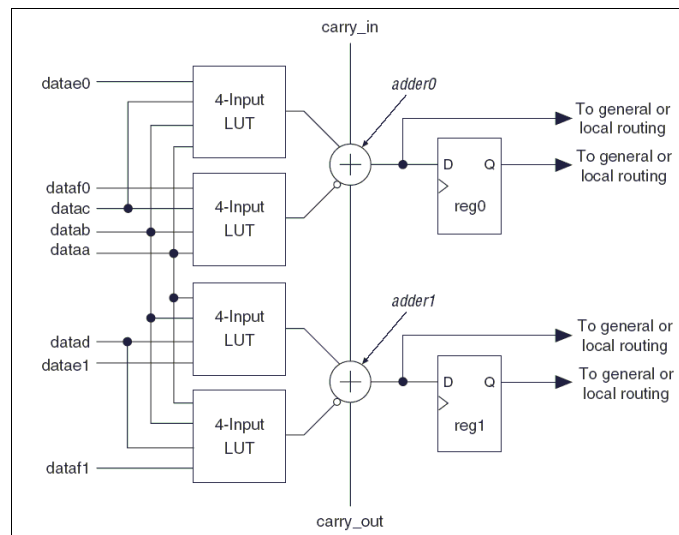


Figure II-5: ALM in arithmetic mode

The shared arithmetic mode is similar to the arithmetic mode, except that an additional carry chain can be created. In this configuration the second carry chain is fed into the next adder, which can be located in the same or the adjacent ALM. The carry-in signal for the second carry chain is the *shared_arith_in* and carry-out signal is the *shared_arith_out* signal. An ALM with these two carry

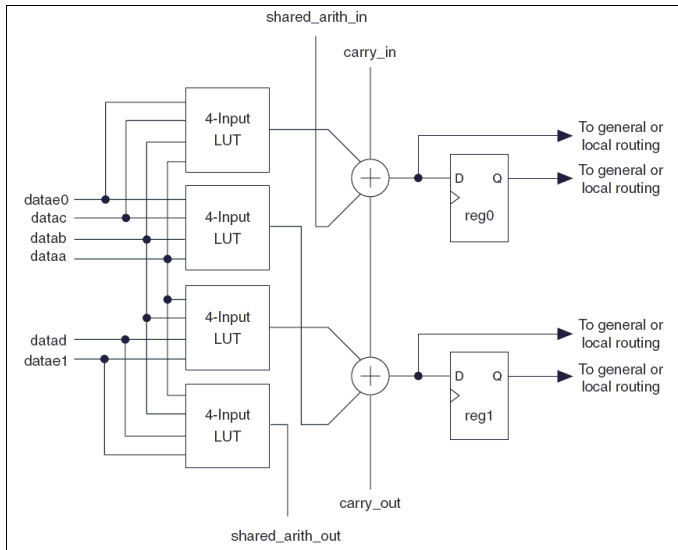


Figure II-6: ALM in shared arithmetic mode

chains is capable of implementing a circuit that adds three two bit numbers. This design is well suited for implementation of circuits such as adder trees. The ALM configuration in this mode is shown in Figure II-6.

II.3 Physical Synthesis

As previously mentioned, the physical synthesis CAD flow for FPGAs is a new approach to the implementation of logic circuits on FPGAs. The main idea behind the approach is to feed back post-routing data into the flow, allowing for better optimizations to be applied. There are currently three types of approaches that fall under the physical synthesis category. The first type applies synthesis, technology mapping and placement in an iterative process. The second type uses the synthesizer to specify to the placer where to place logic elements. This enables the placer to better understand the decisions made by the synthesizer, and possibly accommodate them. The third type permits the placer to evaluate several alternate logic mappings so that their placement can be considered.

An example of the iterative approach is given by Lin *et al.* [Lin03]. During each iteration the mapping algorithm takes some of the gates from one LUT and places them in another, basing its decisions on net delays between the gates. The new mapping is then placed again, using last placement as a guide. The work of Singh and Brown [Singh02] proposes that the placer should be

provided with an incentive to situate logic elements in a specific location on the device. Their approach starts with a regular CAD flow to obtain a synthesized and placed logic circuit implementation. Then layout-driven optimization techniques are used to reduce the delay on critical paths. Each new logic element, which is created in the process, is assigned a location that the placer aims for while minimizing the disruption to the entire logic circuit. The key contribution of their work is that the synthesizer communicates to the placer the intended location of synthesized logic elements, thus allowing the placer to respond accordingly.

The previous two examples maintained the separation between the logic synthesis and the placement stages. The approach proposed by Lou *et al.* [Lou99] breaks this boundary by having the synthesis stage provide several mapping solutions for a subcircuit it considers to be good. The placer then chooses the mapping solution to improve the speed of the logic circuit, since speed is easier to estimate during placement and routing stages.

In the context of commercial FPGAs, a complete physical synthesis flow has been implemented by Singh *et al.* [Singh05] for the Altera Stratix and Stratix II devices. The proposed flow follows the idea of Physical Synthesis and focuses on two areas of optimization: post-technology mapping and post-placement. The post-technology mapping optimizations can address the problem of optimization at a coarse granularity, but require accurate timing models. While for non-critical paths it is not always possible to predict the wiring delay, it is possible to obtain a reasonably good estimate for timing critical ones. This is because timing critical paths have priority to use fast routing resources in order to improve circuit performance. Paths with a lot of slack may have different routing delays due to their high slack and the need to resolve congestion during routing.

The post-placement optimization techniques in [Singh05] focus on improvement of critical paths that could not be addressed at the coarse granularity. At those stages large benefits can be observed as the delays between logic components are better defined. Thus, a critical path can be identified much more accurately, allowing physical synthesis optimizations to focus their efforts better.

Further discussion of works related to layout-driven optimizations and physical synthesis can be found in [Chen06].

II.4 Existing Academic CAD Tools

Currently there are several tools available for use by the academic community. They are: the ABC Logic Synthesis System [ABC05] developed at the University of California at Berkeley, BDS-PGA [Vemuri02] developed at the University of Massachusetts at Amherst, technology mapping package RASP [Cong96], and VPR [Betz99] place and route tool developed at the University of Toronto.

The ABC logic synthesis system [ABC05] is a successor to the popular SiS system [SiS94]. It uses AND/Inverter graphs to represent logic circuits, allowing for an efficient memory storage. It manipulates the AND/Inverter graph in order to optimize the circuit and reduce the depth as well as number of gates a circuit occupies. The optimizations in ABC focus on local changes to the structure of the AND/Inverter graph, using structural and BDD optimization on a small scale. In order to achieve better results on larger circuits, the system applies local operations over the entire circuit, allowing the small changes to affect the entire circuit one step at a time. With an efficient data representation and use of Binary Decision Diagrams (BDD) [Bryant86], the ABC system is able to perform logic optimization rapidly. To the author's knowledge, the ABC system is the fastest currently available for use by academic researchers. Unfortunately, ABC system is unable to efficiently handle large cones of logic, which does not permit it to take advantage of more coarse-grained optimizations.

Another synthesis system currently available is BDS-PGA [Vemuri02]. Similarly to ABC, it performs logic optimization on a gate level netlist, though the netlist can be composed of any set of gates of up to two inputs. BDS-PGA focuses on using BDD based techniques to address a wide variety of logic circuits. One of its strengths is the ability to synthesize XOR-based logic circuits well. However, it is slower than the ABC system and performs logic synthesis one cone of logic at a time by partitioning the circuit into maximum fanout free cones. It has been augmented to allow preprocessing of the logic graph to find good partitioning of logic cones to further minimize area, but the partitioning depends greatly on the initial logic graph.

The RASP [Cong96], or RAPid System Prototyping package developed at the University of California in Los Angeles, is a synthesis and technology mapping package. It consists of a number of works including FlowMap [Cong94] and most recently DAOmap [Chen04]. This package

provides researchers with a variety of algorithms suitable for decomposition and technology mapping of boolean logic circuits into LUT-based FPGAs.

Finally, a widely available CAD tool for FPGA is VPR [Betz99]. VPR is a place and route tool that uses simulated annealing to place logic cells on the FPGA fabric. It has become a popular research tool due to its ability to conduct architecture research. It also includes timing and power models that are useful in architecture research. In its current release it only supports logic cells as part of the FPGA architecture. Unfortunately, it does not support incremental changes to a logic circuit.

II.5 Summary

In this section several related works on physical synthesis, as well as details of some modern FPGA devices, were presented. It is evident from the presented works that in order to facilitate physical synthesis for modern FPGA devices, as well as the future ones, we need to take the following aspects into consideration:

1. FPGA devices are *diverse*. Many devices, even ones produced by the same company have a wide array of specialized components and varying structure of logic elements.
2. The ability to *change logic* implementation of a circuit post-routing is important, as it can meet the needs of a particular circuit.
3. *Incremental changes* are necessary to ensure that changes to the circuit post-routing only improve the sections of a circuit we are interested in. The remaining parts of a circuit should remain unchanged.
4. Ability to *model* circuit parameters, such as delay, area and power are necessary to successfully effect changes in an FPGA circuit.

The following chapters describe a Physical Synthesis Toolkit that addresses the above criteria. The toolkit is able to target the Altera Stratix and Stratix II devices to show its ability to target diverse architectures, and facilitates incremental as well as full synthesis changes to the circuit. To guide such changes well, the toolkit easily facilitates modeling techniques.

Chapter III

Physical Synthesis Toolkit

This chapter discusses a new research framework called the *Physical Synthesis Toolkit*. The toolkit, or PST for short, is a software package designed to operate in tandem with commercial CAD tools, allowing researchers to target their FPGA algorithms for commercial devices.

III.1 Introduction

The Physical Synthesis Toolkit (PST) is a software package intended for use in an academic setting. It has been designed with the end user in mind, in a hope that future researchers will be inclined to target commercial devices, without spending excessive amount of time on developing their own research platform. The software package is easily extendable to support more FPGA devices and CAD tools, with little effort on the side of the user.

III.1.1 The PST Flow

The Physical Synthesis Toolkit was designed to work with a simple flow. It is described by the following 6 steps:

1. Design, synthesize, place and route a design using a commercial CAD tool.

2. Open the design into PST in tandem with a commercial CAD tool.
3. Apply changes to the circuit inside of PST and transmit them to a commercial CAD tool.
4. Receive validation for changes from step 3 from a commercial CAD tool.
5. Repeat steps 3 and 4 until no more changes are needed.
6. Close the design both in PST and in a commercial CAD tool, preserving applied changes.

The flow is sufficiently general to facilitate any change to a design. Furthermore, it represents at a high level the Physical Synthesis flow, thereby facilitating Physical Synthesis research.

III.1.2 Features

The Physical Synthesis Toolkit includes a multitude of features that make it a useful package for researchers. The features are listed below:

1. Easy to use Graphical User Interface
2. Interface to commercial CAD tools. Currently, an interface for Altera Quartus II is implemented. A similar interface can be easily added to support Xilinx ISE CAD tool.
3. Support for commercial FPGA devices, including:
 - a. Altera Stratix
 - b. Altera Stratix II
4. Support for complex FPGA blocks and logic features, including:
 - a. Stratix carry chains
 - b. Stratix II ALM, including the carry chains
 - c. Random Access Memory blocks, for both Stratix and Stratix II devices
 - d. Digital Signal Processing blocks for Stratix and Stratix II devices
 - e. I/O pads
5. Flexible data structures
 - a. The same data structures support a wide variety of FPGA devices
 - b. Logic circuit is represented as a set of nodes, nets and pins, making it easy to describe any logic circuit for any device
6. Support for the Physical Synthesis flow by:
 - a. allowing incremental changes to a logic circuit

- b. providing tools for modeling and re-synthesis of a logic circuit
- 7. Circuit analysis tools, including:
 - a. timing analysis
 - b. toggle rate analysis, described in detail in Chapter VI
- 8. Circuit optimization tools, including:
 - a. an XOR-based logic decomposition technique, described in detail in Chapter IV
 - b. a power reduction technique, which utilizes negative-edge-triggered flip-flops to reduce glitching power, described in detail in Chapter V
- 9. Extendable code base
 - a. adding support for new CAD tools and FPGA devices is simple
 - b. adding new optimizations is a straightforward process
- 10. External packages to support:
 - a. XML parsing - Xercesc 2.6 XML parser package, compiled into a Dynamically Linked Library
 - b. BDD representation - CuddBDD package, version 2.4.1, developed by Fabio Somenzi [Somenzi241]. The package is compiled into a Dynamically Linked Library.
 - c. ARCH/ICD parsing - architecture file and intracell delay file parsing source code provided by Altera Corporation.

The following sections describe each of the PST features in greater detail. Please note that the source code for the Physical Synthesis Toolkit has in excess of 80000 lines of code, and hence not all of the details can be included in the following description.

III.2 Graphical User Interface

The Graphical User Interface, or GUI, is the part of the software the user interacts with. It is used to display pertinent information about a design the user works with. It also permits the user to set up and initiate circuit optimizations.

Upon starting the program, the user is presented with a window shown in Figure III-1. The window consists of four main parts: a menu at the top, a command window at the bottom, a hierarchy

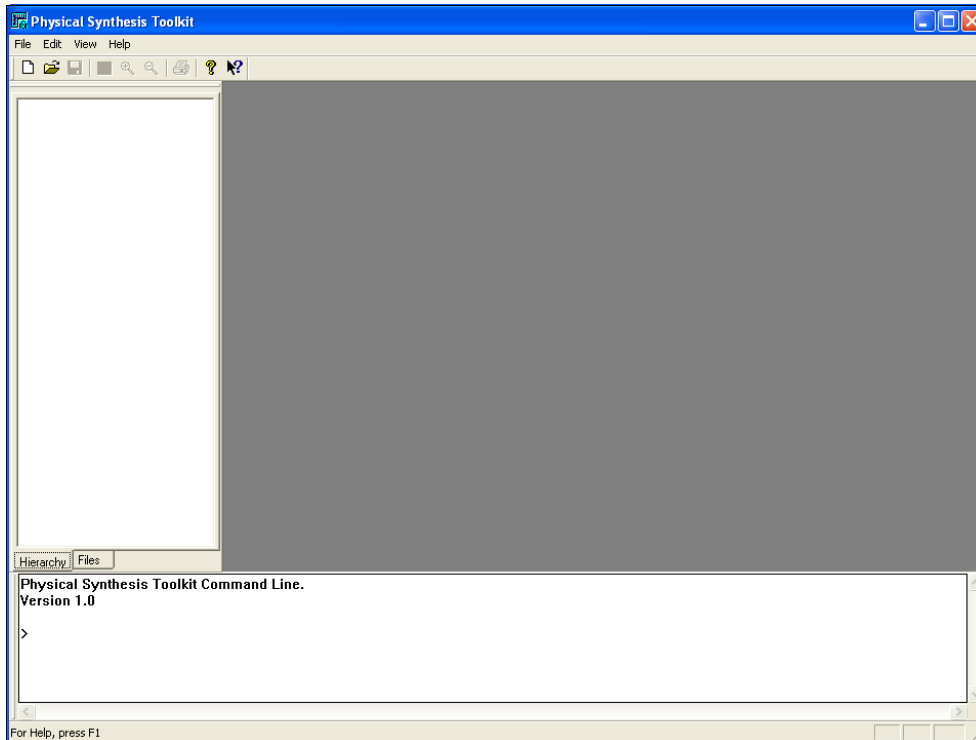


Figure III-1: PST Main Window

view on the left, and a workspace on the right. To begin working with the PST it is necessary to open a project. To do so, the New Project, or Open Project, commands can be used. Executing either of these commands opens the New Project Window, shown in Figure III-2.

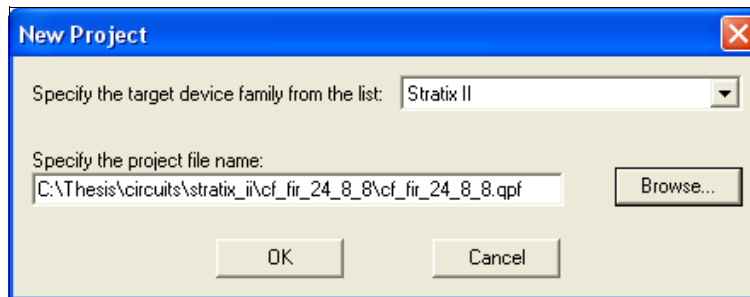


Figure III-2: New Project Window

The New Project Window is a simple window that allows the user to pick an existing project to work on. In particular, when working with commercial CAD tools, it is good to point to projects created by those tools and use them as input to PST. For example, to load a project *cf_fir_24_8_8*, the New Project Window would be set up as shown in Figure III-2. In addition to setting up the reference to the Quartus II project file, we need to specify the target device to be Stratix II. This is necessary to allow PST to read the specified project file and initialize CAD tools to work with Stratix

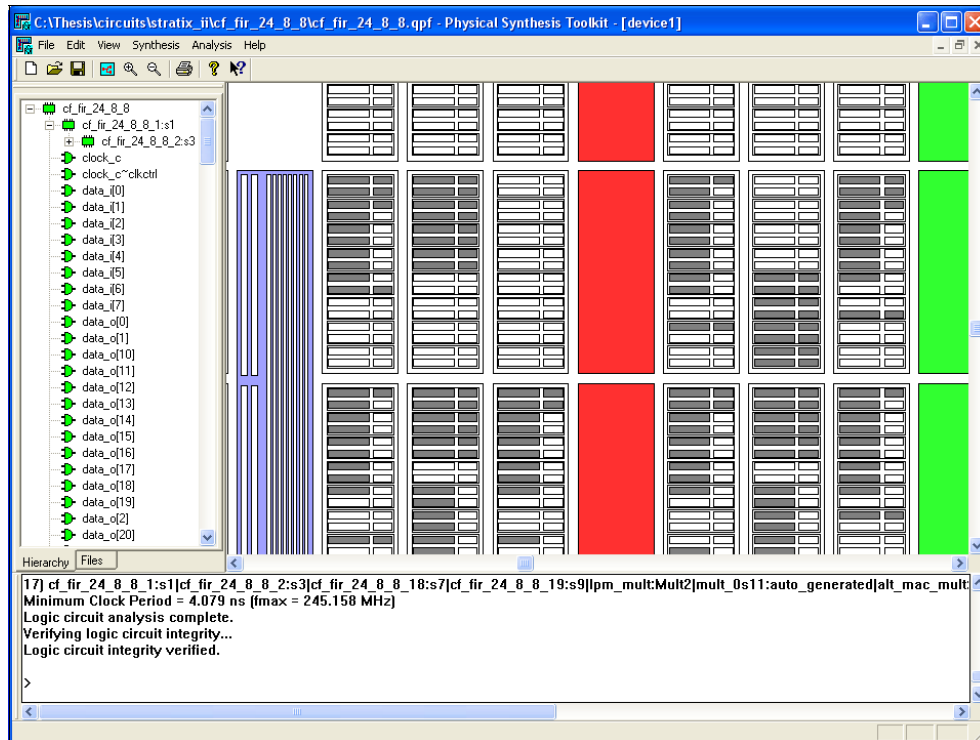


Figure III-3: PST with a loaded project for Altera Stratix II device

II FPGA device. Once a project is selected, we press the *Ok* button to load a project.

Once a project is loaded into PST you will notice a number of changes to the display, as shown in Figure III-3. The changes include: new menus (Synthesis, Analysis) are now available, some of the toolbar icons are now active, the hierarchy window contains a list of nodes in the design, and the command window displays a number of messages. In addition, if the *View Chip* command was selected from the View menu, or by pressing the *View Chip Window* button on the toolbar, a window showing the FPGA device and the design mapped onto it appears on the right hand side.

The new menus to appear, the Synthesis and Analysis menus, contain options available for use in a given target device. The synthesis menu contains synthesis optimizations implemented for the given FPGA device, while the analysis menu contains tools to perform circuit analysis, such as timing and toggle rate analysis. Both menus are device sensitive, that is they only contain options available for a given device.

A new tree appears on the left hand side of the screen. The tree represents the hierarchical structure of a logic circuit for a given project. Notice that each node in this tree is associated with an icon - an AND gate icon representing a single node in a design, and a chip-like icon representing

a larger logic module consisting of many nodes. To see the nodes contained within a logic module, press the + button on its left hand side and the module will be expanded to show its contents. To see the properties of a particular node, you can right-click on a specific node and select *Node Properties* from the popup menu.

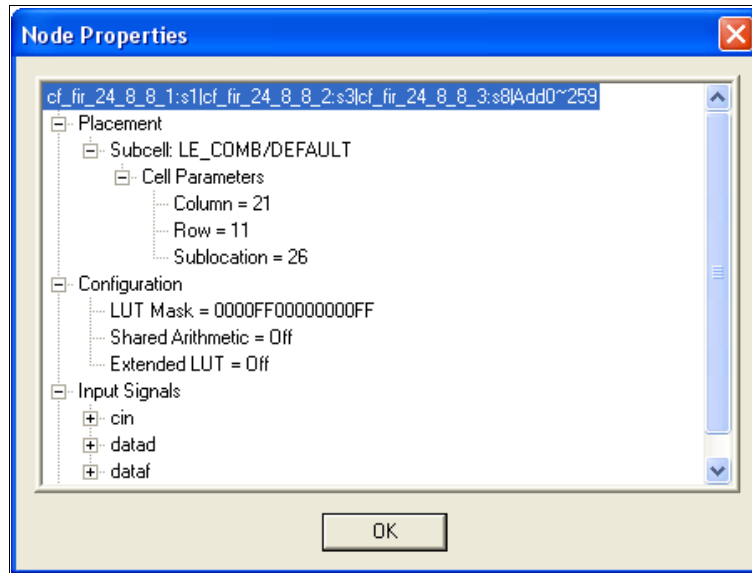


Figure III-4: Node Properties Window

The Node Properties window, shown in Figure III-4, details information about a particular node. The information about a node is organized in a tree structure, where the node name appears at the top and several categories are displayed beneath it. They are: placement, configuration, input signals and output signals. Each of the categories can be expanded to view the information contained within by pressing the + sign on its left hand side. In the example above we can see the placement information for the node as well as the configuration flags for the node and some of the input signals connected to the node.

In addition to the new options node available through the menus and the hierarchy display described above, notice the new messages that appeared in the command window, as shown at the bottom of Figure III-3. These messages are a result of loading the project into PST. During that process several different procedures are executed to properly read the design into PST. These procedures, such as timing analysis or circuit verification, produce output visible to the user in the command window. A timing analyzer is run by default and thereby provides information about the design performance. A circuit verification routine on the other hand produces warnings, or errors,

if a particular design contains problems PST is not equipped to handle. For example, a circuit with multiple clock signals, or with a derived clock, would cause the PST to produce an error and abort the project loading process.

Lastly, on the right hand side of Figure III-3 a portion of the Altera Stratix II device is shown, along with some circuit nodes placed on the circuit. In the figure, a number of elements are drawn in grey colour, indicating occupied Adaptive Logic Modules (ALMs) of the Stratix II device. Each ALM consists of four blocks, two wide rectangles representing Lookup Tables (LUTs) and two smaller rectangles representing the associated flip-flops. In addition to the ALMs, the figure shows three other types of elements on the FPGA device. They are: a blue Digital Signal Processing block on the left hand side, a red small memory block (M512 RAM block) and a green medium size memory block (M4K RAM block).

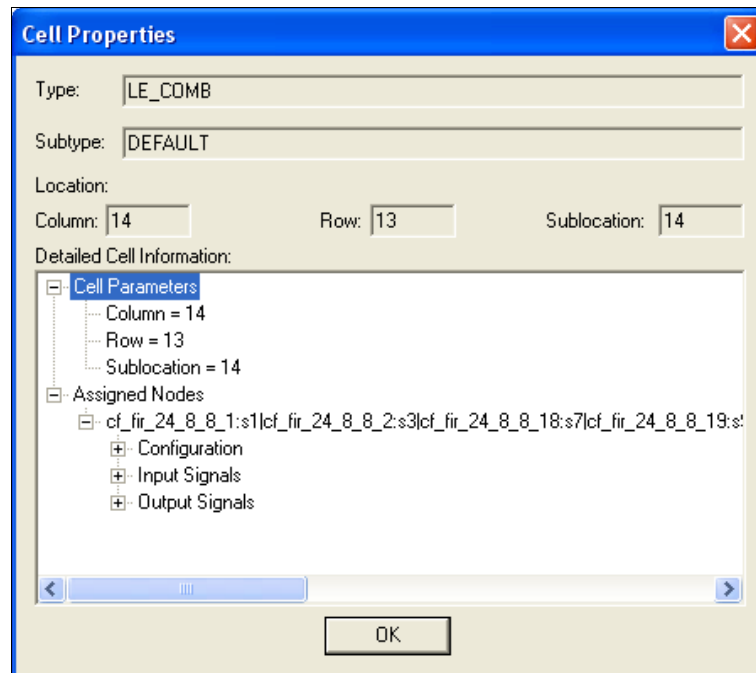


Figure III-5: Cell Properties Window

To view the contents of any particular logic element on the FPGA device it is possible to double-click it to display its properties. When you do so, a window called *Cell Properties*, shown in Figure III-5 will appear. This window contains the type and location information for the selected cell, including a list of nodes assigned to it. Each node in the list is an expandable tree that contains the same information as is shown in the Node Properties window described earlier. The only

difference is that while a Node Properties window displayed information for a single node, a Cell Properties window may contain data on several nodes mapped into the same cell. This is particularly important for FPGA devices where multiple nodes can occupy the same cell and their combined configuration constitutes the full description of a logic cell. An example of such case is the Altera Stratix logic element, where a LUT node and flip-flop node can both occupy the same cell, indicating that a LUT and a flip-flop are packed together into the same logic element.

With the ability to view the design as described above, the next step for PST is to provide an interface to facilitate Physical Synthesis optimizations. This functionality is accessed through the Synthesis menu. This menu contains a list of available optimizations for a given device as well as a *Run* command. When executed, the Run command causes the *Optimization Target* window to be displayed, shown in Figure III-6.

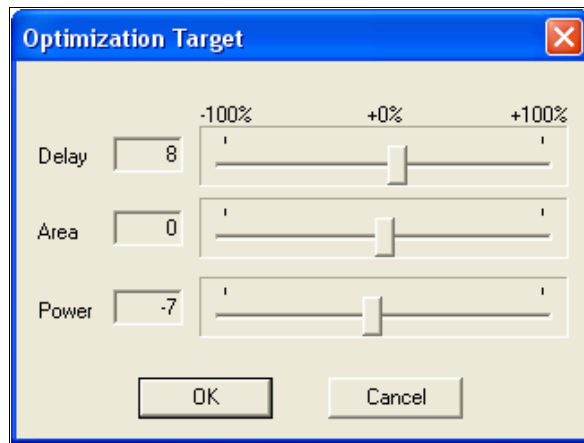


Figure III-6: Optimization Target Window

The Optimization Target window allows the user to pass desired delay, area and power information to the optimization algorithms. The idea behind this approach is to allow the user to specify the trade-off between the three parameters to optimize their circuit to meet the desired specifications. The delay, area and power information is represented using slider bars, where the values indicate the percent change in the circuit parameters that is desired. It is up to the author of the algorithm to decide how to use such information to better guide their optimization algorithm. For example, the settings provided above can be used to guide a power optimization algorithm to attempt to reduce power dissipation by at least 7%, provided that the delay penalty resulting from a given optimization does not exceed 8% and the total circuit area is not increased.

III.3 CAD Tool Interface

The Graphical User Interface presented in the previous section shows the information and options available to the user. However, there are some aspects of PST that work in the background. In particular, there is a CAD Interface that allows PST to interface with existing CAD tools in order to perform circuit optimization. In general, the CAD Tool Interface performs operations as shown in the flow chart in Figure III-7.

The CAD Tool Interface is instantiated every time a new project is created. At that time the CAD tool interface initializes 3rd party software to communicate with. When the 3rd party tool loads, PST loads in a project by communicating to the external CAD tool through the interface. This step usually entails loading the project in the CAD tool and then generating the netlist and placement information in a format PST can read. Once these steps are complete, the CAD interface can be used to perform optimizations on a logic circuit. The exact mechanism is described later, however the

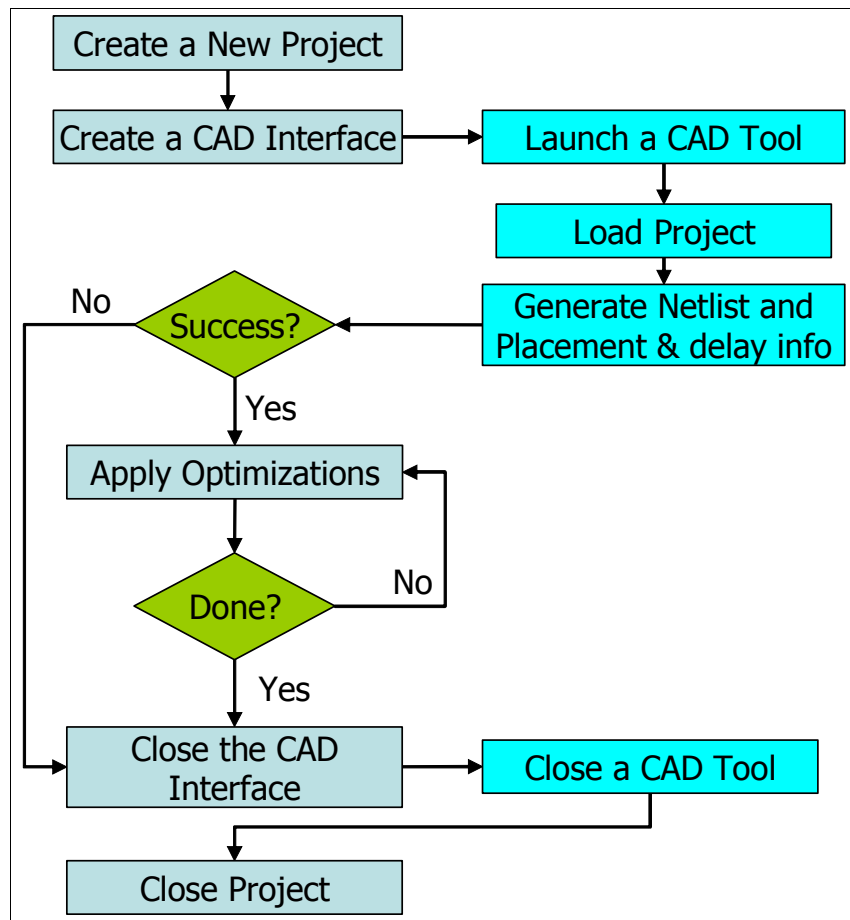


Figure III-7: CAD Tool Interface Flow Chart

general idea is to apply optimizations one at a time, until the optimization process is complete. At that point a user can request the project to be closed, which first ensures to close the CAD interface, terminating any 3rd party CAD software running in the background.

The CAD Tool interface is a standardized interface within PST. It consists of procedures necessary to inflict changes onto a logic circuit implemented using a particular set of CAD tools. The main idea is to allow PST to add/remove nodes, change logic connections and alter the placement of nodes in a design. The CAD Tool interface is generalized to the point where the PST user interface need not be concerned with how the interaction with the CAD tool actually occurs, but rather focuses on simply requesting from the 3rd party CAD tool to implement incremental changes in a design and waits for confirmation that the request has been completed.

An example of a CAD Tool interface is the interface to Altera Quartus II. This interface works by the means of named pipe between PST and `quartus_cdb.exe`, launched in the background. The interface is implemented using text-based commands and TCL scripts that have been configured to cause Quartus II to perform desired actions. These include: addition, removal, and modification of a logic element, creation and removal of nets, as well as a few others. At present only the Quartus II Interface has been implemented, however the framework supports interface with other CAD Tools as well.

III.4 Commercial FPGA Support

One of the main goals of the Physical Synthesis Toolkit, as well as the motivation behind interfacing with existing CAD Tools, is to support modern commercial FPGA devices. As mentioned in Chapter II, modern devices are quite complex and diverse in the logic components they provide. In particular, this work includes the support for Altera Stratix and Stratix II devices.

Both devices contain several different types of logic blocks in addition to Lookup Tables and Flip-Flops. Most notably there are memory blocks and Digital Signal Processing blocks. These blocks are displayed in the chip view window as described earlier, and any logic component that is placed within them will be included in the circuit. The inclusion of such complex blocks is an important contribution of the Physical Synthesis Toolkit. This is because most large circuits utilize these blocks in addition to Lookup Tables and Flip-Flops. In the Physical Synthesis Toolkit these

blocks are included in the timing analysis process and the connections to regular logic are visible.

It is important to note however, that the support for such blocks is limited to what commercial CAD Tools permit the PST to do with them. For example, in the current release of Quartus II, we cannot change the placement of memory or DSP blocks by requesting a placement change through the Quartus II interface. This requires a full recompilation of the design, though it may be permitted in the future releases of Altera Quartus II software. Nonetheless, connections to and from such blocks can be altered allowing the logic components and connections between them to be modified.

III.5 Flexible Data Structures

To support various logic modules on modern FPGA devices, PST uses flexible data structures that can be used to describe FPGA devices. The basic idea is to model an FPGA device as an array of elements. Each element can have elements within it, thereby forming a hierarchy of elements. Each of the elements has a type associated with it to differentiate it from other elements on an FPGA device.

The generic structure of FPGA elements also requires a generic structure of a logic circuit to go along with it. In this work a logic circuit is represented using three data structures: a node, a net, and a pin. A *node* is a structure that holds the information necessary to configure any particular element in an FPGA. It is responsible for not only storing the configuration data for any element, but also sufficient data to identify FPGA elements with which it is compatible. The nodes are connected to one another through *nets*. A net is a structure that abstracts the physical connection between elements on an FPGA device. Finally, there is a *pin* structure that allows nodes and nets to be linked. In particular, a pin refers to a connection point between a node and a net. For example, a connection from element **A** to **B** via net **temp** is implemented by creating an output pin on node **A** and an input pin on node **B**. A net then uses the output pin on node **A** as a connection source and an input pin on node **B** as a sink (destination). The net itself only contains information about pin it is attached to, and only through the pins it can determine the actual node it is connected to. This idea is illustrated in Figure III-8.

The advantage of using these data structures is their ability to describe any logic circuit.

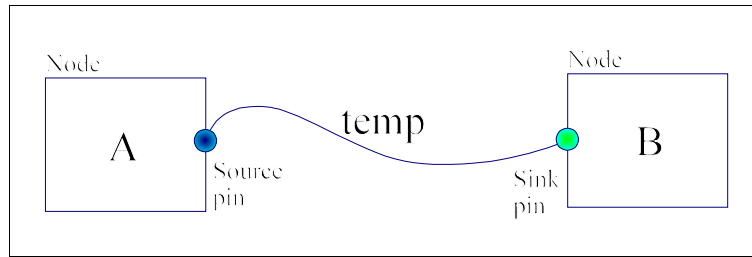


Figure III-8: Relationship between nodes, nets and pins

Nodes can be used to describe any logic component by allowing them to carry configuration flag. A flag can both identify a type of an element, which a node is compatible with, as well as a means to configure the element properly. Also, this allows a node to be a generic enough representation of a logic circuit element that incorporating new FPGA devices into PST is not arduous. Second, using pins to represent unique points of contact on any particular node allows each connection point on a node to have different parameters, such as signal arrival and required times, as well as toggle rates. This is especially important in modern architectures where a distinction between connection points allows a CAD tool to take advantage of varying propagation delay through logic elements as well as facilitating constraints on use of dedicated resources, such as carry chains for example.

III.6 Physical Synthesis Flow Support

With the above setup of the FPGA architecture and the data structures to represent a logic circuit, a Physical Synthesis flow can be supported. Within the Physical Synthesis Toolkit this flow is supported by means of incremental changes to an already placed and routed circuit. To facilitate such changes, PST decomposes all optimizations into micro changes that alter a logic circuit one node, net, or pin at a time. By doing so it is easy to use a CAD Tool interface described earlier to request and validate changes made during and after an optimization.

The flow chart representing the process of applying optimizations to a logic circuit is described in Figure III-9. First, an optimization is decomposed into a set of basic commands (micro changes), such as create node, add pin, add net, etc. Then, the list of micro changes is processed in an iterative fashion, by first applying a change on the side of a 3rd party CAD tool, and then if successful, implementing the same change in local data structures. If all micro changes succeed, the circuit is modified and the operation is considered a success. On the other hand, if at some point

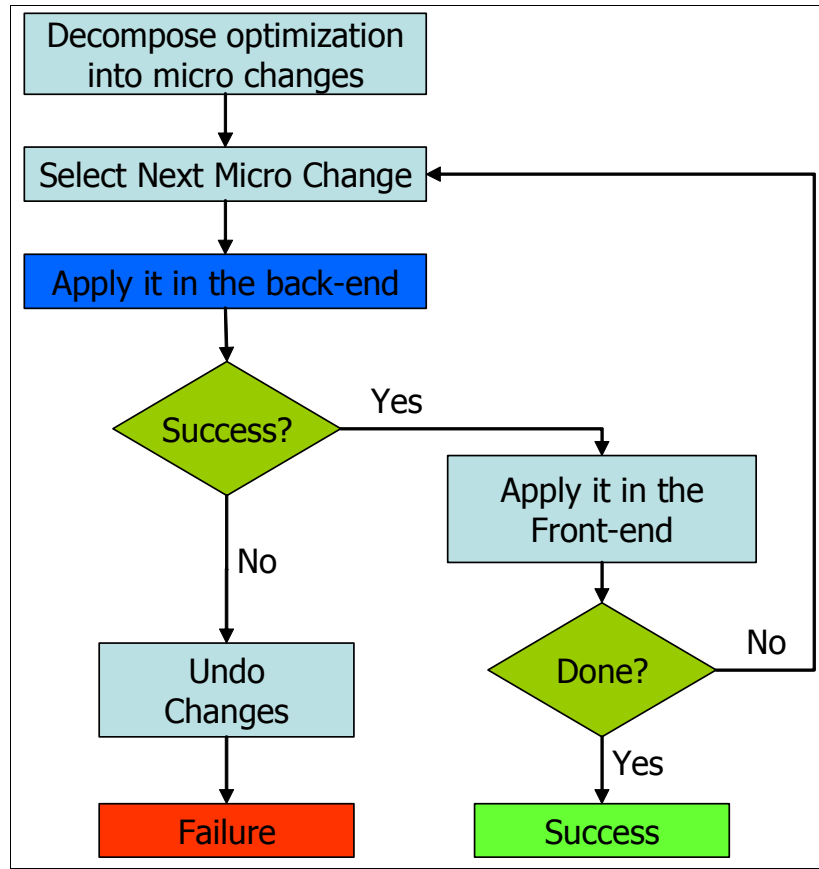


Figure III-9: Applying optimizations

application of a micro change fails, the PST software will undo all previous changes in a reverse order, both remotely on the 3rd party CAD tool and locally in PST data structures, to restore the circuit to its previous valid state.

The above approach allows for circuit optimization to happen one optimization at a time, updating the final implementation of the circuit at each step, while maintaining complete timing and placement information about a design. This is the heart of Physical Synthesis.

III.7 Other features

In addition to the above described features, the Physical Synthesis Toolkit contains a variety of tools that can aid in the implementation of physical synthesis optimizations. Some of the tools aid in the analysis of logic circuits and their re-synthesis. Others range from file parsers and readers to 3rd party tools that support Binary Decision Diagram manipulation [Somenzi241] and XML

processing. These tools are an essential part of the toolkit, as they support the operation of PST. However, they can be replaced with other existing tools and hence are not described in this dissertation as a technical contribution.

III.8 Summary

The Physical Synthesis Toolkit described above provides several contributions to the FPGA field. First, it is the first software package to support modern FPGA devices, such as the Altera Stratix and Stratix II. The toolkit supports flexible data structures that make it easy to augment the toolkit with support for new devices. In this author's experience it took approximately a month to add Altera Stratix II FPGA support, once the toolkit was operational and supported the Altera Stratix device. As these devices are quite different in the logic components they utilize, this indicates that the Physical Synthesis Toolkit is flexible enough for use in an academic setting.

It is also the first academic toolkit to support the physical synthesis flow. Some of the key elements of the flow include: providing synthesis tools to optimize a logic circuit, implementing incremental changes in a step by step manner, as well as modeling circuit parameters ranging from timing to power. In the following chapters technical contributions that address each of the above points are presented.

Chapter IV

Functionally Linear Decomposition and Synthesis

In the previous chapters the Physical Synthesis Toolkit was discussed, showing how a researcher can utilize it to work in the context of Physical Synthesis. The toolkit has been shown to provide an ability for the user to work with the three key features of the Physical Synthesis flow. In this chapter the first feature is addressed, namely Logic Synthesis.

As previously discussed, Logic Synthesis is responsible for restructuring a logic network to reduce the total circuit area and optimize the speed. Synthesizing logic circuits is not a straight-forward process, because different logic functions require widely varying approaches to successfully create a good logic network for them. A particularly challenging class of logic circuits are those that heavily depend on Exclusive-OR (XOR) logic gates for their implementation.

The class of XOR-based logic circuits presents a unique challenge for logic synthesis. Unlike for their AND/OR counterparts, a good decomposition of XOR-based logic is not immediately evident by looking at the truth table of a logic function [Sasao99]. The approaches currently used to address this problem focus mostly on the use of Binary Decision Diagrams (BDDs). BDDs are used to represent a logic function efficiently [Bryant86], and the structure of the BDD itself is analyzed to identify the location of XOR gates. In principle, a BDD is a mechanism to rapidly process a logic function in a manner similar to the classical work of Ashenhurst and Curtis [Ashenhurst59, Curtis62,

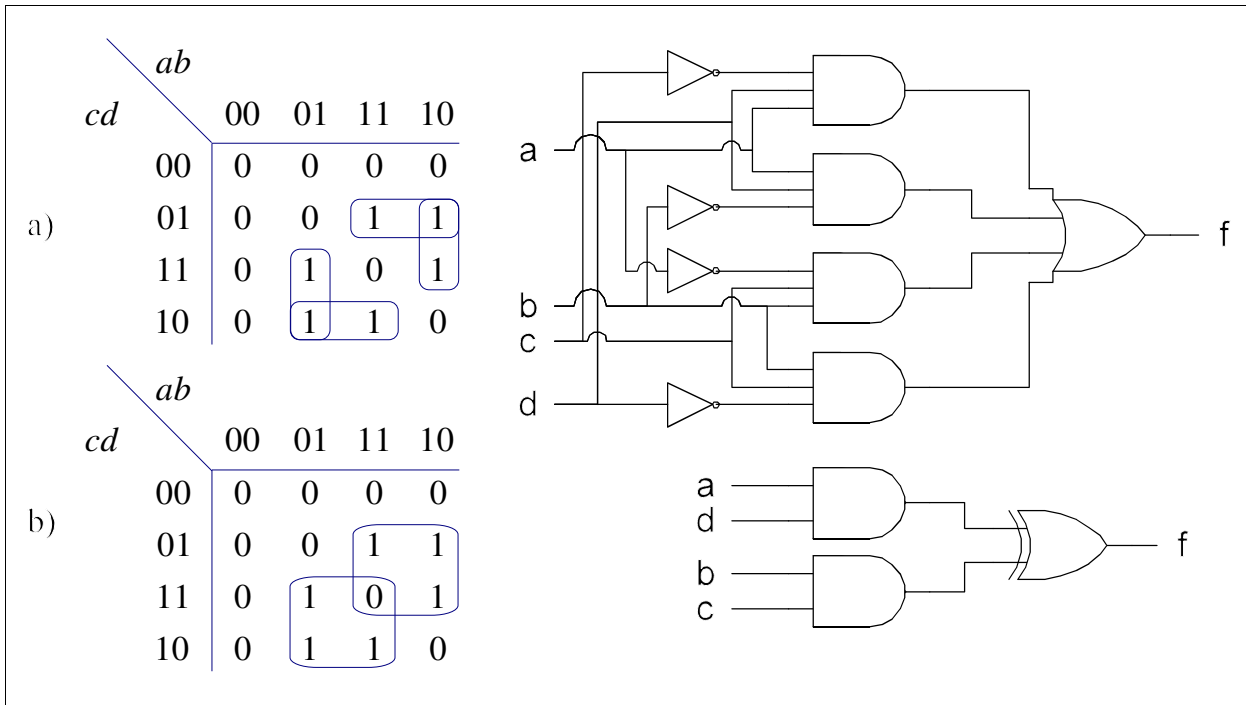


Figure IV-1: Example of synthesis of a logic function a) without using XOR gates, and b) with the use of XOR gates.

Perkowski94]. However, it does not address the fundamental problem of exposing the XOR-based logic decomposition.

A simple example of the effectiveness of XOR gates is shown in Figure IV-1. In this example a four input logic function is implemented two different approaches: one without the use of XOR gates, and one that takes advantage of the optimization opportunities provided by XOR gates. First, consider the implementation shown in Figure IV-1a, where XOR gates are not used. In this case the logic function is synthesized as a logical OR of four product terms indicated on the Karnaugh map. The resulting logic network consists of four AND gates, four inverters and an OR gate as shown. This implementation can be improved by using XOR gates as shown in Figure IV-1b. In this figure, the same logic function is implemented, however product terms used in its implementation encompass a logic 0 value at $abcd=1111$. Although each of the product terms, ad and bc , will produce a logic 1 for this input pattern, using an XOR of these two product terms allows the logic function to be implemented correctly. The resulting implementation is much smaller and uses only three 2-input gates. Although in this example realizing the need for the use of XOR gates to reduce logic area was relatively straightforward, in general the problem is much more challenging.

This chapter presents a novel approach to the synthesis of XOR-based logic circuits called Functionally Linear Decomposition and Synthesis (FLDS). It uses Gaussian Elimination to expose XOR relationships within a logic function, thereby fundamentally breaking away from the approach of Ashenhurst and Curtis. FLDS is still able to utilize BDDs to efficiently represent logic functions, making it fast and scalable. It was tested on a set of 99 MCNC benchmarks, where 25 of the benchmarks have been classified by previous researchers as XOR-based logic circuits, while the remaining 74 are classified as mostly AND/OR logic. In comparison to the leading logic synthesis tools, ABC [ABC05] from the University of California at Berkeley and BDS-PGA 2.0 [Vemuri01] from the University of Massachusetts, FLDS produces XOR-based circuit with 25.3% and 18.8% smaller area, respectively. The logic circuit depth is also improved by 7.68% and 14.46% respectively. On AND/OR logic circuits FLDS produced results merely 6.2% and 4.8% larger than it's respective competitors.

IV.1 Introduction

The problem of logic synthesis has been approached from various angles over the last 50 years [Perkowski94]. For practical purposes it is usually addressed by focusing on either AND/OR, multiplexor, or XOR-based logic functions. Each of the above types of logic functions exhibits different properties and usually a synthesis method that addresses one of these types very well, does not work well for the others.

The XOR-based logic functions are an important type of functions as they are heavily used in arithmetic, error correcting and telecommunication circuits. It is challenging to synthesize them efficiently as there are 7 different classes of XOR logic functions [Sasao99], each of which has distinct approaches that work well to implement logic functions of the given class. In this work we focus on XOR-based logic functions and show that they exhibit a property that can be exploited in an area-driven CAD flow.

A wealth of research on XOR logic decomposition and synthesis exists. Some of the early work was geared toward spectral decomposition [Hurst85], in which transforms are applied to a logic function in order to change it into a different domain, where analysis could be easier. It is analogous to how real time signals are analyzed in the frequency domain using tools like the Fourier Transform.

While at the time the methods were exceedingly time consuming, the advent of Binary Decision Diagrams accelerated the computation process and showed practical gains for using spectral methods [Clarke93].

In the late 1970s Karpovsky presented an approach called linear decomposition [Karpovsky77] that decomposed a logic function into two types of blocks - linear and non-linear. A linear block was a function consisting purely of XOR gates, while a non-linear block represented functions that require other gates to be completely described (AND, OR, NOT). This approach was found to be effective for arithmetic, error correcting and symmetric functions.

More recently, XOR based decompositions were addressed using Davio expansions [Tsai96], and with the help of BDDs [Yang00, Vemuri02]. With Davio expansions, Reed-Muller logic equation that utilizes XOR gates can be generated for a function,. On the other hand, the idea behind using BDDs was to look for x-dominators in a BDD that would indicate a presence of an XOR gate and could be used to reduce the area taken by a logic function. Also, tabular methods based on the work of Ashenhurst and Curtis [Ashenhurst59, Curtis62, Perkowski94] have been used to perform XOR-based decomposition [Wan92].

The work presented here addresses two limitations of the above methods. First, Davio expansions perform decomposition one variable at a time so an XOR relationship between non-XOR functions may not necessarily be found. Second, BDD based methods either use x-dominators to perform a non-disjoint decomposition or rely on BDD partitioning, which in essence clings to the idea of column multiplicity introduced by Ashenhurst and Curtis [Ashenhurst59, Curtis62, Perkowski94]. The latter is quite inadequate for XOR-based logic synthesis as column multiplicity does not address the XOR relationship between the columns of a truth table.

In this work we introduce a novel approach that is based on the property of linearity. Rather than looking at linear blocks as in [Karpovsky77], x-dominators or partitioning a BDD, this approach exploits a linear relationship between logic sub-functions. Thus we define *functional linearity* to describe a decomposition of a boolean function into the following form:

$$f(X) = \sum_i G_i(Y)H_i(X - Y) \quad \text{(IV-1)}$$

where X and Y are sets of variables such that $Y \subseteq X$, while the summation represents an XOR gate.

In this representation the function f is a weighted sum of functions G_i , where the weighting factors are defined by functions H_i . This approach retains the ability to synthesize XOR logic functions using Davio and Shannon's expansions [Sasao99], and retains the ability of a BDD to find relationships between logic functions of many variables, while fundamentally breaking away from the concept of column multiplicity.

The logic synthesis method proposed in this chapter utilizes Gauss-Jordan Elimination to decompose a logic function into linearly independent sets of functions, and synthesizes the original function by a linear combination of these functions, as in Equation 1. Results show that the approach presented here in comparison to the state-of-the-art synthesis tools produces better results on average for XOR-based logic circuits.

The remainder of this chapter is organized as follows: Section 2 contains the background information for the new synthesis approach. Section 3 details the approach for single output logic synthesis and Section 4 presents the variable partitioning algorithm used in our technique. Section 5 discusses an algorithm to further reduce the area of a logic network. Section 6 extends the approach to multi-output logic synthesis, while Section 7 discusses the nuances of the proposed logic synthesis approach and highlights key points required for reducing synthesis time. Section 8 addresses FPGA specific synthesis considerations and in Section 9 the differences between this work and prior research are presented. Section 10 details the results obtained using this method and compares them to ABC [ABC05] as well as BDS-PGA 2.0 [Vemuri02].

IV.2 Background

In this section we focus on the description of basic concepts of logic synthesis and linear algebra used in this chapter.

IV.2.1 Logic Synthesis

In the last 50 years numerous approaches to logic synthesis have emerged, which generally fall into one of the following three categories:

- tabular methods
- algebraic methods

- BDD-based methods

The tabular methods are based on the work of Ashenurst and Curtis [Ashenurst59, Curtis62, Perkowski94] in which a logic function is represented as a truth table, where both rows and columns of the table are associated with a particular valuation of input variables. Ashenurst and Curtis chose to represent a logic function in a tabular form so that what is referred to as *column multiplicity* could be exploited. This concept is illustrated on an example in Figure IV-2.

<i>de</i> \ <i>abc</i>	000	001	010	011	100	101	110	111
00	0	0	0	0	0	0	0	0
01	0	0	0	0	1	1	1	1
10	0	1	0	1	0	0	0	0
11	1	1	1	1	1	1	0	0

Figure IV-2: Column multiplicity example

In this example the truth table consists of 8 columns. The columns are indexed by variables *abc*, called the free set, and the rows are indexed by variables *de*, called the bound set. A closer look at the table reveals that there are 4 unique columns, each of which is used multiple times (2) in the table; hence the term *column multiplicity*. An easy way to realize this function is to synthesize each unique column and AND it with a function to specify in which columns it is used. For example, columns 000 and 010 are identical and implement the function $h_1 = de$. The function that describes in which columns h_1 is used is $g_1 = \bar{a} \cdot \bar{c}$. If we process every unique column this way, producing functions h_i and g_i , we can synthesize the original function as

$$\bigcup_{i=1}^4 h_i g_i$$

Clearly, the smaller the number of distinct columns the easier it is to synthesize a logic function. An example of such methods is shown in [Wan92].

Algebraic methods extract a subfunction of a logic expression by processing a logic equation for a given function. The basic idea is to apply tools such as the Shannon's expansion, the

DeMorgan's Theorem or the Davio expansions to reduce the complexity of a logic expression. By applying these and other logic reduction techniques an equation can be simplified and thus implemented using fewer gates. Representative samples of such approaches are [Sasao94] and [Tsai96].

Finally, BDD-based methods use a Binary Decision Diagram [Bryant86] to represent a logic function and manipulate it to reduce the area a logic function occupies. As the size of the BDD and the area occupied by a logic function are related to one another, this approach can be effective in reducing area of logic functions. In the context of XOR synthesis BDD-based approaches utilize the concept of x -dominators to determine if XOR gates are present in the logic function. An x -dominator in a BDD is a node to which a regular and a complemented edge point to. Intuitively, an x -dominator points to a node such that the function it implements can be either zero or one for the same input valuation and depends on the state of the variables above it. Thus, an XOR relationship is exposed. Examples of BDD-based approaches are [Lai93], [Yang99], [Yang00] and [Vemuri02].

IV.2.2 Linear Algebra

The topic of this section concerns fields, vector spaces and Gauss-Jordan Elimination. We review these topics in detail.

In mathematics a field F is defined as a collection of symbols S as well as summation (+) and multiplication (*) operators and constants 0 and 1 that satisfy the following five axioms:

- + and * are associative
- + and * are commutative
- Multiplication distributes over addition
- 0 is an additive identity, while 1 is a multiplicative identity ($0 \neq 1$)
- Additive and multiplicative inverses exist

It is also important to note that the result of any operation on elements of F must produce an element that is also in F . Using elements of a field F as entries in an n -tuple we can form vectors that satisfy the properties of vector spaces. Namely, we can scale and add vectors together and the resulting vector will also have entries that belong to the field F . A collection of such vectors would then be said to form a vector space.

A very important property of a vector space is that it is spanned by a subset of the vectors it contains. We illustrate this on an example. Consider the following 5 vectors:

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 2 \\ 3 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 6 \\ 5 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix}$$

Notice that the third entry is 0 for each vector. Further inspection reveals that each vector in the set is a weighted sum of the two leftmost vectors. We therefore say that the two leftmost vectors are a *basis* for the vector space consisting of the given set of vectors. This is a very powerful idea as, among other applications, it clearly identifies basic components that form a vector space.

While in the above example the set of basis vectors was easy to find, a general procedure to find a basis for any vector space exists. Suppose that we take six vectors and represent each vector as a column of a matrix [Anton94]:

$$\begin{bmatrix} 1 & -3 & 4 & -2 & 5 & 4 \\ 2 & -6 & 9 & -1 & 8 & 2 \\ 2 & -6 & 9 & -1 & 9 & 7 \\ -1 & 3 & -4 & 2 & -5 & -4 \end{bmatrix}$$

To find a basis for the columns of the matrix we must first apply Gaussian Elimination to the matrix.

Gaussian Elimination is a process of applying elementary row operations (addition, subtraction and scalar multiplication) to the rows of the matrix in order to reduce the matrix to an upper triangular matrix with left-most elements set to 1. To do this, we first pick a row with the non-zero entry in the left most column. In our example, let us pick the top row. We then look through all other rows that have a non-zero entry in the same column and try to make their entries in the same column a zero. To do that we subtract a scalar multiple of the top row from a scalar multiple of the other rows. For example, the entry in the first column of the second row can be changed to a zero by subtracting twice the first row from the second. By following this procedure we reduce the matrix to a row-echelon form [Anton94]:

$$\begin{bmatrix} 1 & -3 & 4 & -2 & 5 & 4 \\ 0 & 0 & 1 & 3 & -2 & -6 \\ 0 & 0 & 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

To obtain the basis for the column vectors we now look at each non-zero row of the resulting matrix. For each of those rows the column with the leading 1 corresponds to the column in the original matrix that is a basis vector for the space spanned by the column vectors [Anton94]. In our case these are columns one, three and five, and therefore the following are the basis vectors for the column vector space:

$$\begin{bmatrix} 1 \\ 2 \\ 2 \\ -1 \end{bmatrix} \quad \begin{bmatrix} 4 \\ 9 \\ 9 \\ -4 \end{bmatrix} \quad \begin{bmatrix} 5 \\ 8 \\ 9 \\ -5 \end{bmatrix}$$

Notice that each column in the original matrix can be represented as a weighted sum of the above basis vectors.

Gaussian Elimination is sometimes carried out one step further. That is, in addition to reducing the matrix to a row-echelon form, we further reduce the matrix so that the leading 1 of each row is the only non-zero entry in the corresponding column. When the procedure is carried out to this point it is referred to as Gauss-Jordan Elimination and the resulting matrix is said to be in the reduced row-echelon form. This is useful when matrices are used for solving a system of linear equations.

For the purpose of logic synthesis of digital circuits the mathematical operations performed here are in a modulus field, also called a Galois Field. A Galois Field with modulus m is denoted as $GF(m)$. In particular, we are interested in $GF(2)$, because it facilitates operations on Boolean logic functions. In this field the additive operator is the XOR operation, while a logical AND is equivalent to multiplication.

IV.2.3 Notation

In the previous subsections we introduced background information, however some notation

and terms used in logic synthesis may appear to conflict with the well established norms in linear algebra. Thus, we explicitly state the notation used throughout the remainder of this chapter.

In this chapter the matrix notation is used with the assumption that all operations are performed in $GF(2)$. Thus, summation is equivalent to an XOR operation and multiplication is equivalent to a logical AND operation. We use \oplus to represent a sum and a $+$ to represent a logical OR operation. In addition we will use an \uparrow to represent a NAND operation and \odot to denote an XNOR.

A matrix is denoted by a capital letter, while a column vector is identified by a lower case bold letter. For example, $\mathbf{Ax}=\mathbf{b}$ indicates an equation where a matrix \mathbf{A} is multiplied by a column vector \mathbf{x} , resulting in a column vector \mathbf{b} . To distinguish regular column vectors from basis vectors, we denote basis vectors with capital bold letters. To distinguish input variables of a logic function from matrices and vectors, input variables of a logic function, as well as the function output, are italicized (eg. $f=abcd$).

In this chapter linear algebra is used in the context of logic synthesis and hence we often represent columns/rows of a matrix as a logic function, rather than as column vectors of 0s and 1s. For example, if a column has 4 rows, indexed by variables a and b , then we index the rows from top to bottom as 00, 01, 10 and 11. Thus, a statement $\mathbf{X}=a$ indicates that the basis vector \mathbf{X} is $[0\ 0\ 1\ 1]^T$, whereas $\mathbf{Y}=b$ indicates a basis vector $[0\ 1\ 0\ 1]^T$.

IV.3 Functionally Linear Decomposition and Synthesis

Functionally Linear Decomposition and Synthesis (FLDS) is an approach that exploits an XOR relationship between logic functions. It gets its name from the way an XOR relationship is derived, by using the basis of the column space of a truth table to decompose a function into a set of sub-functions.

The basic idea behind this method is to find sub-functions that can be reused in a logic expression and thus reduce the size of a logic function implementation. In essence it is analogous to boolean and algebraic division, or kernel extraction. In these methods we first derive a divisor, or a kernel, and then decompose a logic function with respect to it. In FLDS we seek to achieve a similar effect, however we take advantage of linear algebra to do it. In our approach a basis vector

is analogous to a "divisor" and a selector function to specify where a basis vector is used is analogous to a "quotient" from algebraic division. The key advantages of our method are: a) we can compute basis and selector functions simultaneously, b) the initial synthesis result can be easily optimized using linear algebra, and c) the method is computationally inexpensive even for large functions. In this section we explain the basics of our approach and then present methods to refine the synthesis results in subsequent sections.

		<i>ab</i>			
	<i>cd</i>	00	01	10	11
	00	0	0	0	0
	01	0	0	1	1
	10	0	1	0	1
	11	0	1	1	0

Figure IV-3: Truth Table for Example 1

Consider a logic function represented by a truth table in Figure IV-3. The figure shows a truth table for logic function $f=ad\oplus bc$, with variables ab at the top and variables cd on the left hand side. We will first decompose this function and then synthesize it.

The first step is to find the basis vectors for the truth table shown in Figure IV-3. To do this, we will apply Gaussian Elimination to the above truth table, as though the truth table was a regular matrix in GF(2). The procedure as it is applied to this truth table is illustrated in Figure IV-4.

We begin with the initial matrix that directly represents the truth table and swap the rows such that the rows are ordered from top to bottom based on the column index of their respective leading one entries. The next step is to perform elementary row operations to reduce the matrix to the row-echelon form. Thus, we replace row 1 with the sum (XOR in GF(2)) of rows 0 and 1. This causes row 0 to be the only row with a 1 in the second column. Finally, we replace row 2 with the sum of rows 1 and 2, making row 2 consist of all zeroes.

In the resulting matrix the first two rows have leading ones in the middle two columns. From linear algebra we know this to indicate that the middle two columns in the original truth table, or equivalently columns $ab=01$ and $ab=10$, are the basis vectors for this function. Therefore, this function has two basis vectors, $\mathbf{G}_1=c$ and $\mathbf{G}_2=d$.

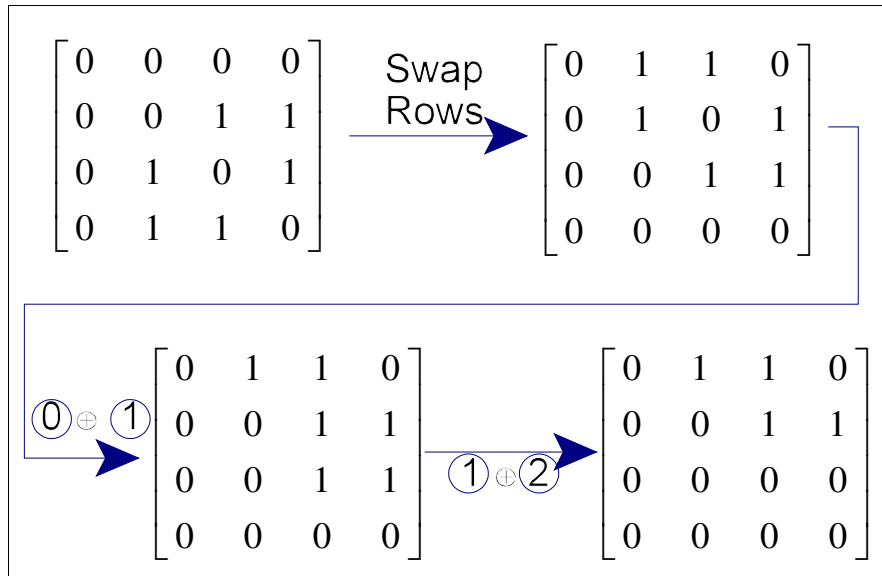


Figure IV-4: Gaussian Elimination applied to Example 1

The next step is to express the entire truth table as a linear combination of these two vectors, which means expressing each column C_i as $h_{1i}\mathbf{G}_1 \oplus h_{2i}\mathbf{G}_2$, where h_1 and h_2 are constants. To find h_{1i} and h_{2i} we solve a linear equation $\mathbf{A}\mathbf{x}=\mathbf{b}$, where \mathbf{A} is a matrix formed by basis vectors \mathbf{G}_1 and \mathbf{G}_2 , $\mathbf{x}=[h_{1i} \ h_{2i}]^T$ and \mathbf{b} is the column we wish to express as a linear combination of basis vectors \mathbf{G}_1 and \mathbf{G}_2 . For example, to express column C_3 we solve the following equation:

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} h_{1i} \\ h_{2i} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad (\text{IV-2})$$

By inspection the solution to this equation is $h_{1i} = 1$ and $h_{2i} = 1$.

We now need to form selector functions that will identify the columns in which a given basis vector appears. To create a selector function for basis vector \mathbf{G}_1 we look at the columns for which h_{1i} was found to be 1. These columns are $ab=01$ and $ab=11$. Thus, the selector function $H_1=b$. Similarly, the selector function for \mathbf{G}_2 is found to be $H_2=a$. Finally, we can synthesize the function f as $H_1\mathbf{G}_1 \oplus H_2\mathbf{G}_2$, producing logic equation $f=bc \oplus ad$. The resulting circuit is shown in Figure IV-5.

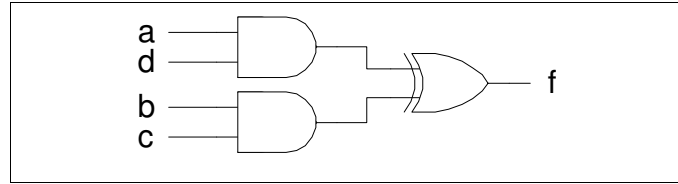


Figure IV-5: Circuit synthesized for Example 1

IV.4 Heuristic Variable Partitioning

The success of a logic synthesis approach depends heavily on how variables are chosen during logic decomposition. In fact, by varying the assignment of variables to rows and columns of a truth table we can create a table with varying number of basis vectors, costs of basis and selector functions, or both. The problem then becomes, as with most synthesis approaches, how to order, or in our case partition, variables. The problem can be demonstrated on the following example.

		<i>ab</i>			
		00	01	10	11
<i>cd</i>	00	1	0	0	0
	01	0	1	0	0
	10	0	0	1	0
	11	0	0	0	1

Figure IV-6: Truth Table for Example 2

Consider the function $f=(a \odot c)(b \odot d)$, with variables a and b indexing the columns and c and d indexing the rows, as shown in Figure IV-6. By following the procedure outlined in Section 3, we find four basis vectors and synthesize it as the XOR of four minterms. To synthesize the function using XOR gates efficiently, we need to rearrange variables in the truth table such that rows, or columns, of the truth table are indexed by variables a and c . Rearranging the variables this way will show that only one basis vector $(a \odot c)$ is present with a selector $(b \odot d)$, and the resulting function will be expressed as $f=(a \odot c)(b \odot d)$.

While many variable ordering approaches exist most of them are designed for tabular methods or for BDDs. None of these directly expose basis functions of a logic expression and are

thus inadequate for our purposes. Hence, we introduce a new heuristic variable partitioning algorithm that takes advantage of functional linearity. The algorithm is presented in Figure IV-7. It takes as input a logic function f of n variables and the number of variables to be assigned to index rows m . It produces a variable partitioning that is suitable for linear decomposition by returning the bound set.

The algorithm works by incrementally partitioning variables into groups that have 2^k variables, where $2^k < m$. During the first iteration the algorithm partitions variables into $n/2$ groups of two variables, such that each variable appears in at most one group. Each combination of variables is created and evaluated by determining the number of basis functions created if the given set is used to index rows as well as computing the cost of basis and selector functions. The cost of a function, basis or selector, is determined by the size of its support plus one, unless the function evaluates to zero in which case the cost is equal to zero.

Once each grouping of two variables is evaluated, the algorithm retains only $n/2$ least cost groupings, keeping in mind that each variable may only appear in one grouping. The procedure then repeats in exactly the same fashion for groupings of four variables, except that the groupings are formed by putting together two groupings of two variables.

The above procedure repeats, doubling the grouping size at each iteration, until the grouping

```

Partition_Variables(f, n, m)
{
  Determine all possible partitions of n variables such that the bound set size is 2. Pick
  n/2 best groupings such that each variable appears in at most one grouping.
  for (k=4; k < m; k=k*2)
  {
    Using previously created groupings of k/2 variables, create groupings with k
    variables. Keep the best n/k groupings.
  }
  If m is not a power of 2 then
  {
    Use the generated groupings to form bound sets of m variables and pick the best
    one.
  }
  Reorder variables in f to match the best grouping of size m.
}

```

Figure IV-7: Heuristic variable partitioning algorithm

size exceeds m . In a case where m is not a power of 2 the algorithm proceeds to evaluate all partitions of size m by putting together input variables and groupings generated thus far. For example, if $m=5$ then for each grouping of size 4, the algorithm combines it with an input variable to make a grouping of 5 variables. The algorithm then evaluates the grouping and keeps it if and only if it has lower cost than any other grouping of size 5.

The algorithm presented here is a heuristic and as such does not explore the entire solution space in the interest of reducing computation time. To evaluate the effectiveness of this heuristic we compared the area results obtained by FLDS when using the above algorithm, to an exhaustive search and a random variable partitioning. The exhaustive search proved to generate the best results, while a random partitioning provided on average the worst results. Our proposed variable partitioning algorithm proved to be a close second to the exhaustive search for XOR-based logic circuit, with a distinct advantage in runtime.

IV.5 Basis and Selector Optimization

In the previous section we described the variable partitioning algorithm that determines which variables should belong to the bound set and which to the free set. However, a proper variable assignment is not always sufficient to optimize area of a logic function. This is because for any given variable partitioning there can exist many sets of basis vectors. While each set will have the same size, the cost of implementing each set of basis functions, and their corresponding selector functions, may be different.

To illustrate this point consider the example in Figure IV-8. This is the same function as in Example 1, but this time the variables are partitioned differently. Despite that, we can still synthesize this function optimally. Instead of reordering variables we can optimize basis and selector functions.

In this particular example, we have two basis vectors: $\mathbf{G}_1=bc$ and $\mathbf{G}_2=b\uparrow c$. The selector functions corresponding to these basis functions are $H_1=a\uparrow d$ and $H_2=ad$. It is possible to replace one of the basis functions with $\mathbf{G}_1\oplus\mathbf{G}_2$ and still have a valid basis function. Notice that $\mathbf{G}_1\oplus\mathbf{G}_2=\mathbf{1}$ and thus has a smaller cost than \mathbf{G}_2 itself. It is therefore better to use $\mathbf{G}_1\oplus\mathbf{G}_2$ and \mathbf{G}_1 in order to represent the function in Figure IV-8. By inspection, the function f now has basis functions $\mathbf{G}_2'=\mathbf{G}_1\oplus\mathbf{G}_2=\mathbf{1}$ and $\mathbf{G}_1=bc$. The corresponding selector functions are now $H_2'=ad$ and $H_1=1$. This is because the fourth

		<i>ad</i>			
	<i>bc</i>	00	01	10	11
00		0	0	0	1
01		0	0	0	1
10		0	0	0	1
11		1	1	1	0

Figure IV-8: Truth Table for Example 3

column of the truth table can now be represented as $\mathbf{1} \oplus \mathbf{G}_1 = \mathbf{G}_2$. We can now synthesize function f as $f = (H_1 \mathbf{G}_1) \oplus (\mathbf{G}_2' H_2') = (bc(1)) \oplus (ad(1)) = ad \oplus bc$. We obtain the same result as in Section 3.

The basis replacement idea is therefore rather simple - keep the set of basis functions linearly independent of one another, while reducing the cost of basis and selector functions in the process. Clearly the basis function pairs $\{\mathbf{G}_1, \mathbf{G}_2\}$, $\{\mathbf{G}_1, \mathbf{G}_2'\}$, $\{\mathbf{G}_2', \mathbf{G}_2\}$ are all valid sets of basis functions, as neither element can be represented as a linear combination of the others. Also, each set is capable of expressing each column in the original matrix. The only consideration left is how the basis replacement affects the corresponding selector functions.

A careful analysis reveals that a basis function \mathbf{G}_2 can be replaced by $\mathbf{G}_1 \oplus \mathbf{G}_2$, but it requires the selector function for basis \mathbf{G}_1 to be replaced with $H_1 \oplus H_2$. This is because we must now use basis function \mathbf{G}_1 wherever \mathbf{G}_2 was originally by itself. In addition, wherever \mathbf{G}_1 and \mathbf{G}_2 were both used, now only the new basis function $\mathbf{G}_2' = \mathbf{G}_1 \oplus \mathbf{G}_2$ is used and hence basis function \mathbf{G}_1 should not be used.

Based on the above analysis Figure IV-9 presents the basis-selector optimization algorithm. The algorithm takes as input the set of basis functions and their corresponding selector functions and returns a modified set of basis and selector functions. The algorithm works as follows.

First, the algorithm computes the cost of each basis and selector function. The cost is determined by the support set size of each function, plus 1, with the exception of a function equal to 0, for which the cost is 0. The algorithm then proceeds to evaluate possible improvements to basis and selector functions. This is done by putting together a pair of basis-selector function pairs to determine if the resulting basis and selector function can replace one of the existing ones and reduce the basis/selector cost. If replacing an existing basis with a new one does not increase the overall cost of implementing a logic function, then the algorithm replaces one of the basis and selector functions.

```

Optimize_Basis_and_Selectors(basis_fs, selector_fs)
{
  Cost = sum up the costs of all basis functions and their respective selector functions
  repeat {
    For (i=0; i<m-1;i++)
      For (j=i+1; j<m;j++)
        New basis = basis(j) XOR basis (i)
        New selector = selector (j) XOR selector (i)
        Replace_i = cost of replacing basis(i) and selector(j)
        Replace_j = cost of replacing basis(j) and selector(i)
        Pick the least costly of replace_i and replace_j and if the cost of changing
          the basis/selector function is no more costly than current implementation
          then replace the basis and the selector functions
        Update cost
    } until (last 3 iterations produced the same cost)
}

```

Figure IV-9: Basis-Selector optimization algorithm

Otherwise, the algorithm attempts to combine a different set of basis and selector functions.

The algorithm will iterate until the last 3 iterations produce the same basis-selector cost, which indicates that further improvement is unlikely to be found. Note that the algorithm allows for basis-selector replacement even if the cost is unchanged, because it may sometimes be necessary to go through a few intermediate steps before a lower cost basis-selector pair is found.

When this algorithm is applied to the truth table in Figure IV-8, it will find that using basis function $\mathbf{G}_1 \oplus \mathbf{G}_2 = \mathbf{1}$, in place of \mathbf{G}_1 or \mathbf{G}_2 is preferable. An important point to notice in this example is that an all 1s column used does not appear in the truth table in Figure IV-8. It illustrates one of the key distinctions between this method and the classical approach of Ashenurst and Curtis [Ashenurst59, Curtis62, Perkowski94].

IV.6 Multi-Output Synthesis with FLDS

The decomposition and synthesis procedure outlined in Section 3 addresses the issue of synthesizing a single output logic function. While it illustrates how the approach generates a 2-level logic network, it is important to extend a synthesis technique to generate multi-level networks as well as the synthesis of logic functions with multiple outputs.

This section discusses two aspects of multi-level and multi-output function synthesis with

FLDS. First, it shows how the function decomposition of a single output function may result in a multiple output logic function synthesis. It then follows to discuss how different approaches to multiple output function decomposition work in the context of FLDS. Finally, a multi-output synthesis algorithm implemented in FLDS is presented.

IV.6.1 Multi-Level Decomposition

A multi-level decomposition of a logic function is needed when the final logic expression requires more than 2 gates on the longest input to output path. In the context of FPGAs the depth metric is the number of LUTs on the longest input to output path. When a logic function has more than k inputs, it may be necessary for more than 2 k -LUTs to be created on an input to output path to correctly synthesize a function.

With FLDS there are two ways in which this situation can arise. One case is when the number of basis and selector functions exceeds $k/2$, in which case AND/XOR logic created to combine them will require more than one level of LUTs to implement. This is not an issue as the synthesis technique has already addressed this by creating the AND/XOR logic and a function will synthesize.

The second case is more interesting, and that is when either the basis or the selector function have more than k inputs. Such a case requires further decomposition. It is important to notice though that in such a case the variables for the basis functions come from a shared set (the selector functions also share variables), which presents an opportunity for further area reduction by extracting their common sub-functions.

IV.6.2 Multi-Output Synthesis

With FLDS a set of functions that share a subset of variables can be synthesized rather easily. A way to do this is to select a subset of common variables as the bound set and put the truth tables for all functions in a single table. We describe this procedure on the following example.

Consider two logic functions, f and g , both with 4 input variables. Both functions share variables a , b and c , while inputs d and e are the fourth input to functions f and g respectively. For this example we chose to put variables a and b on the rows of the truth table, as shown in Figure IV-10. If these two functions were to be synthesized separately, function f would be found to have 2

		<i>f</i>				<i>g</i>			
		<i>cd</i>				<i>ce</i>			
<i>ab</i>		00	01	10	11	00	01	10	11
00		0	0	0	0	1	1	1	1
01		0	1	0	1	0	1	0	1
10		0	1	0	1	0	1	0	1
11		0	1	1	0	0	0	1	1

Figure IV-10: Multi-output synthesis example

basis vectors, while function *g* would have 3 basis vectors. However, when synthesized in tandem, there are only 3 basis vectors (2nd, 3rd and 5th column) for the two functions put together.

We can now apply the same steps as for single output function synthesis to further optimize the logic function. In this case we can use basis and selector optimization algorithm to determine that a simpler function implementation can be found when an all 1s column is used instead of the 5th column.

The function *f* is synthesized as $f=(a+b)d\oplus abc$ and shares its basis functions with *g*. For function *g* we find that the basis function from the 2nd column is used in column $ce=10$, the basis function from the 3rd column is used in columns $ce=\{01, 10\}$, while the all 1s column is used in every column of function *g*. Once we synthesize each basis function and its selector function, we can see that the synthesis of the all 1s column and its selector function simplify to a constant 1. A

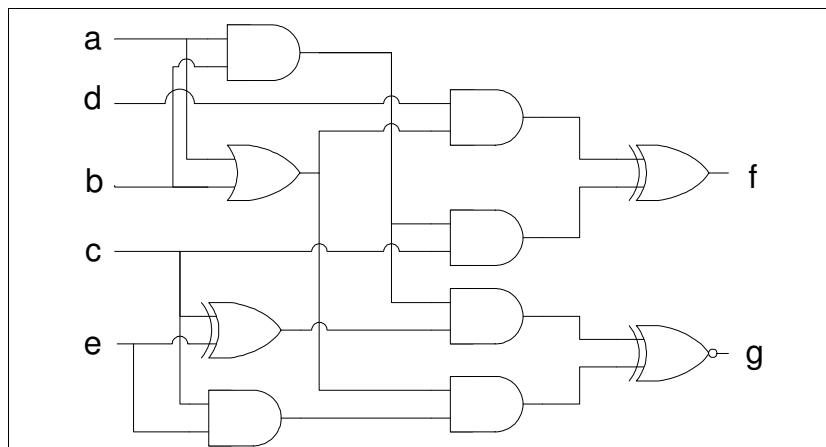


Figure IV-11: Synthesized functions *f* and *g*

constant 1 input to a 3-input XOR gate reduces it to a 2-input XNOR gate. The resulting circuit is shown in Figure IV-11.

The above example showed how FLDS can be used to decompose a pair of functions that share variables in order to reduce their total area. While in some cases choosing a subset of shared variables to index the rows is desirable, it is not always the case. It is also possible to use unique variables from one function to index the rows to get better results. Using this approach we can synthesize arithmetic functions, like a ripple carry adder, efficiently. To demonstrate this, consider the example of a 2-bit ripple carry adder.

	C_{out}				S_1				S_0			
	x_0	y_0	x_0	y_0	x_0	y_0	x_0	y_0	x_0	y_0	x_0	y_0
$x_1 y_1$	00	01	10	11	00	01	10	11	00	01	10	11
00	0	0	0	0	0	0	0	1	0	1	1	0
01	0	0	0	1	1	1	1	0	0	1	1	0
10	0	0	0	1	1	1	1	0	0	1	1	0
11	1	1	1	1	0	0	0	1	0	1	1	0

Figure IV-12: 2-bit ripple carry adder example

Figure IV-12 shows a truth table for the carry out and 2-bit sum output of a 2-bit ripple carry adder. The ripple carry adder has inputs $x_1, x_0, y_1,$ and y_0 , where x_1 and y_1 are the most significant bits of inputs x and y . Notice that in this example the rows are indexed by x_1 and y_1 , in contrast to the example in Section 6.2 where the common variables were used to index the rows. In this case the idea is to remove the unique variables from each functions and be left only with sub-functions that have the same support.

By applying the FLDS synthesis procedure from Section 3 we find basis functions $\mathbf{G}_1=x_1 y_1$, $\mathbf{G}_2=x_1+y_1$, and $\mathbf{G}_3=x_1 \oplus y_1$. We can optimize basis and selector functions with the procedure from Section 5 and find that we can replace \mathbf{G}_2 with $\mathbf{G}_1 \oplus \mathbf{G}_2$, forming $\mathbf{G}_2'=x_1 \oplus y_1$, and replace \mathbf{G}_3 with $\mathbf{G}_2' \oplus \mathbf{G}_3$, forming $\mathbf{G}_3'=1$. Using these basis functions we can synthesize C_{out} and S_1 as follows:

$$C_{out} = x_1 y_1 \oplus [(x_1 \oplus y_1) x_0 y_0]$$

$$S_1 = (x_1 \oplus y_1) \oplus x_0 y_0$$

If we rename the selector function x_0y_0 as C_{in} , then the circuit we obtain is a familiar full adder, as shown in Figure IV-13. The only step left to complete is to synthesize C_{in} and S_0 , which will synthesize as a familiar half adder. This example can be extended to an n-bit ripple carry adder.

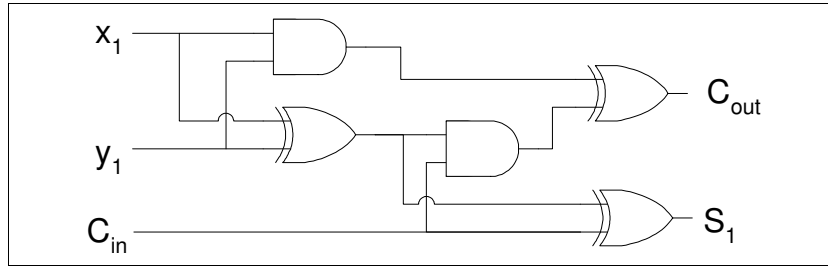


Figure IV-13: Full adder synthesized using FLDS

IV.6.3 Multi-output Synthesis Algorithm

Examples in Section 6.2 show that it is important to carefully consider which variables are assigned to the rows of the decomposition matrix when applying the FLDS algorithm. Based on these examples we present a generalized multi-output synthesis algorithm.

The algorithm initially considers a decomposition for multi-output functions by using least common variables of a set of functions for indexing the rows of the truth table. By doing so we can reduce the original set of functions to a set that depends on a common set of variables. Once the set of functions has been reduced to one where their support is common to all functions, we can then use the shared variables to index the rows of a matrix. The algorithm is presented in Figure IV-14.

The first step in the algorithm is to determine if the functions to be synthesized share all variables, or just a subset. If all variables are shared then we can perform a decomposition where the subset of shared variables are used to index the truth table rows. The decomposition with respect to shared variables will try to do a balanced decomposition so long as the number of basis functions is low in an effort to reduce logic circuit depth. If the number of basis functions becomes large, the number of variables indexing the rows will be reduced to decrease the number of basis functions.

The second step of the algorithm is to perform a decomposition in the case where some variables are shared by all functions and some are not. We thus create a list \mathbf{L} of these functions and perform the next set of steps while the list has functions left to be synthesized. The list will shrink and grow as we synthesize functions, either because of the decomposition, or because some of the

```

Multi_Output_Synthesis(function_set, k)
{
  Create a list of common variables and a list of all variables used by the function set
  if (#common variables == #all variables) then
    Assign Common Variables to rows and perform decomposition
  else
    {
      Create a list of functions L left for synthesis
      while (L is not empty)
        {
          Synthesize functions with fewer than  $k$  input variables
          Find set of variables S that are used in fewest functions
          Find a set of functions F that use variables in S and remove this set of
            functions from L
          If (L is empty) then
            Do Decomposition with shared variables for F
          Else
            {
              Synthesize basis functions for set F
              Put the selector functions generated in the previous step into L
            }
        }
    }
}

```

Figure IV-14: Multi-output Synthesis algorithm

functions will be reduced to have fewer than k input variables, synthesized into a k -LUT and removed from the list.

The next set of steps focuses on finding a subset of variables **S** that are used by the fewest number of functions in the set **L**. We take these functions out of the set **L** and synthesize their basis functions with respect to the input variables in **S**. The basis functions will be synthesized separately, while selector functions will be put back into the set **L**. This is because selector functions can potentially share variables with functions in set **L** and some logic sharing may be possible.

An exception to the process described above is the case where all functions in the set **L** end up sharing all variables. In this case, the algorithm has no choice but to pick variables to index the rows of a decomposition matrix from a set of variables common to all functions. This is the case from the first example in Section 6.2.

The motivation behind the above algorithm is to decompose a set of logic functions in such a way as to identify the subexpressions that share the same support. This forces the algorithm to first find the subset of functions that heavily depend on one another, and then decompose them in as area efficient manner as it is able.

IV.7 Performance Considerations

The previous sections demonstrated how FLDS can synthesize logic functions using XOR relationship between logic sub-functions. While the approach itself is relatively straightforward, its usefulness in Computer Aided Design tools depends significantly on how fast the synthesis process can be performed. Clearly, storing a truth table for a logic function as an array of bits will become problematic past 20 variables as it will take time to fill the table with 0s and 1s and the array itself will begin to occupy increasing amount of memory. In this section we discuss the issue of performance and show how the synthesis process can be accelerated by using BDDs and Gauss-Jordan Elimination.

Binary Decision Diagrams introduced by Bryant [Bryant86] are an efficient structure for storing and processing logic functions. In FLDS we have chosen to use BDDs to encode the truth table of a functions, specifically by encoding each row of the truth table as a BDD. While in extreme cases this may not work well, in most practical cases the BDD for each row will be small and easy to process. The package we used for this is the Cudd BDD package [Somenzi241].

The second method of accelerating the synthesis process is the use of Gauss-Jordan Elimination in place of Gaussian Elimination. Recall from Example 1 in Section IV.3 that the process of Gaussian Elimination was used to determine the basis functions. After finding basis functions we proceeded to determine the logic expression for the selector function of each basis vector. These two steps can be replaced by simply performing Gauss-Jordan Elimination.

In Figure IV-15 the matrix on the left is the matrix in the row-echelon form from Example 1. By applying the Gauss-Jordan Elimination the matrix is further reduced to a reduced row-echelon form shown on the right. In this form a column in which a leading 1 entry exists, contains only a single non-zero entry. By closer inspection we can see that the first row in the reduced row-echelon form matrix corresponds to the truth table of the selector function for the first basis vector. Similarly,

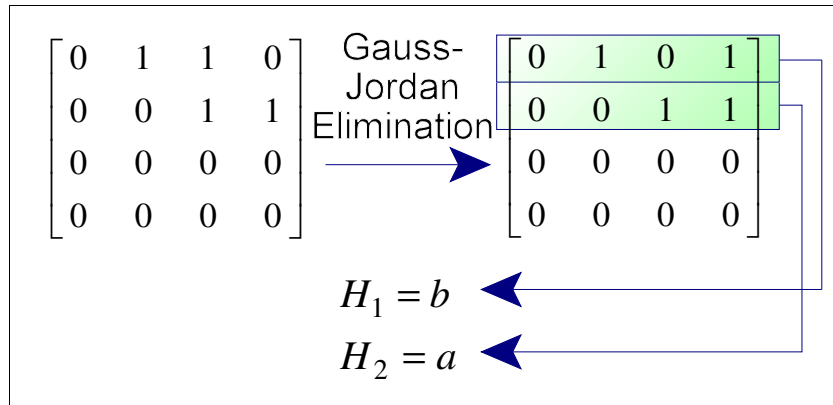


Figure IV-15: Gauss-Jordan Elimination applied to Example 1

the second row of this matrix corresponds to the selector function for the second basis vector.

Notice that in Section 3 we found the equations for H_1 and H_2 by solving linear equations for each column of the truth table. Gauss-Jordan Elimination solves this problem for us. This is a property of the Gauss-Jordan Elimination [Anton94], and while it is useful in solving linear equations, it is shown here that it is also important in the context of logic synthesis as the columns of the matrix are related by variables used to index them. Hence, the selector functions can be computed rapidly.

IV.8 FPGA-Specific Considerations

In the previous sections synthesis and processing time issues were addressed by using algebraic manipulation. The above need to be augmented by considering the target platform for synthesis, which in our case are FPGAs.

In particular, a key parameter is the size of a LUT, k , used by an FPGA. The idea here is that the decomposition process should take into consideration the amount of space basis and selector functions will take. In our implementation any function of k or fewer variables is synthesized into a single LUT. This presents an opportunity for area recovery as it is possible for the same function to be synthesized multiple times. This can either happen by synthesizing two functions in tandem, or one after another. As such, it is important to recognize repeated functionality and reuse it. For this purpose we use a hash table of logic functions of k and less inputs. Every time a LUT is synthesized it is put in a hash table referenced by input variables. If another LUT is to be created, we first check if the LUT function or its complement have already been synthesized. If so, rather than creating a

new LUT, a wire or an inverter connection is added to the logic graph thereby saving area.

Another consideration is that modern FPGAs contain more complex logic than simple LUTs and flip-flops. In fact, both Altera and Xilinx FPGAs contain carry chains to implement fast ripple carry adders. Because these adders contain XOR gates outside of the LUT network, it is possible to utilize these XOR gates to further reduce the area taken by a logic function. This however is a topic for future work.

IV.9 Contrast with Prior Work

FLDS resembles tabular methods based on the work of Ashenurst and Curtis [Ashenurst59][Curtis62][Perkowski94]. However, there are key differences between these methods and FLDS.

The tabular method looks for column multiplicity in a truth table, hoping to exploit it in an effort to reduce circuit area. In the example in Figure IV-2 we found four distinct columns using the tabular method, but by inspection we can see that FLDS finds 3 basis functions for this truth table. A second difference between the tabular method and FLDS was illustrated in Example 3 in Section 5. Specifically, we showed that during synthesis FLDS can utilize columns that do not actually appear in the truth table in order to reduce circuit area.

In a special case FLDS simplifies to the tabular method. An example of such a case is the synthesis of a wide AND gate (eg. $f=abcde$). In this particular case we find that the number of distinct columns and the number of basis vectors is the same. In more general terms, FLDS reduces to the tabular method if one of the following is true:

1. a product of all pairs of selector functions is 0
2. a product of all pairs of basis functions is 0

When one of the above is true, the XOR gates used to combine the product of selector and basis functions can be replaced by OR gates.

The above relationship between FLDS and the tabular method can be further exploited. As a matter of fact we know that XOR gates use more transistors than OR gates when transmission gates are not utilized. Thus, for fabrication purposes it is advantageous to replace XOR gates with OR gates if at all possible. In FLDS it is quite simple to determine if an XOR gate can be replaced by

an OR gate. If a product of a pair of basis functions, or their respective selector functions, is equal to 0, then the XOR gate used to sum the basis-selector products for this pair of sub-functions can be replaced by an OR gate. An example is given in Figure IV-16.

		<i>ab</i>			
	<i>cd</i>	00	01	10	11
	00	0	0	1	1
	01	0	1	1	0
	10	1	0	0	1
	11	1	1	0	0

Figure IV-16: XOR gate replacement example

Figure IV-16 shows a simple logic function with three basis functions. They are $G_1=c$, $G_2=d$ and $G_3=1$. It is reasonably easy to see that G_1 is used in columns $ab=00$ and 10 , G_2 is used in columns $ab=01$ and 11 , while G_3 is used in columns $ab=10$ and 11 . Using only two-input gates this function can be realized as shown in Figure IV-17a. Notice however that the selector functions for G_1 and G_2 are complements of one another, and thus their product is equal to 0. It is therefore unnecessary to sum these two sub-functions using an XOR gate. An OR gate can be used instead to implement the same logic expression. This is shown in Figure IV-17b.

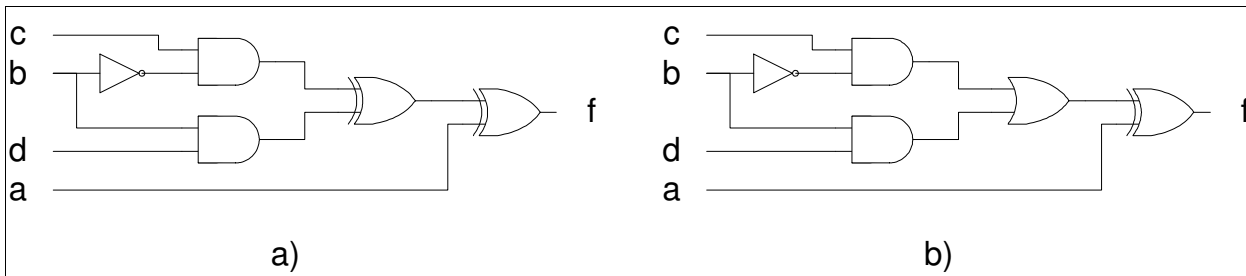


Figure IV-17: Example of XOR gate replacement

The tabular method is not the only one that can be recreated using FLDS. Positive and negative Davio expansions, useful for XOR decomposition are also a special case of FLDS. An easy way to derive them using FLDS is to assign a single variable, x , to the bound set. This will result in two basis functions and two selector functions, synthesizing the function f as follows:

$$f = xf_x \oplus \bar{x}f_{\bar{x}} \tag{IV-3}$$

By applying the basis and selector optimization algorithm from Section 5, one of the basis functions can be simplified to $x \oplus \bar{x} = 1$ resulting in one of the following equations:

$$f = f_x \oplus \bar{x}(f_x \oplus f_{\bar{x}}) \quad \text{(IV-4)}$$

$$f = (f_x \oplus f_{\bar{x}})x \oplus f_{\bar{x}} \quad \text{(IV-5)}$$

Equations 4 and 5 are the familiar negative and positive Davio Expansions, respectively.

The consequence of the ability to decompose a logic functions using equations IV-3, IV-4 and IV-5 using a cost function is the ability to synthesize exclusive OR sum of products (ESOP) expressions of various classes. In particular, if FLDS is set up to decompose a logic function one variable at a time, it will generate either a Pseudo Kronecker expression or Generalized Reed-Muller expression for a logic function [Sasao99]. The choice of the expression generated depends on the cost of implementing the basis and selector functions. If during the decomposition equations IV-3, IV-4 and IV-5 are used then a Pseudo Kronecker expression will be formed. However, if the expansion is carried out using only equations IV-4 and IV-5 then the resulting expression will be in a Generalized Reed-Muller form. The choice between the two forms depends on which option produces basis and selector functions of lower cost.

Another work closely related to FLDS is called *factor* [Hwang90]. *Factor* is an approach to logic decomposition based on Single Value Decomposition (SVD). FLDS differs from *factor* in several ways.

During the decomposition of a single output function the main difference between the two approaches is the use of basis and selector optimization algorithm in FLDS. This algorithm, as has been shown in Section 5, provides the means to further reduce the cost of logic implementation, even if the initial set of basis functions is minimal. As a result, if FLDS can generate an optimal circuit for a given logic function, it can do so for several difference logic partitions equally well. This is not the case with *factor*. In fact, *factor* requires not only a good variable partitioning, but also a good variable ordering within each partition to achieve a good result, because it does not use an algorithm similar to the basis and selector optimization. Thus, to get a better result *factor* must rely on solving a problem of variable ordering, which is more time consuming than variable partitioning. For larger

functions, it may be impractical to use *factor*.

Factor also differs from FLDS for the decomposition of a multi-output function. In *factor* multi-output decomposition is performed in two phases. The first phase is to determine the variable partitioning for each output. To consider multi-output functions, *factor* chooses the same partitioning for all functions. The cost of each partition is based on the maximum number of sub-functions the SVD approach generates for any of the output functions individually. For example, if functions f and g , were being synthesized by *factor*, and for a given partition function f had two sub-functions and function g had four, the partition would be assigned a cost of four. *Factor* then picks the partition with least such cost. It then synthesizes each function separately, reusing any sub-functions related by equality or inversion as needed.

There are two main differences between *factor* and FLDS in terms of multi-output decomposition. First, a decomposition generated by *factor* may result in sub-functions of function f to be linearly dependent on sub-functions of g . Allowing this dependency to persist causes an area penalty to be incurred. Second, *factor* does not attempt to reduce the cost of the sub-functions. As noted by Hwang *et al.* [Hwang90] for n sub-functions there are $n!$ possible valid combinations of them and hence optimization is not performed. Both of the above result in more emphasis being put on variable partitioning, making it more complex to get the same result.

FLDS addresses both linear dependence and sub-function minimization of multiple output functions. First, it generates only linearly independent set of sub-functions. Any linear dependencies are removed because FLDS uses Gaussian Elimination to decompose logic functions. Second, a basis and selector function optimization algorithm, that runs in $O(n^2)$ for n basis vectors, does a good job at finding lower cost sub-functions. As demonstrated in Section 5, this helps us reduce logic area. To illustrate the differences consider the following example.

Figure IV-18 shows two functions, f and g , as they are represented by FLDS and *factor* [Hwang90]. During synthesis, the algorithm presented by Hwang *et al.* [Hwang90] uses sub-functions $\mathbf{G}_1=a$ and $\mathbf{G}_2=b$ to decompose f and $\mathbf{G}_3=a\oplus b$ and $\mathbf{G}_1=a$ to decompose g . However, the algorithm does not exploit the relationship $\mathbf{G}_3=\mathbf{G}_1\oplus\mathbf{G}_2$ to reduce logic area, and as a consequence synthesizes functions f and g using a total of 6 two-input gates. In contrast, FLDS optimizes \mathbf{G}_3 away because of a linear dependence on \mathbf{G}_1 and \mathbf{G}_2 . FLDS decomposes both f and g with respect to \mathbf{G}_1 and

ab	f				g				Factor
	cd	01	10	11	cd	01	10	11	
00	0	0	0	0	0	0	0	0	00
01	1	1	0	0	1	1	0	0	11
10	0	0	1	1	1	1	1	1	01
11	1	1	1	1	0	0	1	1	10

FLDS

Figure IV-18: Example of the differences between FLDS and *Factor*

G_2 , synthesizing the circuit using only four two-input gates.

IV.10 Experimental Results

In this section we present area, depth and compilation time results and compare to previously published results. The following subsections contain the experimental methodology, summary of results, and discussion.

IV.10.1 Methodology

In this work the area results obtained from FLDS were compared to two academic synthesis systems: ABC [ABC05] optimizes a circuit using structural transformations, and BDS-PGA 2.0 [Vemuri02] performs boolean optimization by utilizing BDDs.

To obtain results using ABC, each circuit was first converted to an AND-Inverter Graph using the *strash* command to allow ABC to apply its optimizations. The circuit was then resynthesized using the *resyn2* script and the resulting logic was mapped into 4-LUTs using ABC's *fpga* command. The time measurement included the *strash*, *resyn2*, and *fpga* steps.

To obtain results using BDS-PGA 2.0 a few more steps had to be taken, because the BDS-PGA 2.0 system contains various flags that guide its synthesis process. The flags of particular interest are:

- **xhardcore (X)** - a flag to enable x-dominator based XOR decompositions

- sharing (**S**) - a flag used to perform sharing extraction prior to decomposition
- heuristic (**H**) - a flag used to enable a heuristic variable ordering algorithm.

Depending on the settings of these flags a circuit synthesized with BDS-PGA 2.0 may have a different final area. Thus, we performed a sweep across all combinations of these three parameters and recorded the best result, based first on area, then depth and finally on processing time. The resulting network was then mapped into 4-LUTs using the ABC system.

The results for FLDS were obtained by running the FLDS algorithm on each circuit, varying only one parameter called cone size. The cone size parameter is used within FLDS in the preprocessing step to create cones for synthesis. The cone size parameter specifies the maximum number of inputs to which a cone can be grown. To obtain our result the cone size was set to 8, 12, 16, 20 and 24 inputs. The resulting circuit was mapped into 4-LUTs using the ABC system.

IV.10.2 Results

Table IV-1 presents the results obtained for 25 XOR-based logic circuits. The name of each circuit is given in the first column. The following three columns list the area (LUT-wise), depth and processing time results ABC Synthesis System [ABC05] obtained for the given set of circuits. Columns 5 through 7 list the area, depth and processing time results for BDS-PGA 2.0 [Vemuri02]. The setting used to obtain the result for BDS-PGA 2.0 is given in columns 8 through 10, which corresponds to a particular BDS-PGA 2.0 flag (**X**, **H**, or **S**) as described in the previous section. For a given circuit an ‘X’ appears in these columns if the flag was turned on to generate the given result.

In columns 11 through 13 area, depth and processing time results obtained by our FLDS system are shown. For each circuit, column 14 states the cone size used to generate cones for logic synthesis. The final four columns compare results obtained with FLDS to ABC (columns 15 and 16) and to BDS-PGA 2.0 (columns 17 and 18).

Table IV-2 shows the results for non-XOR based logic circuits from the MCNC benchmark set. The columns in Table IV-2 have the same meaning as in Table IV-1.

To compare the results obtained using various tools we used the following equation:

$$\%difference = 100 \frac{(final - initial)}{MAX(initial, final)} \quad \text{(IV-6)}$$

Table IV-1: Area Results Comparison Table for XOR-based Circuits

Name	ABC			BDS-PGA-2.0					FLDS				FLDS vs. ABC		FLDS vs. BDS		
	LUTs	Depth	Time (s)	LUTs	Depth	Time (s)	X	H	S	LUTs	Depth	Time (s)	Cone Size	Area	Depth	Area	Depth
5xp1	40	4	0.08	25	4	0.07	X			22	4	0.02	8	-45.00	0.00	-12.00	0.00
9sym	114	6	0.08	58	6	0.54	X			13	4	0.031	20	-88.60	-33.33	-77.59	-33.33
9symml	85	5	0.08	19	6	0.17	X			13	4	0.051	12	-84.71	-20.00	-31.58	-33.33
alu2	150	9	0.09	208	12	0.82	X	X		102	7	0.082	24	-32.00	-22.22	-50.96	-41.67
C1355	75	4	0.14	77	6	3.87	X	X	X	80	6	0.036	8	6.25	33.33	3.75	0.00
C1908	107	8	0.11	122	9	0.91	X	X		167	13	0.077	8	35.93	38.46	26.95	30.77
C3540	363	12	0.22	418	14	2.99				613	17	0.236	8	40.78	29.41	31.81	17.65
C499	77	5	0.13	80	4	3.42				77	6	0.035	8	0.00	16.67	-3.75	33.33
C880	112	9	0.14	123	9	1.21	X	X	X	167	11	0.067	8	32.93	18.18	26.35	18.18
cordic	304	8	0.16	161	18	1501				24	5	9.735	24	-92.11	-37.50	-85.09	-72.22
count	42	5	0.06	39	5	0.12	X	X		52	5	0.036	8	19.23	0.00	25.00	0.00
dalu	255	6	0.16	326	9	2.34	X	X		363	9	0.128	8	29.75	33.33	10.19	0.00
des	1340	6	0.67	1338	7	6.03				1155	8	5.456	16	-13.81	25.00	-13.68	12.50
f51m	41	4	0.06	32	5	0.12				18	4	0	12	-56.10	0.00	-43.75	-20.00
inc	51	4	0.06	45	4	0.1				51	4	0.035	8	0.00	0.00	11.76	0.00
my-adder	32	16	0.08	33	16	0.89	X	X	X	35	16	0.051	24	8.57	0.00	5.71	0.00
rd53	10	3	0.05	12	3	0	X	X		7	2	0	12	-30.00	-33.33	-41.67	-33.33
rd73	63	4	0.09	15	4	0.09	X			11	3	0.015	12	-82.54	-25.00	-26.67	-25.00
rd84	106	6	0.09	25	5	0.4				15	3	0.031	8	-85.85	-50.00	-40.00	-40.00
sqrt8	26	4	0.06	22	4	0.06				12	3	0	12	-53.85	-25.00	-45.45	-25.00
sqrt8ml	20	5	0.08	26	6	0.07	X	X		12	3	0.02	16	-40.00	-40.00	-53.85	-50.00
squar5	18	3	0.06	18	3	0.03	X	X		15	2	0.015	24	-16.67	-33.33	-16.67	-33.33
t481	135	6	0.11	70	6	22.81	X	X		5	2	0.078	24	-96.30	-66.67	-92.86	-66.67
xor5	2	2	0.05	2	2	0				2	2	0.015	8	0.00	0.00	0.00	0.00
z4ml	7	3	0.05	6	3	0.01	X	X		8	3	0.02	8	12.50	0.00	25.00	0.00
Total/Average			2.96			1548						16.27		-25.26	-7.68	-18.76	-14.46
Ratio			1			523						5.497					

Table IV-2: Area Results Comparison Table for non-XOR based logic circuits

Name	ABC			BDS-PGA-2.0					FLDS				FLDS vs. ABC		FLDS vs. BDS		
	LUTs	Depth	Time (s)	LUTs	Depth	Time (s)	X	H	S	LUTs	Depth	Time (s)	Cone Size	Area	Depth	Area	Depth
apex3	706	6	0.31	808	13	17.12	X	X	X	843	8	0.971	8	16.25	25.00	4.15	-38.46
apex6	252	5	0.11	252	6	0.64				337	7	0.593	24	25.22	28.57	25.22	14.29
apex7	73	5	0.08	82	6	0.18	X	X		103	9	0.092	24	29.13	44.44	20.39	33.33
b1	4	1	0.03	4	1	0				4	1	0.02	20	0.00	0.00	0.00	0.00
b12	29	3	0.06	34	4	0.04				36	4	0.02	12	19.44	25.00	5.56	0.00
b9	38	3	0.08	40	3	0.04	X			57	4	0.035	8	33.33	25.00	29.82	25.00
bw	105	4	0.08	79	5	0.29	X	X	X	68	4	0.035	12	-35.24	0.00	-13.92	-20.00
C17	2	1	0.05	2	1	0	X	X		2	1	0.02	12	0.00	0.00	0.00	0.00
C432	63	11	0.06	166	16	1.33				123	15	0.123	12	48.78	26.67	-25.90	-6.25
c8	37	3	0.06	37	3	0.06	X			38	4	0.015	8	2.63	25.00	2.63	25.00
cc	25	2	0.05	24	2	0.01	X			26	3	0.0	20	3.85	33.33	7.69	33.33
cht	38	2	0.05	41	2	0.13	X	X	X	48	3	0.0	24	20.83	33.33	14.58	33.33
clip	102	5	0.09	52	5	0.29				37	4	0.082	12	-63.73	-20.00	-28.85	-20.00
cm138a	10	2	0.06	10	2	0.01	X			10	2	0.015	8	0.00	0.00	0.00	0.00
cm150a	13	4	0.05	13	4	0.03				25	6	4.39	24	48.00	33.33	48.00	33.33
cm151a	7	3	0.03	8	3	0.01	X			11	4	0.02	8	36.36	25.00	27.27	25.00
cm152a	6	3	0.03	6	3	0	X			6	3	0.035	24	0.00	0.00	0.00	0.00
cm162a	12	3	0.05	18	3	0.01	X			18	4	0.02	8	33.33	25.00	0.00	25.00

cm163a	11	3	0.05	11	3	0.01	X	13	4	0.0	8	15.38	25.00	15.38	25.00
cm42a	10	1	0.05	10	1	0.01	X X X	10	1	0.02	8	0.00	0.00	0.00	0.00
cm82a	4	2	0.06	4	2	0	X X	5	2	0.0	20	20.00	0.00	20.00	0.00
cm85a	12	3	0.05	11	3	0.05	X X	12	4	0.015	12	0.00	25.00	8.33	25.00
cmb	16	3	0.05	19	4	0.04	X X	11	2	0.015	20	-31.25	-33.33	-42.11	-50.00
comp	30	4	0.06	33	6	2.51	X X	30	5	0.895	20	0.00	20.00	-9.09	-16.67
con1	6	2	0.03	5	2	0	X X	7	2	0.0	16	14.29	0.00	28.57	0.00
cu	17	3	0.05	20	3	0.07	X X X	19	3	0.015	12	10.53	0.00	-5.00	0.00
decod	20	2	0.03	20	2	0.02	X	24	2	0.015	20	16.67	0.00	16.67	0.00
duke2	192	5	0.11	227	6	0.35	X	268	6	0.077	8	28.36	16.67	15.30	0.00
e64	226	4	0.08	233	5	0.62	X X	280	5	0.067	16	19.29	20.00	16.79	0.00
ex5p	882	7	0.41	871	8	14.04	X X	173	4	0.208	16	-80.39	-42.86	-80.14	-50.00
example2	121	4	0.08	125	5	0.18		140	6	0.036	8	13.57	33.33	10.71	16.67
frg1	39	6	0.06	55	9	0.29	X	70	8	0.035	8	44.29	25.00	21.43	-11.11
frg2	279	6	0.16	265	6	0.96	X X X	355	9	0.097	8	21.41	33.33	25.35	33.33
i1	19	3	0.05	17	3	0.01	X	19	3	0.0	16	0.00	0.00	10.53	0.00
i2	73	5	0.05	86	6	0.74		76	7	0.035	8	3.95	28.57	-11.63	14.29
i3	46	3	0.05	46	3	0.28		46	3	0.067	12	0.00	0.00	0.00	0.00
i4	82	5	0.08	85	5	0.87	X	97	6	0.083	8	15.46	16.67	12.37	16.67
i5	101	6	0.08	101	6	0.15	X X	115	6	0.015	16	12.17	0.00	12.17	0.00
i6	144	2	0.08	121	2	0.21	X X	108	2	0.02	12	-25.00	0.00	-10.74	0.00
i7	181	2	0.08	167	2	0.34	X	167	3	0.067	16	-7.73	33.33	0.00	33.33
i8	447	5	0.2	414	7	2.33	X X X	450	9	0.404	12	0.67	44.44	8.00	22.22
i9	262	5	0.13	211	5	0.86	X X	300	8	0.097	8	12.67	37.50	29.67	37.50
lal	32	3	0.05	30	3	0.04		33	3	0.036	8	3.03	0.00	9.09	0.00
ldd	35	3	0.05	34	3	0.09	X X X	41	4	0.035	8	14.63	25.00	17.07	25.00
majority	3	2	0.03	3	2	0		3	2	0	20	0.00	0.00	0.00	0.00
misex1	21	3	0.05	20	3	0.01	X X	19	3	0.02	8	-9.52	0.00	-5.00	0.00
misex2	40	3	0.05	43	4	0.07	X X X	46	4	0.015	16	13.04	25.00	6.52	0.00
misex3c	440	6	0.19	570	12	34.49		291	7	5.816	24	-33.86	14.29	-48.95	-41.67
mux	13	4	0.06	13	4	0.04		20	5	0.036	8	35.00	20.00	35.00	20.00
pair	534	7	0.22	529	9	3.76	X X X	736	9	0.268	8	27.45	22.22	28.13	0.00
parity	5	2	0.06	5	2	0.01	X	5	2	0	8	0.00	0.00	0.00	0.00
pcle	21	3	0.05	20	3	0.03		23	4	0.035	8	8.70	25.00	13.04	25.00
pcle8	31	4	0.06	32	4	0.06	X X X	34	4	0.02	8	8.82	0.00	5.88	0.00
pdcc	3302	8	1.56	3966	13	18.74	X	693	9	44.14	20	-79.01	11.11	-82.53	-30.77
pm1	19	2	0.06	18	2	0.04	X X	20	3	0	24	5.00	33.33	10.00	33.33
rot	246	8	0.11	272	10	17.59	X X X	328	11	0.097	8	25.00	27.27	17.07	9.09
sao2	82	4	0.08	96	8	0.35	X	32	3	0.076	12	-60.98	-25.33	-66.67	-62.5
sct	23	3	0.06	23	3	0.01	X X	24	4	0.02	8	4.17	25.00	4.17	25.00
spla	2634	8	1.45	334	8	22.44		398	9	0.925	16	-84.89	11.11	16.08	11.11
table3	393	7	0.17	445	9	3.55	X X X	537	9	0.144	8	26.82	22.22	17.13	0.00
table5	412	6	0.17	462	10	7.73	X X X	550	9	0.144	8	25.09	33.33	16.00	-10.00
tcon	16	1	0.06	16	1	0	X X	16	1	0.0	16	0.00	0.00	0.00	0.00
term1	61	5	0.08	68	5	0.29	X X X	93	8	0.036	8	34.41	37.50	26.88	37.50
too-lrg	141	6	0.09	176	8	0.54	X	226	10	0.051	8	37.61	40.00	22.12	20.00
ttt2	49	4	0.06	56	5	0.14		53	5	0.035	12	7.55	20.00	-5.36	0.00
unreg	48	2	0.06	37	2	0.06	X	40	3	0.016	8	-16.67	33.33	7.50	33.33
vda	260	5	0.13	273	7	1.41	X X	364	8	0.3	8	28.57	37.50	25.00	12.50
vg2	33	4	0.06	42	5	0.15	X X	68	9	0.051	24	51.47	55.56	38.24	44.44
x1	120	4	0.08	128	5	0.23		154	5	0.036	8	22.08	20.00	16.88	0.00
x2	19	3	0.03	17	4	0.03		18	4	0	12	-5.26	25.00	5.56	0.00
x3	231	5	0.11	248	5	0.91	X X	294	7	0.143	12	21.43	28.57	15.65	28.57
x4	123	4	0.08	158	4	0.24		143	5	0.051	20	13.99	20.00	-9.49	20.00
ex1010*	4094	8	1.52	Failed to synthesize				1063	7	13.94	12	-74.04	-12.5	Failed to	
misex3*	1093	6	0.44	Failed to synthesize				493	10	3.8	16	-54.89	40.0	synthesize	
Total/Average			8.83			158.18				61.27		6.20	16.48	4.78	6.19
Ratio			1			17.9				6.94					

* circuits excluded from total/average and ratio computation as they did not synthesize with BDS-PGA 2.0

This equation ensures that both gains and losses presented in the table of results are weighted equally.

Overall, FLDS results show that XOR-based logic functions can be significantly reduced in size and logic depth, 18.8% and 14.5% respectively, as compared to logic minimization tools like BDS-PGA 2.0. The cost of this optimization is the increase in area for non-XOR based logic circuit, both in area and in depth by 4.8% and 6.2% respectively. As compared to a structural optimization techniques such as those used in ABC, FLDS synthesizes XOR-based circuits using 25.3% less area and reduces logic depth by 7.7%. For non-XOR based logic circuits, ABC produces circuits with 6.2% lower area results and 16.5% lower depth on average than FLDS.

IV.10.3 Discussion of Individual Circuits

As shown in Table IV-1 significant area savings are found using FLDS. The largest area savings for XOR based circuits were observed for *9sym*, *cordic*, *sqrt8ml*, and *t481*.

9sym is a circuit with a single 9 input logic function. The function itself consists of a number of XOR gates. Interestingly enough, *9symml* is exactly the same circuit, except with a different initial representation in the BLIF file, was synthesized much better than *9sym*. In *9sym* BDS-PGA 2.0 found the circuit to consist of 9 logic functions and hence was unable to find a compact implementation for it, though it faired 30% better than ABC. However, for *9symml* BDS-PGA found it to be a single output 9 input function and created a single BDD for it. Despite finding a single cone of logic for *9symml*, BDS-PGA produced a 30% worse result than FLDS. This example shows that functionally linear decomposition and x-dominator based decompositions are quite different. While both strive to utilize XOR gates, searching for an x-dominator in a BDD clearly does not guarantee finding a better decomposition.

Another interesting circuit is *cordic*. It consists of two functions sharing 23 inputs. In this particular case the results we obtained come from two sources - the ability to synthesize large functions and synthesizing multi-output functions. FLDS found the functions to be reasonably easy to synthesize using a balanced decomposition.

In contrast, neither ABC nor BDS-PGA found there to be only 2 cones of logic. Instead, many cones were created to represent this circuit. In the case of BDS-PGA, 32 logic functions were

created. While in Table IV-1 the synthesis flags for this circuit do not include x-dominator decompositions usage, the results with the same flag turned on were identical, except for the longer processing time.

Sqrt8ml is an arithmetic circuit that computes a square root of a binary number. It consists of 4 logic functions, with 2, 4, 6 and 8 inputs. The two functions that fit in a single LUT were synthesized separately, but the 6 and 8 input functions were synthesized in tandem. The algorithm found there to be 5 basis vectors shared by these two functions. In the worst case scenario this would have caused 15 LUTs to be created to implement these two functions, but due to our basis and selector function optimization routine, this was not the case. Specifically, the basis and selector function optimization led to the realization of the basis and selector functions using fewer variables for each. Thus, the mapping stage of the CAD flow could successfully map both functions into a total of 10 LUTs. The result we obtained was nearly half the size of the one produced by ABC, and less than half of the area taken by the same circuit synthesized using BDS-PGA.

The *t481* circuit is a 16 input single output logic function, which has a compact representation in the Fixed-Polarity Reed-Muller form. When synthesized correctly, it takes only 5 4-LUTs, which is the minimum for this circuit. In FLDS the function was synthesized in 0.125 seconds, finding only two basis functions during balanced decomposition. In this particular case, the variable partitioning algorithm was successful in finding the correct assignment of variables to the bound and the free set, which allowed for minimum area synthesis of the logic function.

The success of this method is not limited to XOR-based logic functions. As shown in Table IV-2, numerous logic circuits that are considered not to be XOR-based have benefitted significantly as well. Examples of such circuits are *ex5p*, *pdc*, *sao2*, and *ex1010*. In these circuits, *pdc* and *ex1010* in particular, the success is attributed to synthesis of multi-output functions, rather than synthesizing each cone of logic separately. The *pdc* circuit did take FLDS 2.5 times longer to synthesize than when using BDS-PGA 2.0, but the result is nearly six times smaller and with a 30% lower logic depth.

The results produced by FLDS can be further improved by applying existing logic synthesis techniques on top of FLDS. In our research we investigated the effect of applying the *resyn2* script of ABC on top of FLDS results to determine if it can further reduce logic area. We found that if ABC

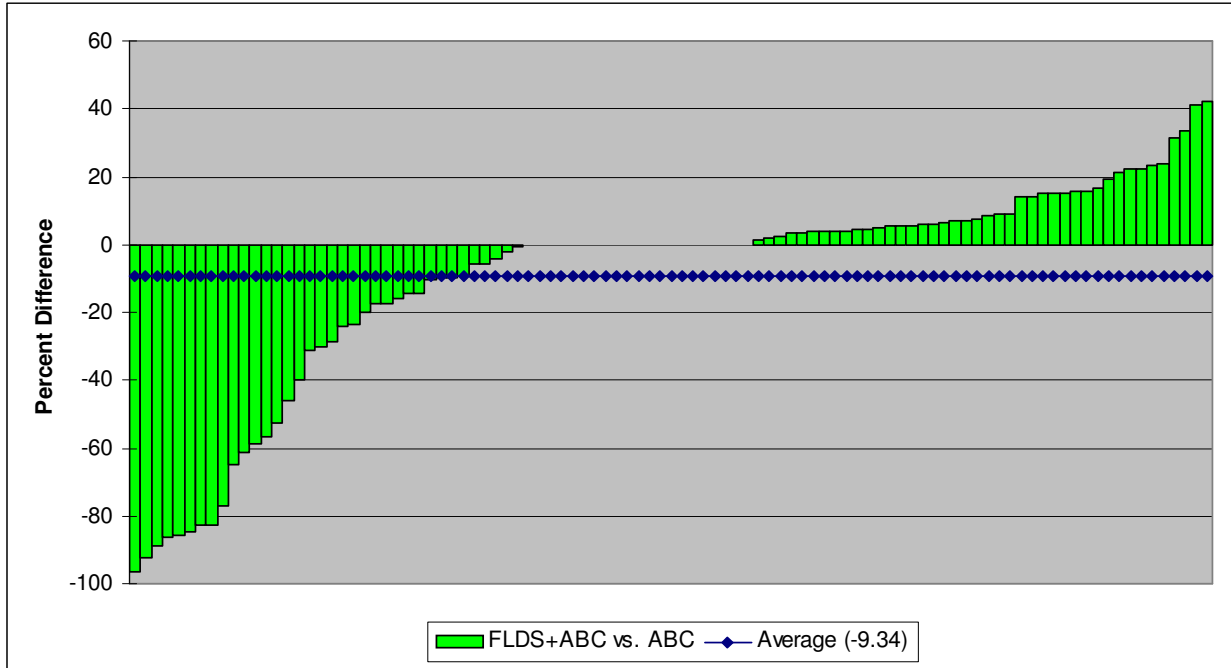


Figure IV-19: Distribution of area savings of FLDS combined with ABC versus ABC alone

is used on top of FLDS and the results are compared to those obtained using ABC alone for the same set of circuits, then we obtain a 24.2% smaller area and a 16.2% lower depth for XOR-based circuits. For non-XOR based circuits area was 4.25% smaller with only a 1% depth increase. Averaged over both XOR-based and non-XOR based circuits, FLDS together with ABC can reduce an average circuit size by 9.3% and reduce the depth by 3.3%, as compared to what ABC can do alone. The distribution of results is presented in Figure IV-19.

IV.11 Conclusion and Future Work

The Functionally Linear Decomposition and Synthesis technique was found to be very effective in reducing area taken by logic functions heavily dependent on XOR gates. It achieved a 25% reduction in area as compared to ABC [ABC05], as well as an 18.8% reduction in comparison to BDD-based approaches used in BDS-PGA 2.0 [Vemuri02]. FLDS also offers a depth reduction of 7.7% and 14.48% with respect to the leading synthesis approaches [Vemuri02][ABC05]. For non-XOR intensive logic circuits the technique showed only a minor area and depth penalty. For both of these groups of functions FLDS performed synthesis rapidly.

The technique presented here can be further improved. In the short term, the most prominent extension of this technique is to cover non-disjoint decomposition. A challenge here will be to modify the variable partitioning technique to select the correct variables to overlap between the bound and the free sets.

Another extension to consider is to apply the proposed approach to multi-valued logic decomposition. Since this approach is quite generic in terms of the underlying mathematical field used, it will be valid for any valid $GF(m)$. However, the addition, subtraction and multiplication operations may prove more complex for $m > 2$, mostly because the final implementation is in terms of Boolean logic.

Chapter V

Dynamic Power Reduction

In this chapter we address the second aspect of physical synthesis - implementation of incremental changes to a logic circuit. While previously we focused on synthesizing a logic circuit, and thereby showing that within the PST framework a synthesis change is feasible, in this chapter we want to make only small incremental changes. The purpose of these changes is to improve a previously synthesized circuit to reduce the amount of dynamic power it dissipates. Furthermore, we show how to take advantage of post-routing information to achieve power reduction.

In this chapter a method for reducing the dynamic power dissipated by Field-Programmable Gate Array circuits is presented. This method uses a probabilistic approach for detecting glitches in a fully placed and routed circuit and then applies a glitch reduction technique that uses negative-edge-triggered flip-flops to reduce the number of glitches in a logic circuit. The key idea behind the glitch reduction technique is to insert negative-edge-triggered flip-flops at outputs of Lookup Tables that produce glitches. It maintains the logic value produced by the LUT in the previous clock cycle for the first half of the clock period, filtering glitches that occur at the output of the LUT. This method reduces the number of logic value transitions that propagate to the general routing network, thereby reducing the dynamic power dissipated by a logic circuit.

This approach was tested on an commercial FPGA, Altera's Stratix II. The power reduction

technique was applied to a set of 8 benchmark circuits and the results were obtained using the Power Play Power Analyzer in Quartus II. The results show a reduction in the dynamic power dissipation by 7% on average and up to 25%.

V.1 Introduction

Field-Programmable Gate Array (FPGA) devices are a popular choice for low to medium volume digital applications. They can implement various different circuits without a need to fabricate a new chip, reducing costs and time to market. This flexibility is paid for by increased power dissipation. The power consumed by a logic circuit implemented on an FPGA device in a 90nm process, or smaller, is of concern for the FPGA community, especially if an FPGA device is to be used in a mobile application.

The power dissipation of a commonly used 4-input LUT based FPGA consists 40% of static power and 60% of dynamic power, with static power increasing with shrinking feature sizes and increasing LUT sizes [Li03]. In this work we are concerned strictly with the dynamic power dissipated by a logic circuit.

Several techniques have been proposed to reduce dynamic power. At the circuit level, effective power reduction techniques include the insertion of control logic that synchronizes inputs of logic blocks to force at most one transition to occur at the output of the logic block [Mahapatra00], and reducing the power supply voltage [Chandrakasan95]. At the logic level, the dynamic power dissipation is addressed during synthesis and technology mapping. During synthesis rewiring can be used to reduce the toggle rate of internal signals [Shen92][Bahar96], local logic transformations can be applied to reduce the number of gates whose output toggles frequently [Kim02], and redundant gates [Pradhan96] can be added to reduce toggle rate on wires. During technology mapping, power can be reduced by covering connections with high toggle rates within a LUT and minimizing logic replication [Anderson02]. Also, proposing several local mappings and selecting the lowest power mapping [Yeh99], or trading off area and depth of a circuit for lower toggle rates on wires [Farrahi94][Li01][Wang01], can be effective. Pipelining and retiming have also been used to rebalance the delays in a circuit and subsequently reduce toggle rates on wires [Leijten95].

This work focuses on logic level techniques, which in prior works is most often addressed early in the CAD flow. While there are works that address dynamic power early in the design flow that can achieve significant power reduction, the contribution of *glitches* to the overall power dissipation is not well defined until the circuit is placed and routed. These glitches can as much as double the toggle rate of wires in the circuit, causing a substantial increase in dynamic power dissipation [Anderson03].

The following approach addresses dynamic power dissipation due to glitches in the context of physical synthesis. In this work we will optimize logic circuits post-routing, where the delays between LUTs are known. To reduce power in a circuit we select LUTs that produce glitches and insert a negative-edge-triggered flip-flop at their outputs. Negative-edge-triggered flip-flops will maintain the output wire state for the first half of the clock cycle. When the clock signal toggles, a flip-flop will update its value and only the new logic value will be observed. This approach masks glitches produced by a LUT during the first half of a clock cycle from the rest of the circuit. Because the glitches are not propagated to the general routing and the subsequent logic, the dynamic power dissipation of a circuit is reduced.

V.2 Background and Terminology

This section provides a review of the background information and outlines the terminology used throughout the chapter. This chapter focuses on the review of probability theory, which we use to model toggle rate of wires in a logic circuit. Also, a short background on power dissipation in FPGAs is presented.

V.2.1 Probability Theory

In the theory of probability we assign a probability value between 0 and 1 to each event of interest. A value of 0 denotes that an event cannot occur, while a value of 1 assumes a 100% probability of an event occurring. If we assume that \mathbf{A} is an event then the probability of that event occurring is denoted as $P[\mathbf{A}]$. For example, if in a logic circuit we have a wire x then the probability of x being at a logic value 1 would be denoted as $P[x=1]$, or $P[x]$ in a short form.

More complex events can also be described. In particular, we are interested in a probability

of several wires changing logic state during a single clock cycle. For example, we may want to know what is the probability of wires x and y changing from 00 to 10. To denote this event we will use notation $\{xy=00 \cap x'y'=10\}$ where xy indicates the current state of wires x and y , while $x'y'$ indicates the logic state of wires x and y in the following clock cycle. The probability of this event occurring would be denoted as $P[xy=00 \cap x'y'=10]$.

To denote a probability of a conditional event we will use the $|$ symbol. For example, $P[\mathbf{A}|\mathbf{B}]$ denotes the probability of event \mathbf{A} occurring given that event \mathbf{B} has occurred. Recall that according to Bayes' Rule:

$$P[A|B] = \frac{P[A \cap B]}{P[B]} \quad (\text{V-1})$$

V.2.2 FPGA Power

Power dissipation consists of three components: static, short-circuit and dynamic power. Static power dissipates when current flows through a transistor even when the transistor is off. The short-circuit power is power dissipated by a CMOS gate during a short period of time when both pull-up and pull-down networks conduct current. Finally, the dynamic power is dissipated every time any capacitor in a circuit is charged or discharged.

Studies show that for a 4-LUT based FPGA the power dissipation is 40% due to static and 60% due to dynamic power dissipation [Li03], while the short-circuit power is negligible in comparison [Shang02]. The dynamic power dissipation can be computed as follows:

$$P_{avg} = \frac{1}{2} \sum_{i=1}^{\text{number of nets}} (C_i f_i V^2) \quad (\text{V-2})$$

where P_{avg} is the average dynamic power dissipation, C_i is the capacitance of a net, f_i is the average toggle rate of a net and V is the power supply voltage. By using Equation 2 and creating models for LUT and flip-flop dynamic power dissipation we can estimate how much dynamic power an FPGA circuit dissipates. These power models are described in the following section.

V.3 Power Models

The focus of this work is to minimize the number of glitches produced by a logic circuit and therefore reduce the dynamic power it dissipates. To do so a power model for our optimizations is needed. In particular, it is crucial to compute the average toggle rate for a net in each circuit as well as estimate the capacitance of that net.

Several works that address the topic of toggle rate estimation and/or prediction already exist. Most notably is the concept of transition density [Najm93] where the toggle rate is represented using probability theory. The main advantage of this approach is that if we know how often inputs to a logic gate toggle, then we can represent this information numerically and can compute with reasonable accuracy how frequently the output of the gate toggles.

In [Anderson03] Anderson and Najm look at the transition density (average number of times a signal toggles per unit time) [Najm93] and how this concept applies to logic circuits implemented on FPGAs. In their work the transition density is represented as a weighted sum of transitions generated and propagated through a LUT. Generated transitions are a function of LUT depth and the number of paths to LUT inputs. Propagated transitions are due to glitches that propagate through a LUT and are based on the logic function the LUT implements. The advantage of this approach is that no simulation is required.

The work of Anderson and Najm was intended to predict the toggle rate of a circuit before it is placed and routed. The work by Tsui, Pedram and Despain [Tsui93] focuses on transition density computation once circuit delays are known. Their approach is basically an event driven simulation of a circuit, except that instead of logic 0/1 signals being propagated through the circuit, a probability waveform is used. This method effectively combines multiple events on primary inputs of a circuit into a single probabilistic waveform to obtain transition density values. This approach is faster than straight forward symbolic simulation, with a 99% estimation accuracy.

In this work we build on the above described works to create a model suitable for glitch reduction in the context of physical synthesis. In our approach we borrow the idea of glitch forwarding using Boolean Difference from [Anderson03] and [Najm93]. We process each LUT in a manner similar to that in [Tsui93], but we use the concept of transition density to propagate glitches through a LUT, rather than a probabilistic waveform originating at primary inputs of a

Table V-1: Signal Properties

Property	Description	Notation
Static Probability	Probability a signal will assume a logic value 1 in any given cycle	$P[y]$
Transition Probability	Probability a signal will change logic state in any given clock cycle	$P_t(y)$
Low to High Transition Probability	Probability a signal will change value to logic 1, given it is currently at logic value 0	$P[y'=1 y=0]$
High to Low Transition Probability	Probability a signal will change value to logic 0, given it is currently at logic value 1	$P[y'=0 y=1]$
Transition Density	The average number of logic transitions per cycle, including glitches	$D(y)$

circuit. A detailed description of our toggle rate computation method is described in Section 3.1.

In addition to toggle rate computation we also discuss the net capacitance model and a LUT power model. The net capacitance model is used to determine if a change of toggle rate on a particular net will save sufficient dynamic power to justify the costs involved with the optimization technique used. The net capacitance model is discussed in Section 3.2. Finally, in Section 3.3 we briefly describe a LUT power model, which represents the dynamic power a LUT dissipates based on the average number of times per second its output toggles.

V.3.1 Average Net Toggle Rate Computation

To determine the transition density of a LUT output net we process a logic circuit in topological order, one LUT at a time. At each LUT we determine the transition density for the output of the LUT and use it in the computation of transition densities in the LUTs down stream. However, the transition density value by itself is insufficient to adequately represent the behaviour of a signal in a logic circuit. Thus, in addition to transition density we define several signal properties that we compute for each net. The signal properties are listed in Table V-1.

The first property is the *static probability*, otherwise known as *equilibrium probability* [Najm95]. This is a value between 0 and 1 which indicates a probability that a signal assumes a logic value one. The second property is the *transition probability*, which defines how often a signal changes state. This property does not account for any glitches that may occur, but rather looks at the

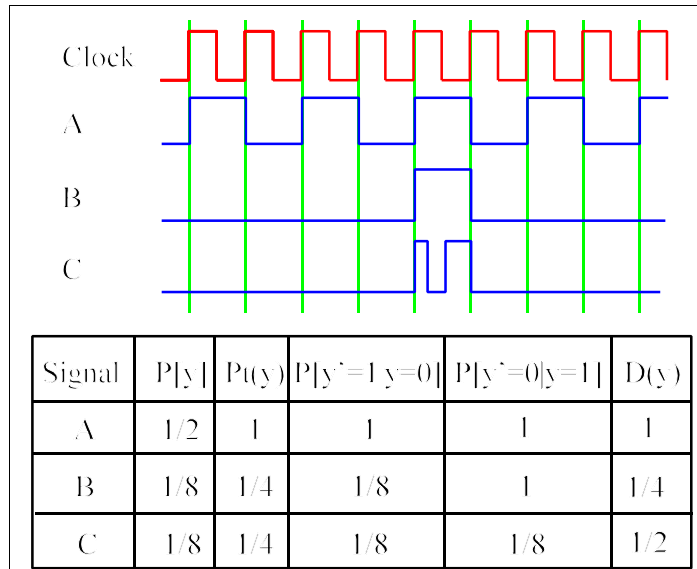


Figure V-1: Examples of logic signals and properties

functional behaviour (zero-delay model) of a signal. This property can be broken down into two components. The first of these is the *low to high transition probability*, which indicates the probability a signal will change state from a logic zero to a logic one. The second is the *high to low transition probability* that indicates a one to zero transition probability. Both properties are considered conditional, since they only apply if the logic value is at a particular state to begin with. For example, the low to high transition probability only applies when a signal is in a logic zero state.

The final property is the *transition density*. The transition density encompasses all transitions on a per clock cycle basis. It is similar to the transition probability, but it also includes transition due to glitches. Thus, to compute the number of glitches that occur on a per clock cycle basis on any given net y , we compute the difference $D(y)-P_t(y)$.

In Figure V-1 we show a few examples of how the above signal properties can describe the behaviour of a signal in a logic circuit. The first waveform is the clock, which is a reference signal for the samples in Figure V-1. Signal A is an oscillating signal that toggles every clock cycle. Its static probability is therefore $1/2$. The other properties are set to 1 because in each clock cycle the signal toggles once. Notice that the conditional transition probabilities are also 1, indicating that during the next clock cycle the wire will assume a different logic state. The same cannot be said for signal B. In this case only the one to zero probability is 1, but the zero to one probability is $1/8$. This indicates that on average the signal will become high for one clock period, once every eight clock

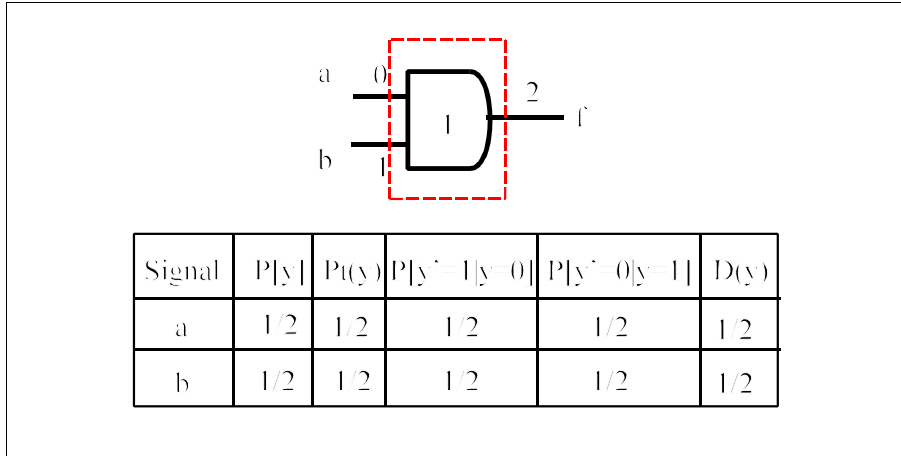


Figure V-2: Logic circuit and signal properties for Example 1

cycles. Neither signal contains glitches and hence the transition density equals the transition probability for the given signal. Signal C is the same as signal B, except it contains glitches. To indicate this, the transition density is higher by $\frac{1}{4}$ with respect to transition probability. This indicates two transitions every eight cycles are due to glitches.

We will now demonstrate how we can compute each of these properties by using two examples. First, we will show how to compute these values when the inputs to a LUT are free of glitches. Then we will show how an input with glitches affects the transition density of a LUT output net.

V.3.1.1 Example 1 - Glitch-Free Inputs

In this example we demonstrate how to compute the net properties in Table V-1 for a LUT, given the logic function of the LUT and a set of inputs defined by the properties listed in Table V-1. In addition to these properties each input has an arrival time, which as explained in Section 2 is the time the input assumes its final value in any given clock cycle.

The logic circuit for the first example is shown in Figure V-2. The inputs to the circuit are a and b , with arrival times of 0 and 1 respectively. The logic function implemented by the LUT is $f(a,b) = a \cdot b$, producing output f . Assuming that $P[a] = P[b] = \frac{1}{2}$, $P_t(a) = P_t(b) = \frac{1}{2}$, $P[a'=0|a=1] = P[a'=1|a=0] = P[b'=1|b=0] = P[b'=0|b=1] = \frac{1}{2}$, and $D(a) = D(b) = \frac{1}{2}$. In this example we wish to compute $P[f]$, $P_t(f)$, $P[f'=0|f=1]$, $P[f'=1|f=0]$ and $D(f)$.

First, we compute the static probability $P[f]$ by applying the following equation:

$$P[f] = \sum_{x=0}^1 \sum_{y=0}^1 (P[a=x] * P[b=y] * f(x,y)) \quad (\text{V-3})$$

Since $f(x,y)$ equals 1 only when $x=y=1$, then $P[f] = P[a=1] * P[b=1] = 1/4$. This is a straightforward computation.

Next we compute the transition probability for signal f , $P_t(f)$. In this case we are concerned with transitions that cause a change in the final output value for a function. In other words, we seek transitions of inputs ab to $a'b'$ such that $(f(a,b) \text{ XOR } f(a',b'))=1$. Thus, if we denote the probability of input transition from ab to $a'b'$ as $P[ab \cap a'b']$ then we can compute $P_t(f)$ as follows:

$$P_t(f) = \sum_{ab=00}^{11} \sum_{a'b'=00}^{11} (f(a,b) \oplus f(a',b')) P[ab \cap a'b'] \quad (\text{V-4})$$

The probability of transition from ab to $a'b'$ can be computed using the static probability, low to high and high to low transition probabilities. This is accomplished by calculating the probability of each input either toggling or remaining unchanged. For example, to compute if inputs ab change from 01 to 11 we compute the probability that a started at 0 and will change to 1 as well as that b started at logic value 1 and will remain at logic value 1. Under the assumption that the inputs a and b are independent, this computation is accomplished as follows:

$$\begin{aligned} P[ab = 01 \cap a'b' = 11] &= P[a = 0] * P[a' = 1 | a = 0] * \\ &\quad P[b = 1] * P[b' = 1 | b = 1] \\ &= (1 - P[a = 1]) * P[a' = 1 | a = 0] * \\ &\quad P[b = 1] * (1 - P[b' = 0 | b = 1]) \end{aligned} \quad (\text{V-5})$$

This computation can be performed for any input transition in a similar manner. Thus, if we use the above method to compute a probability of a specific transition then we will find the transition probability of f , $P_t(f)$, to be 3/8.

The next step is to compute $P[f'=1|f=0]$ and $P[f'=0|f=1]$. In these cases we need only consider the transitions that either cause a 0 to 1 transition on f or a 1 to 0 transition on f , respectively. To compute the 0 to 1 transition probability we apply the same reasoning as when computing $P_t(f)$. There are only two differences: we only consider transitions where $f(a,b)=0$ and $f(a',b')=1$; and we

want to know the probability of such transition knowing that f is already at a logic state zero. To address the second point the Bayes' Rule is applied, and the 0 to 1 transition probability computation is performed as follows:

$$P[f' = 1 | f = 0] = \sum_{ab=00}^{11} \sum_{a'b'=00}^{11} \frac{\left(\overline{f(a,b)} \cdot f(a',b')\right) * P[ab \cap a'b']}{P[f]} \quad (\text{V-6})$$

The term $1-P[f]$ indicates the fraction 0 to 1 transitions constitute in a set of all transitions that start with $f=0$, and thus $P[f'=1|f=0]$ is a measure of the probability signal f will change state given that it is currently at a logic value zero. Similarly, we can compute the 1 to 0 transition probability, using the following equation:

$$P[f' = 0 | f = 1] = \sum_{ab=00}^{11} \sum_{a'b'=00}^{11} \frac{\left(f(a,b) \cdot \overline{f(a',b')}\right) * P[ab \cap a'b']}{1 - P[f]} \quad (\text{V-7})$$

For this example, using equations V-6 and V-7 we can compute $P[f'=1|f=0]$ to be $\frac{1}{4}$ and $P[f'=0|f=1]$ to be $\frac{3}{4}$. These values are consistent with what we know of the behaviour of a 2-input AND gate. Specifically, when the AND gate produces a logic value 1 only when inputs remain unchanged will the output of the AND function remain at a logic value 1. The other three cases, where one or both inputs toggle to zero, the AND function will change its output value to 0. Similarly, only 3 of 12 input transitions will cause the LUT to change output from 0 to 1.

Finally, we can compute the transition density. For this computation we make use of Table V-2, where all possible input transitions for LUT f are shown. In the table the left most column indicates the initial state of inputs a and b , denoted ab . In the second column we show the final state of inputs a and b , denoted $a'b'$. The final column indicates all output transitions on net f , including any glitches that occur. We will refer to the third column as $Trans(ab \cap a'b')$, indicating transitions that occur when inputs change from state ab to $a'b'$.

Up to this point we have not used delay information for LUT inputs. For the transition density computation we will use the arrival time of an input signal to determine the order in which inputs of a LUT change. In our example, we have chosen input a to arrive at time 0 and input b to arrive at time 1. As a result, when inputs change from $ab=01$ to $a'b'=10$, for a single time unit inputs to

Table V-2: Output transition table for function $f=a \cdot b$

Initial State (ab)	Final State (a'b')	# Transitions on f
00	00	0
	01	0
	10	0
	11	1
01	00	0
	01	0
	10	2
	11	1
10	00	0
	01	0
	10	0
	11	1
11	00	1
	01	1
	10	1
	11	0

logic function f are 11, causing the function to evaluate to 1. When input b changes at time 1, the inputs change to 10 and the function output toggles back to 0. This is an example of a glitch and hence two transitions are recorded in the third column for the 01 to 10 input transition.

To compute the transition density, $D(f)$, we have to count all output transitions. To compute transition density we first compute the probability of a transition ab to $a'b'$, $P[ab \cap a'b']$, and multiply it by the number of transitions on output f that occur because of the input transition. We perform the computation similarly when computing the transition probability, $P_t(f)$, except that now we need to account for glitches. Thus, the following equation is used to compute $D(f)$:

$$D(f) = \sum_{ab=00}^{11} \sum_{a'b'=00}^{11} Trans(ab \cap a'b') * P[ab \cap a'b'] \quad (\text{V-8})$$

For our example $D(f)$ is computed to be $\frac{1}{2}$ and we know that the net f can have on average $D(f) \cdot P_t(f) = 2/16$ glitches per cycle, or in other words two extra transitions every 16 clock cycles.

Thus far we have shown how to compute each net property for a gate with no glitches on LUT inputs. However, we need to be able to propagate glitches that appear on the inputs of a LUT to its output. We show how to do this in the following example.

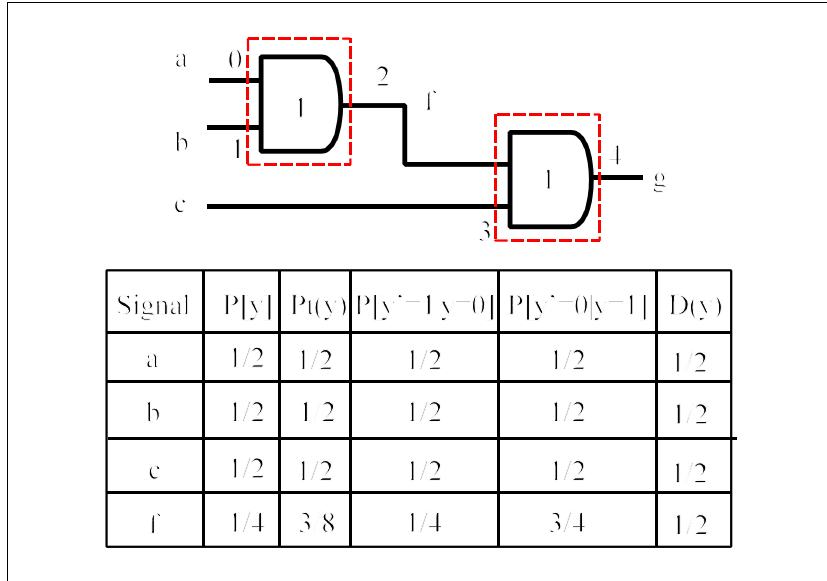


Figure V-3: Logic circuit and signal properties for Example 2

V.3.1.2 Example 2 - Inputs with glitches

Example 1 showed how to compute net properties for a LUT whose inputs are free of glitches. However, this is not always the case and hence it may be necessary to forward the glitches through a LUT. To demonstrate how to accomplish this, consider the example in Figure V-3. Figure V-3 contains the same circuit as in Example 1, except that net f is also an input to another AND gate along with the new input c . The AND gate produces a new output called g . We now want to compute the properties of net g given that the net properties of f are as we computed them in Example 1, properties of input c are the same as those of a and b , and the arrival time of input c is 2.

Except the transition density, the computation of all properties is exactly the same as in Example 1. To compute transition density in the presence of glitches we have to determine how the glitches on an input of a LUT affect the transition density of the output. Since in this particular case the wire f is the only one with glitches, we decompose its transition density into two components: the glitch free component $P_t(f)$; and the glitch component $D_{\text{glitch}}(f)$. Thus,

$$D(f) = P_t(f) + D_{\text{glitch}}(f) \quad (\text{V-9})$$

where $D_{\text{glitch}}(f)$ is the average number of glitches per clock cycle on wire f .

The transition probability component, $P_i(f)$, is responsible for true transitions of wire f . It corresponds to a glitch free input, just like inputs a , b and c . Recall from Example 1 that such waveforms may only cause output glitches due to delay imbalance. Thus, using the arrival time of inputs c and f we can easily compute the transition density for output g , under glitch free input constraint. We denote this transition density component as $D_{\text{glitch-free-inputs}}(g)$.

Computing the glitch transition density component of wire f , $D_{\text{glitch}}(f)$, is a bit more complex. Glitches on wire f can affect the transition density of wire g in three different ways. First, if wire c also contains glitches then for every glitch on wire f we may produce multiple glitches on wire g . Such a case is illustrated in Figure V-4. As we can see, two adjacent glitches on wire c overlap the glitch on wire f causing inputs to the AND gate to be high for two short periods of time, hence two glitches appear on output g . The second case, is where glitches appear on both wires c and f , but neither c or f are high at the same time. Hence, no glitches are produced.

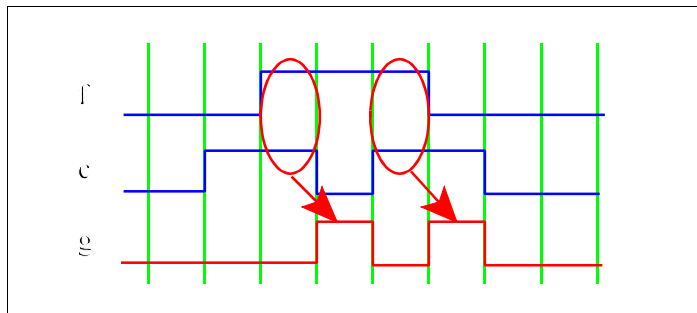


Figure V-4: Example of overlapping glitches

The final case is when the wire c is in a logic state such that any change on wire f causes a change on wire g . In our example this happens when the wire c remains constant at a logic value 1.

From the above three cases, only the first and the third increase the transition density of wire g . However, the contribution of the first case is quite low, because in such a case the resulting glitches would be a very short duration pulses. Such pulses are likely not be sustained by the output of a LUT long enough to fully toggle the state of the output net, thus would not be observed by any logic down stream. Therefore, the only component left to compute is caused by to glitch propagation through a LUT in the third case.

To compute the glitch propagation component of a input through a LUT we utilize the concept of Boolean Difference [Najm93]. The Boolean Difference defines the conditions under

which an input causes a logic function output to toggle. It is defined as:

$$\frac{dy}{dx_i} = y_{x_i=0} \oplus y_{x_i=1} \quad (\text{V-10})$$

If logic function y depends on variables x_1, \dots, x_n , then the Boolean Difference of y with respect to x_i depends on variables $x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n$. When inputs $x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n$ are such that the Boolean Difference evaluates to 1 then for the same set of inputs the logic function y will change value any time input x_i changes value.

The Boolean Difference therefore identifies the set of input $(n-1)$ -tuples for which a function will propagate all glitches on a given input. Thus, to compute the glitch propagation component for wire f , we need to compute the Boolean Difference of function g with respect to f . Since $dy/df = c$, then glitches on wire f are only passed through the LUT when wire c remains 1 for a duration of a clock cycle, or with a probability of $P[c=1] * P[c'=1|c=1] = 1/4$. Thus, the glitches on wire f contribute $D_{\text{glitches-on-inputs}}(g) = 1/4 * D_{\text{glitch}}(f) = 1/32$ to the transition density of wire g . The final transition density of wire g is therefore described as:

$$D(g) = D_{\text{glitch-free-inputs}}(g) + D_{\text{glitches-on-inputs}}(g) \quad (\text{V-11})$$

The $D_{\text{glitch-free-inputs}}(g)$ component is computed as described in Example 1, as though all inputs were glitch-free. The $D_{\text{glitches-on-inputs}}(g)$ is the contribution of glitches on all inputs of LUT g . It can be summarized by the following equation:

$$D_{\text{glitches-on-inputs}}(g) = \sum_{i=1}^n \text{prop}(g, x_i) (D(x_i) - P_t(x_i)) \quad (\text{V-12})$$

where $D(x_i) - P_t(x_i)$ is the average number of glitches on input x_i and $\text{prop}(g, x_i)$ is the glitch propagation factor for inputs x_i in function g , defined by the following equation:

$$\text{prop}(g, x_i) = \sum_{X=0..1} \frac{dg(X)}{dx_i} * P[X \cap X] \quad (\text{V-13})$$

In Equation V-13, \mathbf{X} is the set of inputs $x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n$ and $P[\mathbf{X} \cap \mathbf{X}]$ is the probability all inputs will remain unchanged for the duration of the clock cycle, computed the same way as the probability of a specific transition in Equation V-5. For our example, the transition density of wire g , $D(g)$,

computed as described above is

$$D(g) = \frac{10}{32} + \frac{1}{32} = \frac{11}{32} \quad (\text{V-14})$$

The same result would be obtained if the transition density was calculated for the function $g(a,b,c)=a \cdot b \cdot c$, with the same net properties for inputs a , b and c .

V.3.1.3 Estimation Error

To determine how effective the above method is in estimating transition density for each wire in a circuit, we applied it to a set of benchmark circuits. For each circuit we performed a timing simulation using ModelSIM-Altera 6.0c and obtained transition density values for each net using Quartus II Power Play Power Analyzer. The above method was then applied to compute transition density estimate for each net in every circuit. The transition density estimate was then compared to the transition density values obtained through ModelSIM simulation. The results of the comparison are shown in Figure V-5.

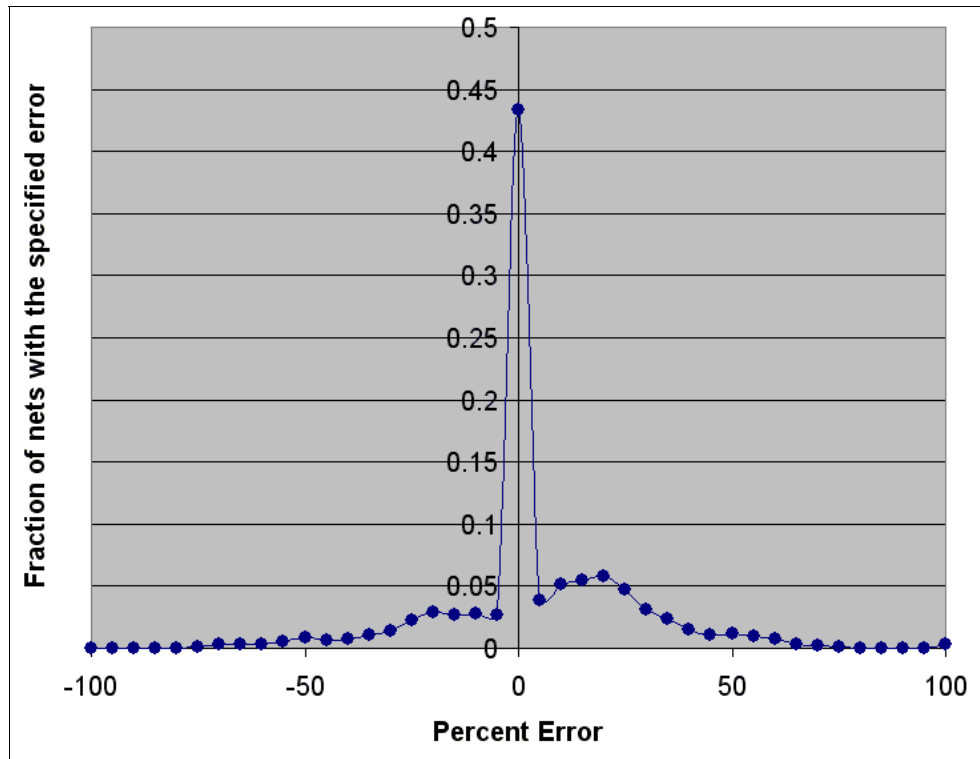


Figure V-5: Transition density estimate error

In Figure V-5 the horizontal axis represents the transition density estimation error in transitions per clock cycle. The vertical axis indicates the fraction of wires for which the specified error occurred. About 75% of all wires have a transition density estimate within 0.05 transitions per cycle from the ones obtained using timing simulation. It is worth noting that in a circuit running at 200MHz a difference of 0.05 transitions per cycle on a single fanout net with delay of approximately 1ns produces a power difference of about 0.01mW. We found this accuracy sufficient to conduct experiments aimed at reducing power dissipation due to glitches.

V.3.1.4 Comments

In the examples above we omitted one parameter that is necessary to compute toggle rate in a real delay model - the *inertial delay*. The inertial delay indicates the minimum duration a pulse can have for the LUT output net to fully toggle logic state. In the method above the inertial delay is only used in the transition density computation when a number of output transitions for any given input transition is computed. Whenever we compute the number of output transitions we perform a local timing simulation of a LUT, using the arrival time of each input as the time the input toggles state. The resulting waveform is then post-processed and any pair of subsequent transitions that occur in an interval of time shorter than the inertial delay are removed. In our experiments we found the value of 0.25ns to work well for Altera Stratix II devices.

While the obtained toggle rate estimation results are promising, it is important to note that this method does not take into account correlation between inputs to a LUT. That is to say, when we compute the probability of transition of wire a (or b) in Example 1, we only considers its historical trend. However, if wires a and b were related in some way, for example state bits of a finite state machine, then $P[a \cap b]$ values could be different and thus contribute to estimation error.

A concurrent work by Cheng *et al.* [Cheng07] presented a similar model for toggle rate computation on FPGAs. However, the work in [Cheng07] does not consider temporal correlation ($P[f'=1|f=0]$ and $P[f'=0|f=1]$). This type of correlation is a significant factor in the computation of toggle rate [Monteiro97]. The approach presented here is further extended in Chapter VI to cover spatial correlation as well.

V.3.2 Net Capacitance Model

The above toggle rate computation technique is only one part of Equation V-2. The other part is the net capacitance, C_i , for a given net. To estimate capacitance we rewrote Equation V-2 as

$$C_i = \frac{2P_{avg}}{f_i V^2} \quad (\text{V-15})$$

The value for V was 1.2V, the toggle rate was obtained from the timing simulation of each circuit, and the power dissipated by the routing of each net was obtained from the Quartus II 5.1 Power Play

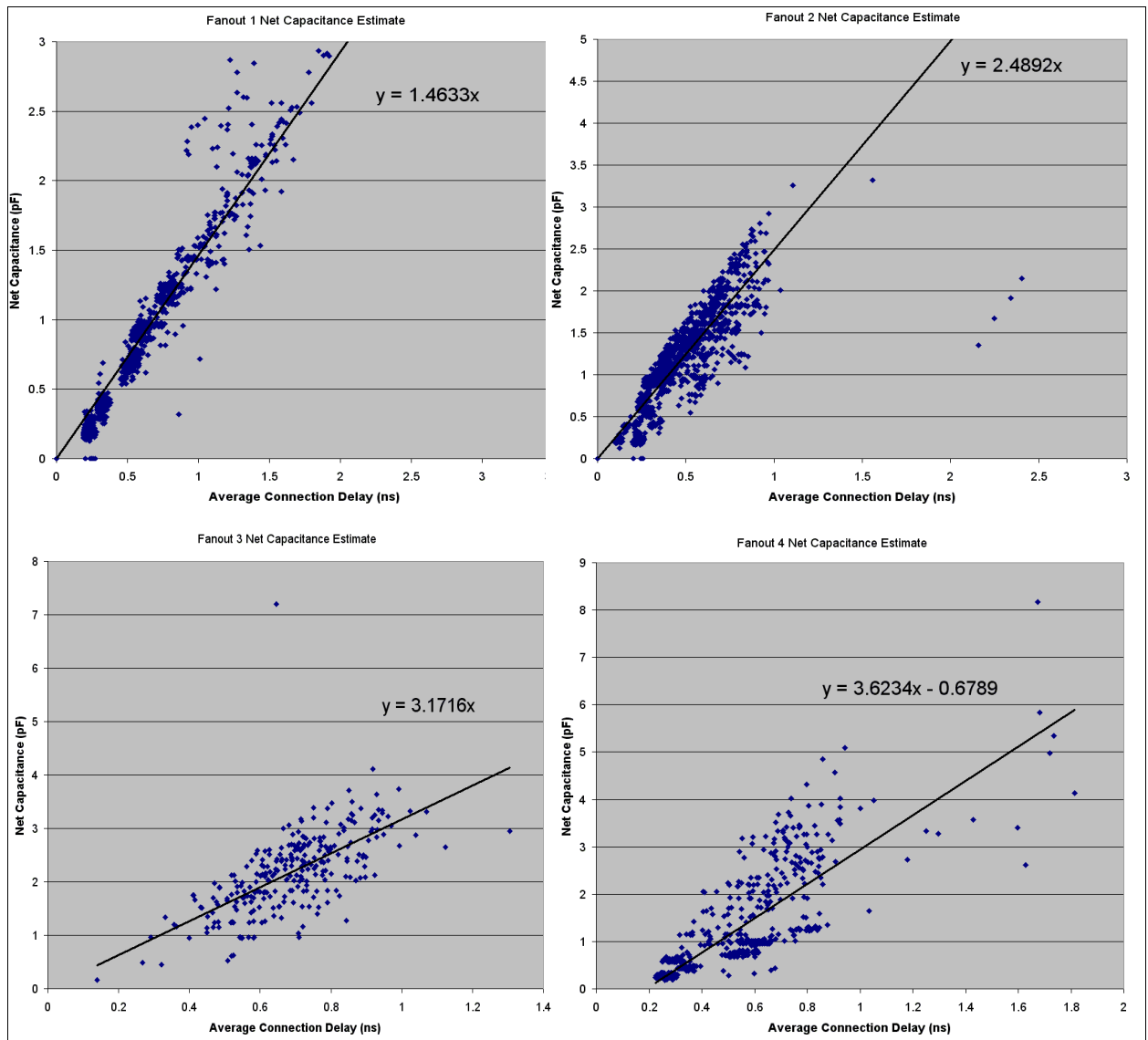


Figure V-6: Net capacitance estimates for nets with fanout 1 (top-left), 2 (top-right), 3 (bottom-left) and 4 (bottom-right)

Power Analyzer. Using the capacitance information obtained this way we modeled the capacitance of a net based on its fanout and the average connection delay. The results are shown in Figure V-6.

Figure V-6 shows graphs with average connection delay on the x-axis and a corresponding net capacitance on the y-axis. These graphs are for nets with fanout 1 through 4, where each graph contains a linear extrapolation of the relationship between average net connection delay and the net capacitance to approximate the capacitance of nets with any fanout and delay. To obtain a net capacitance estimate for a net with fanout $n \geq 5$, the following formula is used:

$$C_{\text{est}}(n) = \left\lfloor \frac{n}{4} \right\rfloor * C_{\text{est}}(4) + C_{\text{est}}(n \bmod 4) \quad (\text{V-16})$$

While this is a simplified recursive estimate, it only applies to nets with fanout higher than 4. In our particular set of benchmarks, fewer than 5% of the nets had fanout higher than 4. We measured the estimated net capacitance error for these nets and found it to be +22%.

V.3.3 LUT Power Model

The final power model discussed in this section is the LUT Power Model. This model

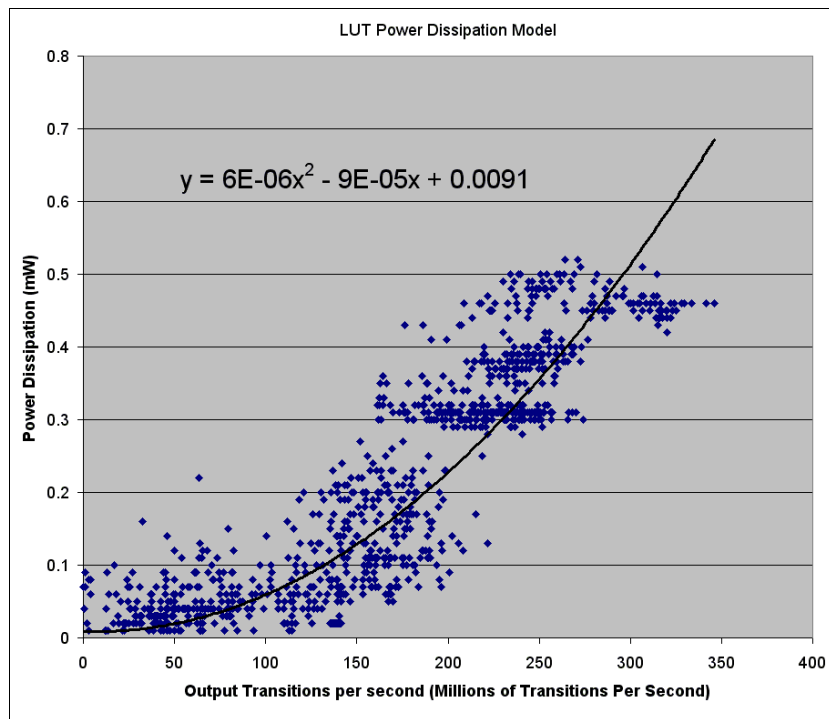


Figure V-7: LUT Power Dissipation Model

represents the dynamic power dissipation due to charging and discharging of capacitors inside of a lookup table. While the general principle of average dynamic power computation is consistent with Equation V-2, in this case the power estimate is represented in terms of the average number of LUT output transitions per unit time. This is because we do not possess sufficient data to recreate the exact capacitance values for each node in the hardware implementation of a Stratix II LUT. By using a model related to transition density we can also determine how much change in dynamic power dissipation will be observed once some of the LUT inputs have their transition densities altered.

To model the dynamic power dissipation of a LUT we used ModelSIM-Altera 6.0c to obtain transition density values in transitions per second and performed power analysis using the Quartus II Power Play Power Analyzer. Using the power analyzer we were able to determine the power dissipated inside each individual LUT and graph it versus the average number of output transitions per second. The graph in Figure V-7 shows that the second order approximation of the average dynamic power dissipated by a LUT is given by the following equation:

$$y = 6.0 * 10^{-6} x^2 - 9.0 * 10^{-5} x + 0.0091 \quad (\text{V-17})$$

V.4 Glitch Reduction

As discussed in the previous section, glitches occur because input signals to a LUT arrive at widely different times. This causes possibly several intermediate transitions to occur before the LUT output stabilizes. We propose to insert negative-edge-triggered flip-flops at an output of a LUT that produces glitches to prevent these spurious transitions from propagating to the routing network and causing subsequent glitches down stream. This technique ensures that during the first half of a clock cycle the output net retains its old value and is updated during the latter half of the clock cycle. The idea is illustrated in the following example.

V.4.1 Negative-Edge-Triggered Flip-Flop Insertion Example

Figure V-8 shows a three level LUT network that starts and ends at positive edge triggered flip-flops. The clock period is 5 and the arrival times of each LUT output are denoted on the right hand side of each LUT. Now suppose that the routing delays are such that input signals x, y and z

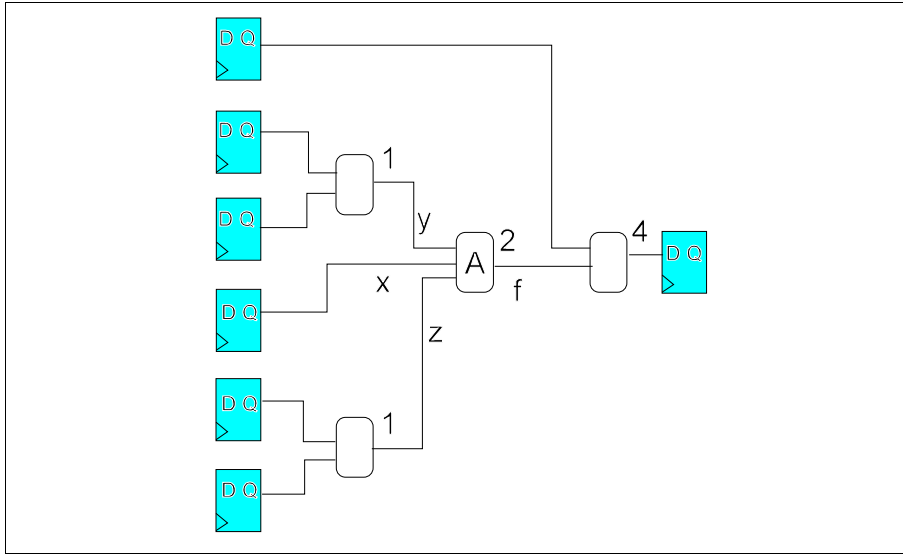


Figure V-8: A three level LUT network

arrive at the inputs of LUT A in order x , y and z . Now suppose that wires x , y and z have values 0, 1 and 0 respectively, and they will all toggle in the current clock cycle. Because a change in signal x will be visible to the LUT before a change in signals y and z , the LUT will initially perceive the input to have changed from 010 to 110 and produce a new output value for wire f . Subsequently, a change in the value of wire y will occur, causing the LUT to again evaluate a new output value for the input sequence 100. Lastly, the wire z will change value and a final output value will be generated for input sequence 101, as shown in Figure V-9. The timing diagram in Figure V-9 shows that the output of LUT A changes value three times, two of which are unnecessary.

To remedy the problem we can insert a negative-edge-triggered flip-flop at the output of LUT A as shown in Figure V-10. From a logical standpoint, the negative-edge-triggered flip-flop will look like a simple wire with timing constraints, and thus will not affect the logical functionality

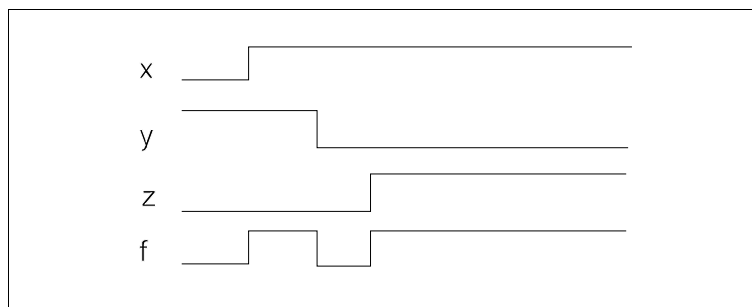


Figure V-9: Timing diagram showing the output behaviour of LUT A given a sample input

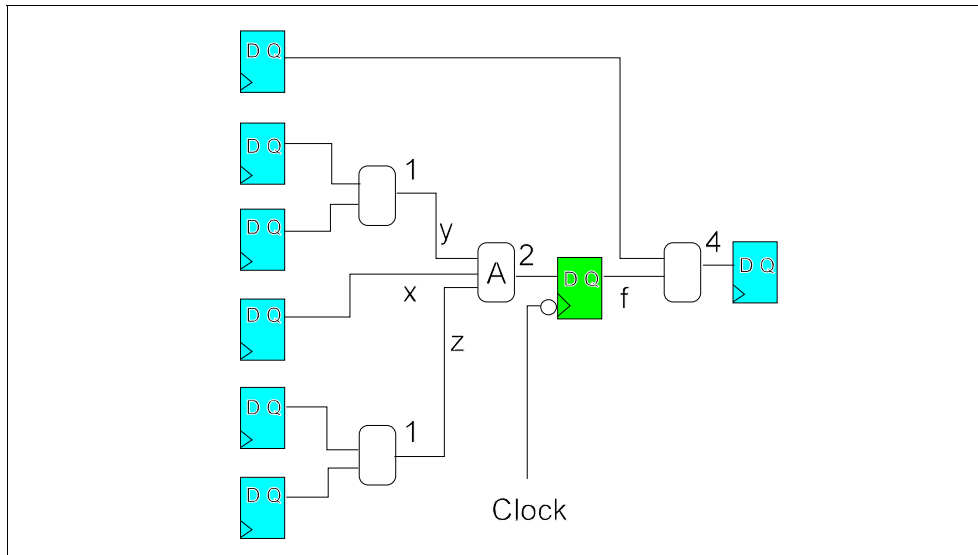


Figure V-10: A three level LUT network with a negative-edge-triggered FF inserted at the output of LUT A

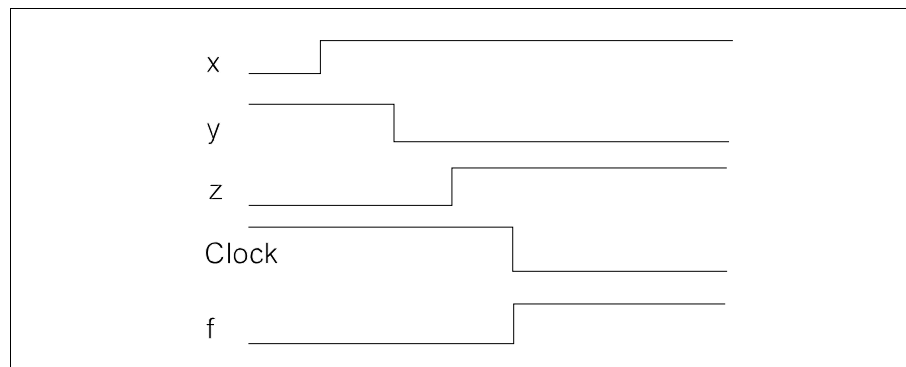


Figure V-11: Activity of signal f after a negative-edge-triggered FF is inserted at the output of LUT A

of the circuit. However, it will hold the state of the given wire during the first half of the clock cycle. At the negative edge of the clock the flip-flop will update its value and toggle at most once. Now wire f , previously driven by LUT A, retains its previous value until the clock signal toggles from one to zero and only changes once at that time. This operation reduces the number of times the output wire toggles by two, thereby reducing the dynamic power dissipation of a circuit. The simulation of the modified circuit is shown in Figure V-11.

V.4.2 Negative-Edge-Triggered Flip-Flop Alternatives

A similar effect can be achieved by using approaches similar to negative-edge-triggered flip-

flop insertion. We considered two such alternatives: a gated D latch, and a gated LUT.

V.4.2.1 Gated D Latch

A gated D latch can be used in a manner similar to a negative-edge-triggered flip-flop. The main difference will be that while the negative-edge-triggered flip-flop toggles only once, a gated D latch will change its state at any time during the latter half of the clock cycle. Figure V-12 illustrates how a negative-edge-triggered flip-flop can be used to reduce the appearance of glitches.

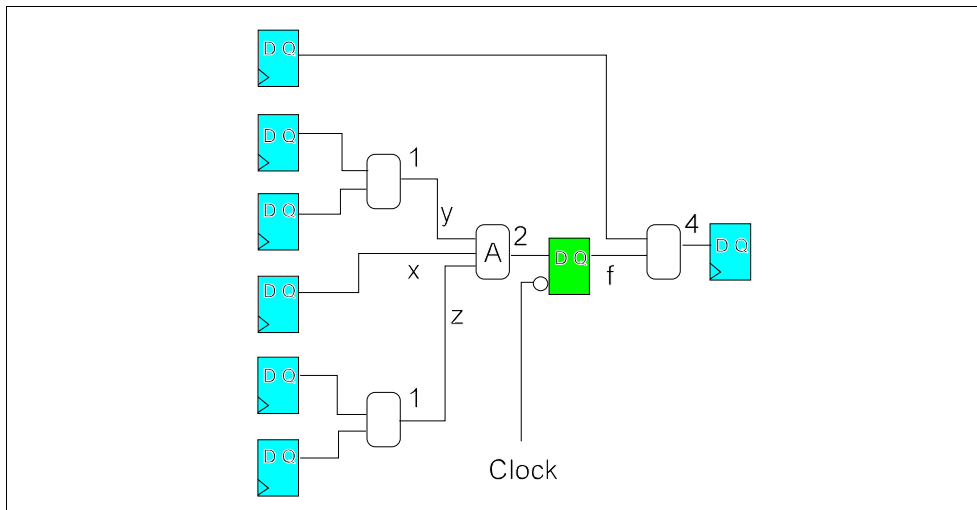


Figure V-12: A three level LUT network with an inserted gated D latch

Figure V-12 shows the same three level network as in Figure V-8, with a gated D latch inserted at the output of LUT A. Because the arrival time of the signal at the output of LUT A is less than half a clock period, the gated D latch will store the final output of LUT A until a negative edge of the clock arrives. The resulting output behaviour is the same as shown in Figure V-11.

V.4.2.2 Gated LUT

A Gated LUT is a LUT whose output is gated with the clock using either an AND or an OR gate. A Gated LUT can be configured to achieve similar glitch reduction to a gated D latch. While a gated LUT will not hold the previous output value during the first half of the clock period, it can force the output to either a zero or one logic value. We demonstrate this in Figure V-13.

In Figure V-13 there are two LUTs, one gated using an AND gate, and the other using an OR gate. In the case of the AND gate the clock input is inverted. This causes the output of the gated LUT

to be at a logic zero state during the first half of the clock cycle. During the latter half of a clock cycle, the LUT output is the controlling input of the AND gate. A similar argument follows for the OR gate.

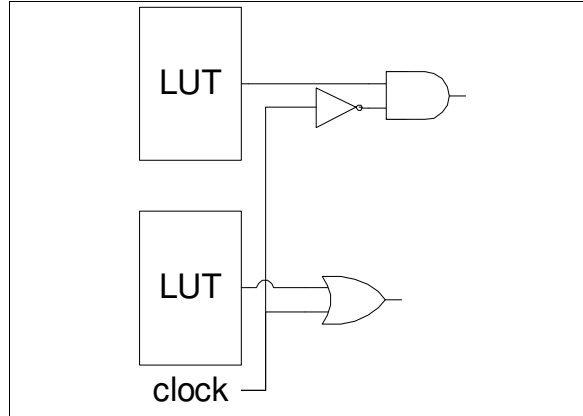


Figure V-13: Examples of Gated LUTs

The downside of using this technique is that additional glitches are produced. For example, when an AND gate is used as the LUT evaluates to a logic 1 in two consecutive clock cycles, additional glitches will appear. The number of glitches per cycle that will be added is equal to

$$D_{\text{added-glitches-AND}}(y) = 2 * P[y = 1]P[y' = 1 | y = 1] \quad (\text{V-18})$$

in the case of an AND gate, or

$$D_{\text{added-glitches-AND}}(y) = 2 * P[y = 1]P[y' = 1 | y = 1] \quad (\text{V-19})$$

in the case of an OR gate.

V.5 Optimization Algorithm

In the previous two sections we described how glitches can be detected and how their presence in a logic circuit can be reduced. We use the aforementioned data to create a physical synthesis optimization algorithm to reduce power dissipated by glitches on an Altera Stratix II device.

V.5.1 The Algorithm

The algorithm is rather straightforward:

1. Scan all nets in a logic circuit to determine if any of the optimization from Section 4 can be applied
2. Analyze the resulting set of nets to determine the benefit of applying the optimization to each net
3. Apply the optimization to a net on which the most power could be saved

The algorithm continuously performs the above three steps until no beneficial choices are found.

The key to the above algorithm is the cost function that determines if an optimization is beneficial.

V.5.2 The Cost Function

The cost function that determines if a glitch reduction optimization on a particular net is beneficial consists of three main components: the power saving, the power cost, and the performance penalty.

The power saving component determines how much power can be saved on a particular net, and its transitive fanout, when the optimization is applied. The power saving for a given net is computed as follows:

$$P_{\text{save}}(y) = \frac{V^2}{2} * C_y * \text{Trans}_{\text{save}}(y) + P_{\text{transitive}}(y) \quad (\text{V-20})$$

where C_y is the capacitance of net y , $\text{Trans}_{\text{save}}(y)$ is the average number of transitions per cycle saved by the optimization, $P_{\text{transitive}}(y)$ is the power saved in the transitive fanout of net y , and V is the power supply voltage (1.2V). C_y is computed as described in Section V.3.2. The $\text{Trans}_{\text{save}}(y)$ component depends on the applied optimization.

For the gated D latch and the negative-edge-triggered flip-flop insertion the number of transitions saved is defined by

$$\text{Trans}_{\text{save}}(y) = D(y) - P_t(y) \quad (\text{V-21})$$

and basically indicates that all glitches are removed from the net. In the case of a gated LUT we use

the same equation, except that we add the glitches generated by the gated LUT structure, as defined by Equations V-18 and V-19.

The $P_{transitive}(y)$ component is computed by calculating how the power dissipated in the transitive fanout of net y is affected by removing glitches from wire y . It is done by summing the change in power dissipation of each LUT and its output net using models described in Section V.3.

For the optimization to be effective the power saving associated with the transformation must outweigh the power consumed by the added circuit elements. To determine how much power a gated D latch, a negative-edge-triggered flip-flop and a gated LUT consume, we measured their power dissipation with respect to the clock frequency. We measured the power dissipated by each of these components on an Altera Stratix II device using the Quartus II Power Play Power Analyzer. Our results are shown in Figure V-14.

Figure V-14 shows a graph of power dissipation as a function of clock frequency. As shown, the gated D latch takes the most power. This is because to create a gated D latch we needed to use a LUT with feedback, thus the gated D latch implementation is far from optimal in terms of power dissipation. The negative-edge-triggered flip-flop dissipated 33% less power and a gated LUT dissipated about half as much power as a gated D latch.

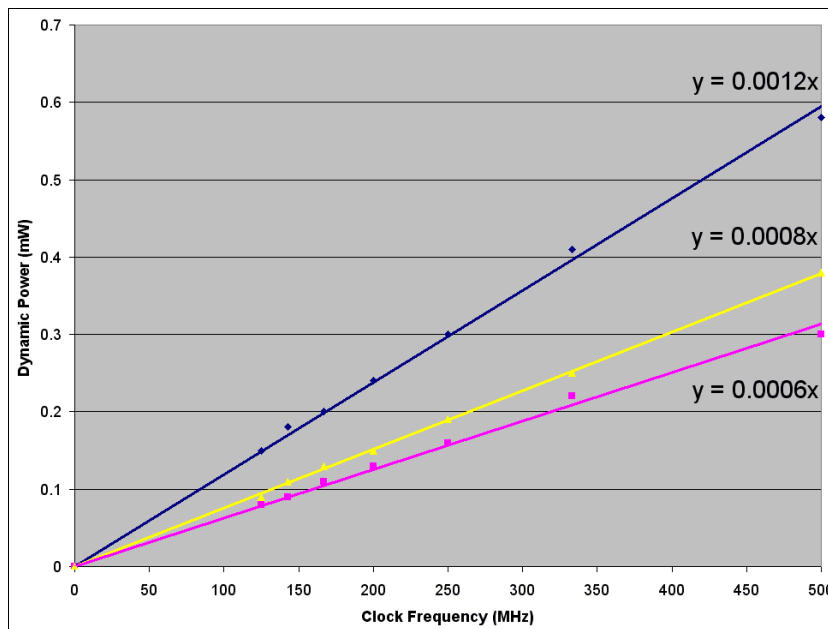


Figure V-14: Power dissipation of a gated D latch (top), negative-edge-triggered FF (middle), and a gated LUT (bottom)

The final component of the cost function is the delay penalty. The delay penalty arises when we apply an optimization and it changes the delay profile of the circuit. For example, inserting a negative-edge-triggered flip-flop requires that paths, which either start or terminate at that negative-edge-triggered flip-flop, need to compute in less than half of a clock cycle. While we can compute the delay penalty for each net, we allow the user to specify how much delay penalty, Δt , is acceptable in order to save power.

Using the above components we can fully formulate the cost function. In its simplest form it is defined as:

$$\Delta C(y, \Delta t_e) = [1 - u(\Delta t_e - \Delta t)] * [P_{\text{save}}(y) - P_{\text{cost}}(y)] \quad (\text{V-22})$$

where Δt_e is the expected increase in the minimum clock period for the circuit, $u(t)$ is the unit step function, $P_{\text{save}}(y)$ is the power saving component, and $P_{\text{cost}}(y)$ is the cost of applying the optimization. The $\Delta C(y, \Delta t_e)$ function returns 0 if the delay penalty, Δt , is exceeded. Otherwise, the function will return the expected change in power dissipation that will be observed if the optimization is applied.

V.6 Experimental Results

To evaluate the effectiveness of the glitch reduction technique, it is applied to several benchmark circuits distributed with the QUIP 5.0 package [QUIP06] and found on the OpenCores.org website. The following subsections describe the experimental setup, methodology and results obtained.

V.6.1 Setup

The experimental setup consists of 8 circuits compiled (ie. synthesized, placed and routed) using Quartus II [QII51]. For each circuit the target clock frequency was set to 1 GHz and the optimization technique was set to *balanced*. The Quartus II Power Play power optimization option was set to *Normal compilation*, indicating that power optimization is performed as long as the design performance is maintained. All circuits compiled this way were then simulated using ModelSIM-Altera 6.0c. The simulation was performed using a random set of 10000 input vectors and lasted 10000 clock cycles. The simulation step was set to 10ps. The simulation result, in the form of a

Table V-3: Set of benchmark circuits

Circuit Name	ALMs	Minimum Clock Period (ns)	Dynamic Power (mW)
barrel64	337	4.386	229.94
mux64_16bit	608	3.052	389.24
fip_cordic_rca	182	7.551	43.28
oc_des_perf_opt	1343	2.989	1058.8
oc_video_compression_systems_huffman_enc	212	3.626	94.88
cf_fir_24_8_8	1401	5.375	290.41
aes128_fast	2458	6.251	879.24
rsacypher	419	6.376	50.73

Value Change Dump File (.vcd), was then used by the Quartus II Power Play Power Analyzer to determine the dynamic power dissipated by each circuit. The *Enable Glitch Filtering* option was turned ON to enable the power analyzer to account for inertial delay of logic components in a circuit.

Table V-3 summarizes area, delay and power statistics of each benchmark circuit. The first column lists the circuit name, the second its area in terms of Adaptive Logic Modules (ALMs), the third shows the minimum clock period for each circuit and finally the fourth column shows the dynamic power dissipation of each circuit, excluding the I/O dynamic power. The I/O dynamic power is excluded as most of the circuits are small. Most of these circuits would be sub-circuits in a larger design and hence their I/O pins would become internal wires in the large design.

V.6.2 Methodology

The aforementioned optimizations were applied in the context of physical synthesis. Each circuit was first read from Quartus II, using QUIP 5.0, to perform toggle rate analysis as described in Section V.3.1. Then, the algorithm from Section V.5 was applied to each circuit, allowing only a 5% delay penalty. The resulting circuit was timing analyzed by Quartus II and simulated using ModelSIM-Altera 6.0c. In each case the same simulation clock frequency was used as before the optimization, ensuring that the clock frequency was lower than the maximum clock frequency for both the original and the modified circuit. The simulation results were used to obtain dynamic power estimate using Quartus II Power Play Power Analyzer. The results obtained are described in the following section.

Table V-4: Final Results

Circuit Name	ALMs		Simulation Frequency (MHz)	Minimum Clock Period		Dynamic Power	
	Final	Change (%)		Final (ns)	Change (%)	After (mW)	Change (%)
barrel64	342	1.46	200	4.806	8.74	189.7	-17.50
mux64_16bit	608	0	275	3.052	0	389.2	0.00
fip_cordic_rca	182	0	125	7.851	3.82	39.49	-8.76
oc_des_perf_opt	1356	0.96	290	3.07	2.64	796.7	-24.75
oc_video_compression_syst ems_huffman_enc	214	0.93	260	3.626	0	95.19	0.33
cf_fir_24_8_8	1401	0	170	5.71	5.87	292.9	0.84
aes128_fast	2514	2.23	140	6.569	4.84	870.6	-0.99
rsacypher	419	0	140	6.563	2.85	48.22	-4.95
Average		0.7			3.6		-7.00

V.6.3 Results

We applied the glitch reduction optimization to the 8 benchmark circuits. The results we obtained are summarized in Table V-4. Table V-4 shows the name of each circuit in column 1, the final size of the circuit in terms of logic cells in column 2, the minimum clock period in column 5, and the dynamic power dissipated by the circuit in column 7. We compare these results to the statistics we listed in Table V-3. The comparison is shown in columns 3, 6 and 8 for ALM count, maximum clock frequency and dynamic power dissipation, respectively. The percentage change in columns 3, 6 and 8 was computed using the following formula:

$$\%difference = 100 \frac{final - initial}{\max (final, initial)} \quad (\text{V-23})$$

The results in Table V-3 show that inserting negative-edge-triggered flip-flops into a circuit is an effective way of reducing the impact of glitches in a logic circuit. We see a 7.0% reduction in dynamic power dissipation, verified by a commercial tool through simulation, at a cost of 3.6% increase in minimum clock period. The change in the ALM count is negligible.

V.6.4 Discussion

In general, the insertion of a negative-edge-triggered flip-flop is successful in reducing the number of glitches in a circuit at a cost of reduced clock frequency. We obtained good power reduction results in most of the circuits tested, ranging from 1% to 25% in dynamic power reduction.

The power saving comes at a cost of circuit delay. The delay increase is due to the delay introduced by inserting a negative-edge-triggered flip-flop. Another factor is the majority of glitches were found on near-critical paths because they generally consisted of the largest number of LUTs. However, for a negative-edge triggered flip-flop insertion to be effective, a target LUT must have the output arrival time less than half a clock period and the required time above half a clock period. For near critical paths this margin is small, thus it is difficult to find suitable candidates for the optimization without incurring a delay penalty.

The largest power reduction was observed in the *oc_des_perf_opt* circuit. This circuit is an implementation of a Data Encryption System that was optimized for performance on Altera devices. The circuit contains a large number of XOR gates with a large number of unbalanced paths. Because an XOR gate allows all glitches to propagate through it, removing glitches on one net causes a large number of glitches to disappear from nets in its transitive fanout. The delay penalty was minor.

The second largest power reduction was observed in the *barrel64* circuit. It is essentially a barrel shifter circuit, rather simple in its design. In this circuit 56 negative-edge triggered flip-flops were added to produce a 12.7% power reduction. A reduction of 17.5% in power was also possible in this circuit, when a delay penalty of 8.7% was allowed. Given that most FPGA circuits run at speeds under 200 MHz, sacrificing more speed in this circuit may be acceptable. With this result we attained an average power reduction of 7% at a minimum clock period increase of 3.6%. Other circuits that operated at above 200 MHz did not produce large power savings when a larger (10%-15%) delay penalty was allowed.

The power reduction of 1% was observed on the *aes128_fast* circuit, which is another cryptographic circuit. A detailed analysis of this circuit showed that we saved 34.14mW (9.2%) of dynamic power in the routing alone. However, the saving came at a cost of 24.6mW of power for adding 173 negative-edge-triggered flip-flops and 1.86mW of power for routing the clock signal to Logic Array Blocks (LABs) in which a clock signal was not needed previously. While the

optimization was successful in reducing the toggle rate of each wire to which it was applied by 50-70%, the capacitance on most of these nets was too low to affect the overall dynamic power in a significant way. In fact, most of the net connections in this circuit were realized through short wire inside of a LAB. This is an excellent example of how power dissipation was reduced not due to toggle rate reduction, but rather by using wires with lower capacitance.

In two circuits, namely *cf_fir_24_8_8* and *oc_video_compression_systems_huffman_enc*, the power dissipation increased, though not significantly. In these circuits the toggle rate was overestimated. However, the overestimate was not significant enough to cause a large power dissipation increase.

Finally, the *mux64_16bit* circuit saw no power, area or speed change. The reason for this rests in the very low presence of glitches. We were able to estimate the toggle rate of all wires almost perfectly and since the circuit consisted of primarily short wires, the cost function of the power optimization algorithm did not recommend negative-edge-triggered flip-flop insertion for any wire in the circuit.

In general, this optimization is successful. It has an advantage over pipelining in that the latency of the circuit remains constant. Also, because the insertion of a negative-edge triggered flip-flop does not affect the logic of the circuit, this technique can be used in tandem with other power optimization techniques, including pipelining and retiming. In such cases it would be advantageous for the optimization to be applied after pipelining and retiming as nets in a retimed/pipelined circuit will have a different delay profile, thus glitches would appear on different wires.

V.6.5 Related Works

The works in [Leijten95] and [Monteiro93] are the closest to our approach. The problem of glitches is discussed in both papers and the research presented there indicates that retiming is a good way to rebalance path delays and at the same time reduce the appearance of glitches. While the general idea is similar, that is to insert flip-flops at glitchy nodes, the process of flip-flop insertion is quite different.

A retiming algorithm rebalances path delays and maintains circuit latency. This is accomplished by pushing flip-flops from the output of a LUT to its inputs (push-back), or vice-versa

(push-forward). The push-back operation removes a single flip-flop from the output of a LUT and adds a flip-flop on each net driving the LUT to preserve circuit latency. However, if one of these nets has fanout greater than 1, then it is necessary to apply push-back operation to all LUTs that net drives. This can potentially affect an entire circuit. A similar effect occurs in a push-forward operation.

On the other hand, the algorithm presented here only needs to ensure that any flip-flop-to-flip-flop path contains at most one negative-edge-triggered flip-flop. The process of insertion therefore becomes a local operation and need not affect the rest of the circuit, and the cost of inserting a negative-edge-triggered flip-flop is limited to the power dissipated by that flip-flop.

V.7 Conclusion

This chapter demonstrated how small incremental changes performed using the Physical Synthesis Toolkit can affect a logic circuit. In this particular case the focus was the dynamic power dissipation of a logic circuit. By looking at the problem post-routing we were able to localize the glitchy nets and reduce their dynamic power dissipation. The algorithm for power reduction in this context was able to achieve an average of 7%, and up to 25%, power reduction on a commercial FPGA device, the Altera Stratix II. The approach presented here showed only a small penalty of 3.6% in delay increase, while having an advantage over pipelining in that the latency of the circuit remains unchanged. Because the insertion of a negative-edge-triggered flip-flop does not affect the logical operation of the circuit, this technique can be used in tandem with other power optimization techniques.

The key feature that made this possible was the ability of the PST to inflict small incremental changes to the circuit post-routing. These changes were implemented in a step-by-step process and allowed the circuit to be fully placed and routed after every change, thereby facilitating the Physical Synthesis flow.

Chapter VI

Estimation of Signal Toggle Rate

In the previous chapter a method for reducing glitches in designs implemented on a commercial FPGA was presented. It showed that localizing glitches accurately can lead to substantial power savings. However, the approach to detect glitches presented in the previous chapter did not account for spatial correlation of logic signals. While it is not a problem in circuits where reconvergent fanout constitutes a small portion of the circuit, we have seen through examples, as well as prior works published on the topic [Schneider96b], that neglecting correlation may lead to increasing power dissipation.

In this chapter a new approach to vectorless toggle rate estimation is presented. It is based on the work in Chapter V for insight into the propagation of glitches on real FPGAs, and uses the functional decomposition method presented in Chapter IV to efficiently and accurately compute correlation between inputs of a LUT in a logic circuit. The toggle rate estimation technique has been implemented within PST and its effectiveness has been examined on a set of circuit implemented on an Altera Stratix II FPGA.

The results of the work presented in this chapter indicate that in comparison to the method presented in Chapter V, the new method to compute toggle rates on wires in a logic circuit produces results with 3 times smaller average error and approximately 15% reduction in the standard

deviation. When compared to vectorless estimation model available in Quartus II 7.1 [QII71], an improvement of 2 times in the average error and 3 times in the standard deviation is observed. Furthermore, the absolute error distribution shows our method to under estimate the actual toggle rate by 2 transitions in 100 clock cycles on a per wire basis, with a standard deviation of 8 transitions per 100 clock cycles.

VI.1 Introduction

Several works that address the topic of toggle rate estimation already exist [Najm93][Tsui93][Chou94][Schneider96a][Chou96][Monteiro97][Costa97][Marculescu98][Theodoridis00][Anderson03][Hu05]. In the work on transition density [Najm93] the toggle rate is represented using the theory of probability. The main advantage of this approach is that toggle rate can be computed using statistical analysis, producing reasonably accurate results for the overall power dissipation in a logic circuit. However, the accuracy of results on a per connection basis is low, preventing power reduction algorithms from effectively optimizing a logic circuit.

The work in [Anderson03] looks at the transition density (average number of times a signal toggles per unit time) [Najm93] and how this concept applies to circuits implemented on FPGAs. The transition density is represented as a weighted sum of transitions generated and propagated through a LUT. The generated transitions are a function of LUT depth and the number of paths to LUT inputs. Propagated transitions are due to glitches that propagate through a LUT and are based on the logic function the LUT implements.

The work in [Anderson03] was intended to predict the toggle rate of a circuit before it is placed and routed. The work in [Tsui93] focuses on transition density computation once circuit delays are known. Their approach is in essence an event driven simulation, except that logic transitions are represented using a probabilistic waveform. This method effectively combines multiple events on primary inputs of a circuit into a single probabilistic waveform to obtain transition density values.

A probabilistic model introduced in [Schneider96a] accounts for both temporal and spacial correlation in a circuit. Using Reduced Ordered Binary Decision Diagrams (ROBDDs) to perform correlation analysis improved the computation time of their method, and the approach scales much

better in terms of processing time than competing approaches presented in [Marculescu98] and [Chou94]. Unfortunately, the method only computes the static and transition probabilities, ignoring glitches caused by delay imbalance in a circuit. In FPGA circuits such glitches can cause the number of transitions on any given wire to double [Anderson03], resulting in an increased power dissipation. Hence, a toggle rate analysis method needs to account for glitching as well as correlation of logic signals.

This chapter presents a fast approach to compute toggle rate on a per wire basis using an XOR-based logic decomposition. It considers temporal and spatial correlations, as well as glitching of logic signals due to delay imbalance in a logic circuit. The proposed approach processes the circuit in topological order computing transition density for each LUT. For LUT inputs that are correlated, the approach performs a Gaussian Elimination based decomposition. It first determines the cones of logic that capture the dependency between these inputs and then decomposes these cones into subfunctions that expose the correlation between them in an efficient manner. The decomposition is then used to account for spacial correlation of LUT inputs.

The approach was tested on several MCNC benchmarks as well as several industrial benchmarks, targeting the Altera Stratix II FPGA. The results show a significant improvement of the approach currently available in the most recent release of Quartus II.

VI.2 Background

This section presents the background information related to toggle rate analysis. Some basic information related to the basics of power dissipation and toggle rate analysis already presented in Chapter V is repeated. In addition, a more detailed description of prior works in this field is given.

VI.2.1 Power Dissipation and Toggle Rate Analysis Review

The average dynamic power dissipation of a logic circuit is defined by the following equation:

$$P_{avg} = \sum_{\text{all nets } i} (f_i C_i V^2) \quad \text{(VI-1)}$$

where f_i is the transition density measured on a per second basis, C_i is the net capacitance, and V is

the power supply voltage. In a circuit where the primary inputs are registered, we can expand f_i as $s_i * f_{clock}$, where f_{clock} is the clock frequency and s_i is the per cycle transition density of a net. The transition density is a key parameter in this equation as it changes depending on the logic function of a component that drives it, as well as logic functions in its transitive fanin, logic and wire delays, as well as glitches.

The computation of transition density involves the use of several parameters: static probability, 0→1 transition probability and 1→0 transition probability. *Static probability* defines the likelihood that a logic signal is at a logic value of 1. The *0→1 transition probability* determines how likely a signal is to change state once it is at a logic 0 value. Similarly, the *1→0 transition probability* accounts for the likelihood of a signal changing state when it is at a logic 1 value. The latter two parameters are commonly referred to as the *temporal correlation* of a signal. While these three parameters are necessary to compute transition density for any given logic function, each of them is affected by what is known as *spatial correlation* [Schneider96b].

Spatial correlation is a measure of the relationship between logic signals at an input of a logic gate. This occurs when logic expressions for each of the inputs are related by common set of signals, either primary inputs or an internal signal. In both cases a particular logic state of one input affects the static probability of the other inputs.

VI.2.2 Existing Toggle Rate Estimation Techniques

A number of methods to estimate toggle rates in logic circuits exist. Most of them are designed to work with circuits formed from a set of 2-input gates, and few deal with toggle rate issues for FPGA devices. In this section a spectrum of works in this field are presented.

Najm's [Najm93] work on toggle rate estimation introduced the idea of transition density as a statistical measure of transitions that occur on any given wire. The idea of transition density is that a logic module propagates transitions from its input x_i to its output y_j only when the logic function defining y_j is sensitive to changes in x_i . More formally, transitions are propagated when:

$$\frac{\partial y_j}{\partial x_i} = y_j \Big|_{x_i=0} \oplus y_j \Big|_{x_i=1} = 1 \quad \text{(VI-2)}$$

where y_j is a function of x_i . The Boolean Difference in Equation VI-2 is equal to 1 only when the

other inputs of function y_j are such that x_i becomes the controlling input of the logic function. When inputs to y_j are independent then the transition density of the wire representing logic function y_j can be defined as a sum of transition densities propagated from each of the function inputs to its output y_j . More formally, the transition density is computed as:

$$D(y_j) = \sum_{i=1}^n P \left[\frac{\partial y_j}{\partial x_i} \right] D(x_i) \quad (\text{VI-3})$$

where n is the number of inputs to y_j .

The strength of the above approach is in its ease of computation. With the use of BDDs, Equation VI-3 can be computed rapidly. Unfortunately, this approach has weaknesses. First, it does not account for relative timing of input signals. As demonstrated in Chapter V, the delay imbalance between logic function inputs affects transition density. Second, the estimate error is large when computing the transition density of individual gates. Thus, when computing the transition density of individual wires in a logic circuit on a gate by gate basis, this approach will incur significant error.

The work of Tsui *et al.* [Tsui93] utilized the concept of probabilistic waveforms, originally introduced by Najm [Najm90]. The main idea in their approach was to replace regular input patterns with time waveforms that indicate the static and transition probability at particular time indices, as well as correlation coefficients. These waveforms are propagated through the circuit by passing probabilistic events through logic gates in the circuit, taking logic delays into account. The advantage of this approach is its ability to compress the data required for toggle rate computation, as well as include propagation delay. However, transitions that occur in close proximity to one another are not eliminated in their approach. This leads to overestimation of glitches in a circuit, as the short pulses that are filtered out due the RC component of the gate output and the subsequent routing. Effectively, any near-simultaneous transitions are considered to cause the gate output to experience a full voltage swing. The works of Ding *et al.* [Ding98] and Hu and Agrawal [Hu05] introduce a filtering mechanism to address this issue, though it further increases the computational complexity of the approach.

Schneider *et al.* [Schneider96a] address the toggle rate computation considering both temporal and spacial correlation. The main contribution of their work was the use of BDDs to

compute correlation. The authors create a BDD to represent a logic function for any given node. The BDD is then expanded to encompass all possible transitions for each variable. This effectively doubles the number of variables in a BDD. Although BDDs in their method grows quickly, their method has been shown to be faster than that of Chou *et al.* [Chou94][Chou96] and Marculescu *et al.* [Marculescu98]. However, Schneider *et al.* assume that 0→1 and 1→0 transition probabilities are the same and ignore glitching due to the delay imbalance, thus introducing errors in the transition density computation.

In the work of Ghosh *et al.* [Ghosh92] and Monteiro *et al.* [Monteiro97] the toggle rate analysis is addressed by using symbolic simulation. The basic idea of the approach is to encompass all possible values and transitions for every gate in the circuit. However, to account for spacial correlation this method requires the creation of a symbolic network, which for practical purposes can become large, even with the use of BDDs. It has been shown by Costa *et al.* [Costa97] that the method becomes time and memory intensive quite rapidly.

The work of Marculescu *et al.* [Marculescu98] is an example of using correlation coefficients to account for correlation between logic signals. In this particular work, the authors define transition coefficients as follows:

$$TC_{ij,kl}^{xy} = \frac{p(x_{i \rightarrow k} \cap y_{j \rightarrow l})}{p(x_{i \rightarrow k}) p(y_{j \rightarrow l})} \quad (\text{VI-4})$$

The intuition behind this approach is to relate in numerical terms the likelihood of both signals x and y changing state to the likelihood of independent transitions of the same signals. In the case of uncorrelated signals the transition coefficient is equal to 1. The computational complexity is introduced because for any pair of signals x and y , 16 such coefficients need to be computed.

While many more works exist, the above works give a good cross-section of approaches employed in the field.

VI.2.3 Important Aspects of Toggle Rate Computation

The aforementioned works represent a wide range of works on the subject of toggle rate prediction and power estimation. These works clearly highlight several key aspects necessary to

compute toggle rate of signals in a logic circuit. They are:

1. temporal correlation,
2. spatial correlation,
3. circuit delays,
4. glitches, and
5. computational efficiency.

As evidence by the body of work on the subject matter, the most arduous task is to create an algorithm that is both computationally efficient and accounts for spatial correlation. The main problem is to include possibly large logic expressions to represent correlated logic signals and quickly determine the nature of the relationship between them.

To address the computational complexity problem, this chapter presents an approach that simplifies the computation of correlation by exploiting a fast XOR-based logic decomposition based on Gaussian Elimination. The basic idea behind this decomposition was discussed in Chapter IV. As demonstrated there, the use of Gaussian Elimination in conjunction with BDDs makes it an effective approach to decompose large logic functions.

The following sections describe how spatial correlation can be computed using this approach. Unlike prior work, where structures such as BDDs were also utilized, this approach decomposes a large logic function into a few smaller ones. As a result, it is possible to process several small logic functions to compute spatial correlation faster than it would take for a single large logic function to be processed.

VI.3 Computing Toggle Rate

The computation of the toggle rate of a logic signal in an FPGA circuit consists of two components - transitions generated by a LUT and glitches propagated from LUT inputs to its output.

The transitions *generated* by a LUT are those that occur due to changes in logic state of LUT inputs. If LUT inputs change at different times, then glitches may be generated. For example, a 3-LUT with inputs a , b , and c can produce two intermediate values while its inputs change from input pattern 000 to 111. Thus if the LUT produces a logic 0 for inputs 000, logic 1 for 001, logic 0 for 011 and logic 1 for inputs 111, then the LUT output will show 3 transitions, whereas only one should

occur. Furthermore, these glitches may be *propagated* by LUTs downstream. Transitions due to glitches would occur on top of the transitions generated by a LUT and should be added to the toggle rate for a given LUT output.

In the following explanations utilize the concept of transition density to express the toggle rate, measuring transitions on a per clock cycle basis.

VI.3.1 Basic Approach

The basic approach is to process each LUT one at a time in a topological order. For each LUT, we evaluate the static probability, transition probability, transition density, 0→1 and 1→0 transition probabilities and use them in computations downstream. This is done by computing transitions generated by, and propagated through, a LUT.

Transitions generated by a LUT are computed using the static probability for each possible input valuation, the probability of transition from one valuation to another, and the arrival time of each signal at a LUT input. The transition density generated by a LUT is computed as follows:

$$D_g = \sum_{i=0..0}^{1..1} \sum_{j=0..0}^{1..1} \left(P[i] P[j|i] Trans(i, j) \right) \quad \text{(VI-5)}$$

where i and j represent input valuations, and $P[i]$ is the static probability of a given input valuation, $P[j|i]$ is the probability that inputs change from pattern i to j and $Trans(i, j)$ is the number of transitions generated when input pattern changes from i to j . The last term uses the arrival time of each input signal to determine the number of transitions that will occur at the output of a LUT by changing input pattern i to j one bit at a time in the order determined by their arrival time.

Now that the generated transitions are accounted for, we need to compute the transition density component due to glitches propagated through a LUT. To account for the glitches we utilize the concept of Boolean Difference in a fashion similar to [Najm93]. However, we impose a constraint on glitch forwarding such that the glitches from an input x of a LUT are forwarded to its output only during such input transitions that both the initial and the final input valuations are sensitive to changes on input x . Because of this restriction, we need to account for the probability of a particular input transition in the same manner as in Equation VI-5. Thus, the probability glitches

on input x are transferred to the LUT output is computed as follows:

$$P_{fx}(i, j) = \frac{df(i)}{dx} * \frac{df(j)}{dx} * P[i] * P[j|i] \quad (\text{VI-6})$$

where f is the function implemented by the LUT and df/dx is the Boolean Difference of f with respect to LUT input x . The above equation will be equal to $P[i]P[j|i]$ only when both the initial (i) and final (j) input valuations cause function f to change value if input x changes value ($df/dx=1$). Otherwise, the equation will evaluate to 0, signifying that glitches from input x will not propagate to the LUT

$$D_p(x) = \sum_{i=0..1} \sum_{j=0..1} (P_{fx}(i, j) * [D(x) - P_t(x)]) \quad (\text{VI-7})$$

output during the specified transition. Using this definition we can express the transition density due to glitching on a LUT input x as:

where $P_t(x)$ is the transition probability of input x and $D(x) - P_t(x)$ represents the average number of glitches per cycle on input x .

Using the above approach we can define the total transition density for any given wire y as:

$$D(y) = D_g + \sum_{i=1}^n D_p(x_i) \quad (\text{VI-8})$$

While the above computation is relatively simple, its key parameters ($P[i]$ and $P[j|i]$) are affected by correlation [Schneider96b].

VI.3.2 Temporal Correlation

The temporal correlation describes the behaviour of a logic signal with respect to its previous value. It is defined by the 0→1 and 1→0 transition probabilities. These probabilities determine how likely a signal is to change state given its present state. For example, a signal that assumes a logic value 1 for a single clock cycle once every 8 cycles on average, would have a 0→1 transition probability of 1/8, while its 1→0 transition probability would be equal to 1. Several examples of how temporal correlation affects toggle rate computation, and its basic application to spatially uncorrelated signals, were given in Chapter V. In terms of equations used in this chapter, the

temporal correlation affects the $P[j|i]$ parameter, which determines the likelihood of a transition from one input pattern to another.

VI.3.3 Spatial Correlation

The spatial correlation is a more complex type of correlation observed in logic circuits. It is important because it can change both the probability of any given input pattern as well as the probability of input transition.

The basic idea behind spatial correlation is that two, or more, signals may share logic in their transitive fanin. As such, their logic state can be related to a varying degree. An extreme example is one where two inputs, x and y , are related by inversion, in which case neither $xy=00$ nor $xy=11$ are possible input valuations.

Similarly, the spatial correlation may change the probability of input transition. For example, suppose that two inputs to a LUT, x and y , were related in such a way that when $y=1$ then x must equal 1, otherwise x can be either 0 or 1. In such a case we know that transition from $xy=11$ to $xy=01$ is not possible.

In our approach we utilize a fast logic decomposition to efficiently compute spatial correlation. The use of a fast decomposition in our approach makes it possible to speed up the computation of correlation by looking at a collection of small logic functions, rather than a few large ones. While many decompositions exist, the approach presented in Chapter IV has been shown to efficiently handle reasonably large logic functions.

To demonstrate our approach consider the example in Figure VI-1. It shows an example of

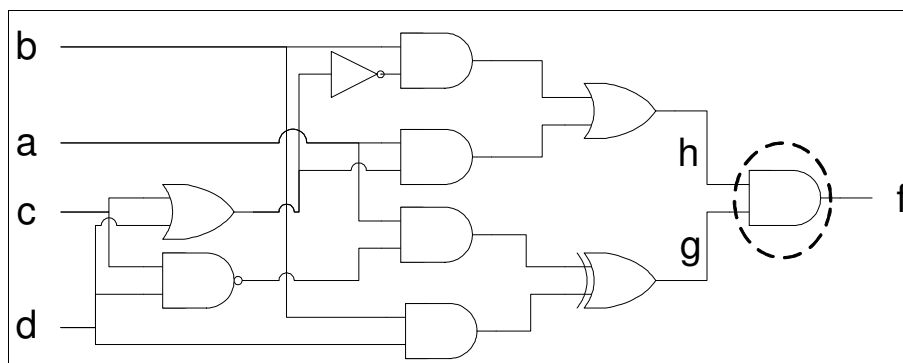


Figure VI-1: Correlated logic cones g and h are used as inputs to an AND gate to form function f

a logic circuit, where the right-most AND gate f depends on two logic functions, g and h , that have the same support. We now show how Functionally Linear Decomposition and Synthesis approach can be used to compute conditional probability of g given h , allowing us to account for correlation of these two logic functions during toggle rate analysis of the AND gate producing signal f (encircled gate in Figure VI-1).

Figure VI-2 shows our initial setup, where logic functions for g and h are put side by side and variables a and b are assigned to the rows of the combined truth table. With this setup in place we apply the decomposition from Chapter IV to find their common subfunctions. They are $\mathbf{X}_1=a$ and $\mathbf{X}_2=b$. The selector functions for \mathbf{X}_1 and \mathbf{X}_2 in function g are $\mathbf{S}_{g1}=c \uparrow d$ and $\mathbf{S}_{g2}=d$. Given this decomposition, our goal is to determine the conditional probability $P[g=1 | h=1]$.

		g				h			
		cd		cd		cd		cd	
ab		00	01	10	11	00	01	10	11
		00	0	0	0	0	0	0	0
	01	0	1	0	1	1	0	0	0
	10	1	1	1	0	0	1	1	1
	11	1	0	1	1	1	1	1	1

Figure VI-2: Truth tables for functions g and h

The basic idea of the following computation is to determine how each of the subfunctions, \mathbf{X}_1 and \mathbf{X}_2 , that cause function g to assume a logic value 1 are affected by the logic state of function h . To do so we first compute when both $\mathbf{X}_1\mathbf{S}_1$ and h are both at a logic value 1 and the probability of such event. This is illustrated in Figure VI-3.

In Figure VI-3 the indicated truth table entries on the left side are the 1s in $\mathbf{X}_1\mathbf{S}_1$ that appear in the same position as 1s in function h . Therefore, for 5 of 8 ones in function h , $\mathbf{X}_1\mathbf{S}_1$ has a corresponding logic 1 entry. Thus, if inputs a , b , c , and d have static probability of $\frac{1}{2}$ and are independent then we can see that $P[\mathbf{X}_1\mathbf{S}_1 | h] = 5/8$. Similarly, we find that $P[\mathbf{X}_2\mathbf{S}_2 | h] = 2/8$.

Now that the simple cases are covered, it is important to account for the fact that both $\mathbf{X}_1\mathbf{S}_1$ and $\mathbf{X}_2\mathbf{S}_2$ could be 1 for the same pattern of inputs a , b , c , and d . In our example this happens for

		X_1S_1				h			
		cd				cd			
		00	01	10	11	00	01	10	11
ab	00	0	0	0	0	0	0	0	0
	01	0	0	0	0	1	0	0	0
	10	1	1	1	0	0	1	1	1
	11	1	1	1	0	1	1	1	1

Figure VI-3: Correlation between X_1S_1 and h

$abcd=1101$. To account for this difference we need to subtract this event from our final result. More formally, we need to subtract $2P[X_1S_1X_2S_2 | h]$, because in this case function g is equal to 0 as a result of using an XOR decomposition. The factor of 2 is also imposed because this event was first included when computing $P[X_1S_1 | h]$ and then again when computing $P[X_2S_2 | h]$.

After accounting for the above terms we find $P[g=1 | h=1]$ to be equal to $\frac{5}{8} + \frac{2}{8} - \frac{2}{8} = \frac{5}{8}$, which we can confirm is the correct result by inspection of Figure VI-2. In a similar manner we can compute $P[g=1 | h=0]$ and thereby have enough information to describe the relationship between static probabilities of these two logic functions.

The above explains how we adjust static probability for any state of inputs of a LUT, but it does not describe how we choose the cones of logic for each input of a LUT. In that respect our approach is consistent with [Schneider96a] as we only look at logic up to a certain depth. While in [Schneider96a] a depth of up to 10 logic gates was used, we utilize a constraint on reconvergent fanout to be within a depth of 4 LUTs and that the cone of logic does not exceed 24 inputs. While in some of the MCNC benchmarks neither constraint is needed due to a relatively small number of inputs, it has been imposed to handle large circuits where the total number of primary inputs exceeds 24.

VI.4 Computing Spatial Correlation

The previous section showed how an XOR-based logic decomposition can be used to compute spatial correlation. In this section mathematical equations used to determine the correlation

between logic signals are derived, taking into account the method in which the logic cones were decomposed.

The derivation begins with Bayes' rule:

$$P[g|h] = \frac{P[g \cap h]}{P[h]} \quad (\text{VI-9})$$

What is needed now is the computation of $P[g \cap h]$. The first step is to expand g as an XOR of products of $\mathbf{X}_{1..n}$ and $\mathbf{S}_{1..n}$:

$$P[g \cap h] = P\left[\left(\sum_i X_i S_i\right) \cap h\right] \quad (\text{VI-10})$$

While we cannot separate $\mathbf{X}_i \mathbf{S}_i$ and h , we can include h within the summation. Doing so allows us to expand the probability of this expression into a sum of probabilities:

$$\begin{aligned} P[g \cap h] = & \sum_i P[X_i S_i \cap h] - \\ & 2 \sum_i \sum_{j=i+1} P[X_i X_j S_i S_j \cap h] + \\ & 4 \sum_i \sum_{j=i+1} \sum_{k=j+1} P[X_i X_j X_k S_i S_j S_k \cap h] - \dots \end{aligned} \quad (\text{VI-11})$$

In Equation VI-11, the number of summations increases by one, until the number of summations is equal to the number of basis vectors functions g and h share. However, it is not necessary to carry out the expansion that far. In this approach the summation is not performed past the triple summation. Exclusion of the remaining terms will show up as an estimate error in the results.

The next step in the derivation is to expand the $P[\mathbf{X}_i \mathbf{S}_i \cap h]$ terms. Notice that the other terms have a similar format, so expansion of the $P[\mathbf{X}_i \mathbf{S}_i \cap h]$ term applies to the other, more complex, terms in the same way. We first expand h as an XOR of product terms:

$$P[X_i S_i \cap h] = P\left[X_i S_i \cap \left(\sum_j X_j R_j\right)\right] \quad (\text{VI-12})$$

where \mathbf{R}_j is a selector function for basis \mathbf{X}_j in function h . Following the same reasoning as in the

expansion in Equation VI-11, we expand the above equation into a sum of probabilities:

$$\begin{aligned}
P[X_i S_i \cap h] &= \sum_j P[X_i S_i \cap X_j R_j] - \\
&\quad 2 \sum_j \sum_{k=j+1} P[X_i X_j X_k S_i R_j R_k] + \\
&\quad 4 \sum_j \sum_{k=j+1} \sum_{l=k+1} P[X_i X_j X_k X_l S_i R_j R_k R_l] - \dots
\end{aligned} \tag{VI-13}$$

The key to simplifying this expression is to note that the basis and selector functions have disjoint support. Assuming independence of distinct variables, we can simplify the above equation to:

$$\begin{aligned}
P[X_i S_i \cap h] &= \sum_j P[X_i X_j] P[S_i R_j] - \\
&\quad 2 \sum_j \sum_{k=j+1} P[X_i X_j X_k] P[S_i R_j R_k] + \\
&\quad 4 \sum_j \sum_{k=j+1} \sum_{l=k+1} P[X_i X_j X_k X_l] P[S_i R_j R_k R_l] - \dots
\end{aligned} \tag{VI-14}$$

In Equation VI-14, the probabilities are computed for functions with smaller support than the union of $\text{sup}(g)$ and $\text{sup}(h)$. As a result, the corresponding probabilities can be computed much faster.

In the above derivation, basis and selector functions have disjoint support and therefore are assumed to be independent. While this assumption may not always hold and we will see errors as a results, to assume otherwise means using logic functions as a function of primary inputs of a circuit, which for is not feasible in practice. However, as we will show through our results, the estimation error is acceptable.

VI.5 Experimental Results

We conducted experiments to evaluate the effectiveness and processing time of the proposed approach on 14 MCNC circuits implemented on the Altera Stratix II FPGA. We used ModelSIM-Altera 6.0c and 10000 random vectors to generate the reference toggle rate for each benchmark circuit. We then ran the Quartus Power Play Power Analyzer [QII71] to determine the actual toggle rate of each wire, accounting for *inertial delay* [Najm93] that would cause some glitches not to appear on a wire.

Table VI-1: Toggle Rate Estimation Results in Comparison to Quartus II 7.1

Circuit	ALUTs	Quartus II 7.1		New Approach			New Approach vs. Quartus II 7.1 (%)	
		Average Error (%)	St. Dev. (%)	Average Error (%)	St. Dev. (%)	Processing Time (s)	Average Error	St. Dev.
alu4	731	23.33	58.39	0.30	22.37	3.00	-98.71	-61.69
apex2	797	55.82	55.26	-2.03	29.49	2.90	-96.36	-46.63
apex4	645	22.01	74.74	-14.65	20.15	2.70	-33.44	-73.04
C2670	117	-8.84	22.56	-1.51	16.10	0.70	-82.92	-28.63
C3450	262	-8.04	45.56	-12.12	20.83	1.50	33.66	-54.28
cps	427	30.33	68.13	-0.14	40.26	1.40	-99.54	-40.91
dalu	193	20.31	53.98	-8.50	20.59	1.10	-58.15	-61.86
ex5p	504	10.25	76.24	-6.53	22.00	3.60	-36.29	-71.14
ex1010	1802	7.88	83.62	-25.47	21.73	31.20	69.06	-74.01
misex3	732	29.80	64.54	-8.06	22.09	2.80	-72.95	-65.77
pair	336	-2.07	32.10	-2.93	11.43	1.90	29.35	-64.39
pdcc	1894	15.25	82.31	-26.57	25.12	30.70	42.60	-69.48
seq	854	44.55	60.92	-3.07	33.34	3.40	-93.11	-45.27
spla	1495	30.84	76.49	-24.13	23.80	21.40	-21.76	-68.88
Average		19.39	61.06	-9.67	23.52		-37.04	-59.00

VI.5.1 Procedure and Results

To generate results we used the transition density and static probability values from simulation to define the behaviour of the primary inputs of each circuit. We then applied our technique to estimate the toggle rate of each benchmark circuit and compared it to the results obtained via simulation.

We also applied the vectorless toggle rate estimation technique available in Quartus II 7.1 to the benchmark suite to see how our method fares in comparison to one currently used in FPGA industry. The input to the Quartus II 7.1 Power Play Power Analyzer was the same as to our approach.

In both cases we compared the estimate to the actual toggle rate for each wire using the following formula:

$$\%difference = 100 \frac{(estimated - actual)}{\max(estimated, actual)} \quad (VI-15)$$

We used the result for each wire in a circuit to determine the average percentage error as well as standard deviation for the error distribution. The results are shown in Table VI-1.

Table VI-2: Toggle Rate Estimation Results in Comparison to approach in Chapter V

Circuit	ALUTs	Chapter V Results for MCNC Benchmarks		New Approach		New Approach vs. Chapter V (%)	
		Average Error (%)	St. Dev. (%)	Average Error (%)	St. Dev. (%)	Average Error	St. Dev.
alu4	731.00	-24.59	29.29	0.30	22.37	-98.78	-23.63
apex2	797.00	-21.34	35.13	-2.03	29.49	-90.49	-16.05
apex4	645.00	-33.48	24.12	-14.65	20.15	-56.24	-16.46
C2670	117.00	-14.72	21.23	-1.51	16.10	-89.74	-24.16
C3450	262.00	-37.65	28.02	-12.12	20.83	-67.81	-25.66
cps	427.00	-19.08	41.71	-0.14	40.26	-99.27	-3.48
dalu	193.00	-28.92	27.35	-8.50	20.59	-70.61	-24.72
ex5p	504.00	-26.67	25.70	-6.53	22.00	-75.52	-14.40
ex1010	1802.00	-42.95	21.72	-25.47	21.73	-40.70	0.05
misex3	732.00	-30.48	26.38	-8.06	22.09	-73.56	-16.26
pair	336.00	-19.13	18.10	-2.93	11.43	-84.68	-36.85
pdcc	1894.00	-42.46	24.68	-26.57	25.12	-37.42	1.75
seq	854.00	-20.78	38.80	-3.07	33.34	-85.23	-14.07
spla	1495.00	-34.09	29.62	-24.13	23.80	-29.22	-19.65
Average		-28.31	27.99	-9.67	23.52	-71.38	-16.69

In Table VI-1 each circuit is listed by name in the left column. In column 2 we list the size in terms of ALUTs the circuit occupied on the Altera Stratix II FPGA. Columns 3 and 4 show the average error and standard deviation for toggle rate prediction made by Quartus II 7.1. In columns 5 and 6 we show the average error and standard deviation for our toggle rate prediction method. In column 7 we show the processing time needed for our approach to analyze toggle rate of each circuit. Finally, in column 8 we compare average error magnitude as computed by our method to Quartus II 7.1, while in column 9 we compare the standard deviation to that obtained by Quartus II 7.1.

In Table VI-2 a similar set of results is presented. This time, the results are compared to those used for toggle rate computation in Chapter V.

VI.5.2 Discussion

The toggle rate estimation technique presented here was able to achieve a -9.67% average error with a standard deviation of 23.52%. While there is room for improvement, the approach proved to be computationally efficient, where the processing time as a function of circuit size grows at a reasonable rate.

In most cases our approach was able to achieve a very good average error. It was particularly

effective in cases where the circuit had a large number of inputs, where expressing LUT inputs in terms of primary circuit inputs was not a viable alternative. Examples of such circuits were *C2670* and *pair*. Other circuits, for which the average estimate error was low, had few levels of LUTs and a low number of primary inputs. An example of such a case is *alu4* and *cps*. For such circuits capturing the reconvergent fanout within four levels of ALUTs was sufficient to either incorporate primary inputs in the computation, or capture enough of the correlation to compute the toggle rate effectively.

This was not the case in three of our circuits, namely *ex1010*, *pdca* and *spla*. Each of those circuits has at most 16 primary inputs. However, what these circuits lack in the number of inputs they make up for in the number of logic levels required to compute the final result. As such, our approach of looking at 4 ALUT levels was insufficient to capture the spacial correlation, though exploring more levels would have lead to encompassing all primary inputs in the computation and producing a better result. Such approach, however, would not have worked well in circuits with larger number of inputs, as the processing time and the cone sizes would be large.

The new approach has been compared to two existing approaches - one used in the latest release of Altera's Quartus II software [QII71], and the approach used in Chapter V. The results of the comparison are presented in Tables VI-1 and VI-2, respectively. In both instances, the new method provides a better estimate of toggle rate.

The result comparison against Quartus II shows that the new method achieved a smaller average error, in terms of magnitude. The new approach underestimates the toggle rate on wires by an average of about 9%, while the approach in Quartus II overestimates the average by approximately 19%, with a standard deviation of 61%. In terms of the average toggle rate error, an overestimate can be viewed in a positive way as it enables estimation of total power for a design. When a power estimate for a design is below the power budget for a given design, and the estimate usually exceeds the actual power dissipation, then the designer can have more confidence that the final implementation of the design will meet desired specification. However, in the context of physical synthesis, where the final goal is to use an estimation technique to perform optimization it is better to slightly underestimate the actual power dissipation on any given wire. The rational behind this argument is that the power saving obtained through an optimization will be underestimated as

well. Thus, if a physical synthesis algorithm determines the optimization to be beneficial, then the optimization will be more successful in reducing power and is much less likely to cause the total power dissipation of the design to increase.

In comparison to the work in Chapter V, the average error has been reduced by approximately a factor of 3. The main reason for the difference is the inclusion of spacial correlation in the new approach. While for the circuits in Chapter V the average error was low, most of those circuits had few reconverging paths within them. Hence an approach that did not utilize spacial correlation was sufficient there. However, when the spacial correlation was present in a logic circuit, we can see that including only temporal correlation in the toggle rate computation is quite insufficient.

In addition to comparing the average error, the standard deviation of the error distribution for each circuit was compared to Quartus II and the approach in Chapter V. In both cases the new approach showed a better standard deviation of the error distribution. This is a definite win of the new approach as the results are much more consistent throughout any given circuit.

On average the standard deviation of the toggle rate prediction error is 23.52%. While this may seem large, the question becomes how significant this error is in terms of absolute estimate error. To determine this we analyzed absolute toggle rate error for the MCNC circuits to find out the

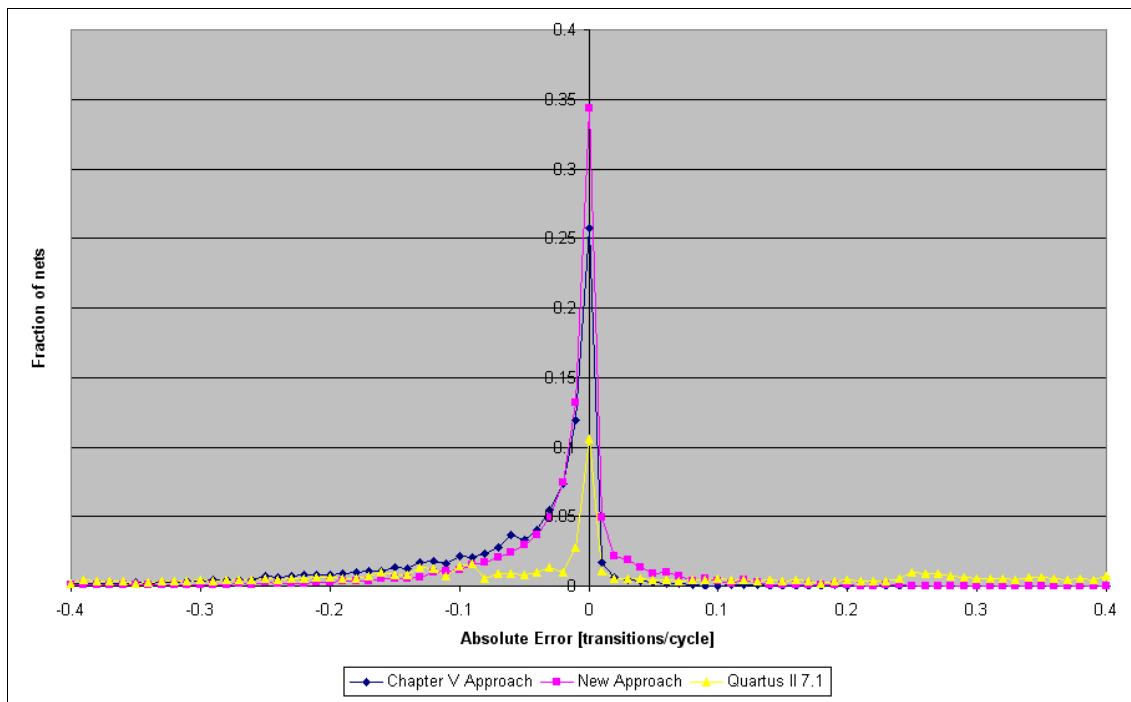


Figure VI-4: Absolute Estimation Error histogram

magnitude of the error we can expect with our approach. The graph in Figure VI-4 shows absolute estimate error achieved by the new approach, as compared to the method in Chapter V as well as Quartus II software. The average error of the new method is -0.024 transitions per cycle and the standard deviation is 0.085 transitions per cycle. In contrast, using the method from Chapter V results in an average error of -0.084 transitions per cycle, and a standard deviation of 0.15. The error observed using the estimation technique in Quartus II shows an overestimate of the transition density by 0.53 transitions per cycle, with a standard deviation of 1.69 transitions per cycle.

VI.6 Conclusion and Future Work

The new approach to toggle rate computation presented in this chapter utilizes a fast XOR-based logic decomposition technique. The decomposition technique from Chapter IV was chosen because of its ability to quickly compute a functional decomposition of logic functions, even for logic cones that exceed 20 inputs. In contrast, other techniques would require more time to process cones of similar size, or the generated decomposition would consist of larger number of subfunctions. This helps to provide a better result in a shorter amount of time.

The method presented here was applied to circuits implemented on the Altera Stratix II FPGA, showing a significant improvement in the accuracy of toggle rate estimating as compared to the one currently used in the Quartus II commercial CAD tool [QII71].

A further area of study along the proposed approach is to further compress the data used in correlation computation. The main idea here would be to preserve only the logic nets that are part of the reconvergent paths, while the other nets could be compressed. This could speed up the correlation computation and allow the cone sizes to be much larger, thereby improving the accuracy of results. This approach could also be used in previously presented approaches as the means to compute correlation coefficients more rapidly.

Chapter VII

Conclusion and Future Work

This dissertation presented several contributions to the field of Computer Aided Design in the context of Field-Programmable Gate Arrays. The contributions include:

1. The development of a new research platform for the purpose of facilitating the advancement of Physical Synthesis algorithms.
2. A novel logic synthesis technique to implement XOR-based logic circuits.
3. A technique to reduce glitches in FPGA devices.
4. A new fast approach to computing toggle rates of wires inside of an FPGA-based logic circuit.

Each of the above contributions have been integrated into a single software package called the Physical Synthesis Toolkit.

The first contribution, the Physical Synthesis Toolkit itself, is to the author's knowledge the first academic software package that allows researchers to perform physical synthesis optimizations, targeting commercial FPGA architectures. It is also the only package to support FPGA devices as advanced as the Altera Stratix II. PST supports both the placement of input and output pins, as well as the full spectrum of hard blocks available on such devices. While it is up to the user to determine how to take advantage of these features, the ability to support memory and digital signal processing

blocks provides an opportunity for the use of more complex and realistic circuits in future research.

The second contribution is a novel logic synthesis optimization that reduces the area of XOR intensive logic circuits. It utilizes the concepts of vector spaces and Gaussian Elimination to find a small subset of logic functions that a logic expression can be decomposed into. In addition to the basic decomposition approach, an efficient variable partitioning algorithm and a multi-output synthesis algorithm are also included. In comparison to the leading academic tools, the approach is fast, able to handle large logic functions and produces logic circuits that are smaller by about 20%.

The third contribution is a glitch reduction technique. The technique is implemented in the context of physical synthesis, and utilizes negative-edge-triggered flip-flops to filter out glitches in a logic circuit. The technique introduces minor disturbance to the logic circuit itself, and the Flip-Flop insertion algorithm ensures that the circuit itself is logically equivalent to the original. A power saving of 7% on the Altera Stratix II was achieved, as measured by Altera's Quartus II Power Play Power Analyzer tool.

The final contribution is the toggle rate estimation technique, which utilizes the decomposition technique presented in this dissertation. The estimation technique covers spatial and temporal correlation of signals, while consuming a small amount of processing time. It compares favourably to the method currently used in Altera's Quartus II CAD tool.

The work presented in this dissertation facilitates further research using the physical synthesis flow. Currently, very few works exist in this context, because most benefits of this new flow are best observed when using commercial FPGAs. This is simply because of the wide variety of components available in commercial FPGAs that can be taken advantage of. The introduction of the Physical Synthesis Toolkit allows researchers to implement their physical synthesis algorithms on commercial devices much more readily and observe gains on real devices.

Another major research path is in the field of logic synthesis. As demonstrated by the decomposition technique in Chapter IV, current synthesis tools have difficulty dealing with XOR based decomposition efficiently. An early attempt of combining currently used approaches with the decomposition presented here showed a promising 10% reduction in area on average. Further study to find an effective way of combining the XOR decomposition and approaches currently in use is a promising research path.

Appendix A – Detailed Synthesis Algorithms

This appendix of the dissertation is devoted to logic synthesis algorithms presented in Chapter IV. The discussion presented here addresses each of the three main algorithms used in the Functionally Linear Decomposition and Synthesis technique, however the notation to describe the algorithms is formalized. While the essence of the algorithms is not altered, a more formal notation will help the readers to reconstruct the algorithms for their own use.

In the sections below the algorithms presented earlier are shown, starting with the variable partitioning algorithm in Figure A-1, followed by basis and selector optimization algorithm in Figure A-2. Lastly, the multi-output synthesis algorithm is described in Figure A-3.

A.1 Heuristic Variable Partitioning Algorithm

In this section of the appendix the heuristic variable partitioning algorithm is described. The description provided in Figure A-1 is a more formal representation of the algorithm than the one shown in Chapter IV.

The algorithm works by incrementally partitioning variables into groups that have 2^k variables, where $2^k < m$. The algorithm begins with a setup of a set of elements $\mathbf{BG}(1)$, where each element is a set consisting of a single variable from the support of function f (line 2). Then, the

```

Partition_Variables( $f, n, m$ ) {
   $\mathbf{BG}(1) = \{\{x_i\} \mid x_i \in \text{sup}(f)\}$ 
  for ( $k=2; k \leq m; k=k*2$ ) {
     $\mathbf{G}(k) = \{a_i \cup a_j \mid a_i, a_j \in \mathbf{BG}(k/2), a_i \cap a_j = \emptyset\}$ 
     $\mathbf{BG}(k) = \emptyset$ 
    While( $\mathbf{G}(k) - \mathbf{BG}(k) \neq \emptyset$ ) {
       $g = \{a_i \in \mathbf{G}(k) \mid \forall a_j \in [\mathbf{G}(k) - a_i], a_i \cap a_j = \emptyset,$ 
         $\text{cost}(a_i) \leq \text{cost}(a_j) \}$ 
       $\mathbf{BG}(k) = \mathbf{BG}(k) \cup \{g\}$ 
       $\mathbf{G}(k) = \mathbf{G}(k) - \{g\}$ 
    }
  }
  If  $m$  is not a power of 2 then {
     $\mathbf{G}(m) =$  all possible combinations of  $\mathbf{BG}(2^i)$  that form a set of  $m$  variables,
    such that  $i^{\text{th}}$  bit of  $m = 1$ 
     $\mathbf{BG}(m) =$  lowest cost grouping from  $\mathbf{G}(m)$ 
  }
  Pick Best Grouping in  $\mathbf{BG}(m)$ .
  Reorder variables in  $f$  to match the best grouping.
}

```

Figure A-1: Heuristic variable partitioning algorithm

algorithm proceeds in a loop (lines 3-12) to create sets $\mathbf{BG}(k)$, where each element is a partition consisting of k input variables. During the first pass of the loop, $k=2$, the algorithm partitions variables into groups of two variables, such that each variables appears in at most one group. Each combination of variables is created and evaluated by determining the number of basis functions created if the given set is used to index rows as well as computing the cost of basis and selector functions. The cost of a function, basis or selector, is determined by the size of it's support plus one, unless the function evaluates to zero in which case the cost is equal to zero.

Once each grouping of two variables is evaluated, the algorithm retains only $n/2$ least cost groupings, keeping in mind that each variable may only appear in one grouping. The procedure then repeats in exactly the same fashion for larger groupings, increasing the number of variables in each grouping by a factor of 2 on each iteration. For example, during the second pass the algorithm creates groupings of four variables using the groupings of size two created previously as a starting point in order to reduce the overall runtime of the algorithm.

In a case where m is not a power of 2 the algorithm proceeds to evaluate all partitions of size m by putting together input variables and groupings generated thus far (lines 13-17). For example, if $m=5$ then for each grouping of size 4 and the algorithm combines it with an input variable to make a grouping of 5 variables. The algorithm then evaluates the grouping and keeps it if it has lower cost than any other grouping of size 5.

The algorithm presented here is a heuristic and as such does not explore the entire solution space in the interest of reducing computation time. The number of partitions tested by our algorithm is a function of bound set size m and number of variables n . When m is a power of 2, the number of

partitions tested is $\sum_{i=0}^{\log_2(m)-1} \binom{2^{-i}n}{2}$ and the algorithm runtime is bounded by $O(n^2)$. The number of

partitions tested is larger when m is not a power of 2. The increase only becomes significant after $m=15$, however we found that after $m=12$ the Gaussian Elimination algorithm begins to slow down significantly. As a result we placed a limit of $m=12$ on the algorithm to prevent Gaussian Elimination from consuming too much processing time.

A.2 Basis and Selector Optimization Algorithm

In this section of the appendix the basis and selector optimization algorithm is described. The description provided in Figure A-2 is a more formal representation of the algorithm than the one shown in Chapter IV.

First, the algorithm computes the cost of each basis and selector function (line 2). The cost is determined by the support set size of each function, plus 1, with the exception of a function equal to 0, for which the cost is 0. The algorithm then proceeds to evaluate possible improvements to basis and selector functions. This is done by putting together a pair of basis-selector function pairs (lines 3-17) to determine if the resulting basis and selector function can replace one of the existing ones and reduce the basis/selector cost. If replacing an existing basis with a new one does not increase the overall cost of implementing a logic function (lines 9-15), then the algorithm replaces one of the basis and selector functions. Otherwise, the algorithm attempts to combine a different set of basis and selector functions.

```

Optimize_Basis_and_Selectors(basis_fs, selector_fs)
Cost =  $\sum_i \text{cost}(\text{basis\_fs}_i) + \sum_j \text{cost}(\text{selector\_fs}_j)$ 
repeat {
  For i=0 to m-2 do
    For j=i+1 to m-1 do
      nb = basis_fs(j) XOR basis_fs(i)
      ns = selector_fs(j) XOR selector_fs(i)
      n_cost = cost(nb)+cost(ns)
      If n_cost ≤ [cost(basis_fs(i))+cost(selector_fs(j))]
        basis_fs(i) = nb
        selector_fs(j) = ns
      else
      If n_cost ≤ [cost(basis_fs(j))+cost(selector_fs(i))]
        basis_fs(j) = nb
        selector_fs(i) = ns
      Update cost
    } until (last 3 iterations produced the same cost)
}

```

Figure A-2: Basis and Selector Optimization Algorithm

The algorithm iterates until the last 3 iterations produce the same basis-selector cost, which indicates that further improvement is unlikely. Note that the algorithm allows for basis-selector replacement even if the cost is unchanged, because it may sometimes be necessary to go through a few intermediate steps before a lower cost basis-selector pair is found.

A.3 Multi-Output Synthesis Algorithm

In this section of the appendix the multiple output synthesis algorithm is described. The description provided in Figure A-3 is a more formal representation of the algorithm than the one shown in Chapter IV.

The first step (lines 2-3) in the algorithm determine if the functions to be synthesized share all variables, or just a subset. If all variables are shared then we can perform a decomposition where the subset of shared variables are used to index the truth table rows (line 5). The decomposition with respect to shared variables will try to do a balanced decomposition so long as the number of basis functions is low in an effort to reduce logic circuit depth. If the number of basis functions becomes large, the number of variables indexing the rows will be reduced to decrease the number of basis

```

Multi_Output_Synthesis(function_set, k) {
  all_vars =  $\{x_i \mid (\exists j)(x_i \in \text{sup}(function\_set(j)))\}$ 
  common_vars =  $\{x_i \mid (\forall j)(x_i \in \text{sup}(function\_set(j)))\}$ 
  if  $\|common\_vars\| == \|all\_vars\|$  then
    Assign common_vars to rows and decompose
  else
    L = function_set
    while ( $\|L\| \geq 1$ ) {
      For all  $\{f_i \in L \mid \text{sup}(f_i) \leq k\}$  do
        L = L -  $\{f_i\}$ , Synthesize  $f_i$  into a  $k$ -LUT
      For  $i=1$  to  $\|L\|$  do  $Q_i = \{f \in L \mid x_i \in \text{sup}(f)\}$ 
      S =  $\{x_i \in all\_vars \mid (\forall j)(\|Q_i\| \leq \|Q_j\|)\}$ 
      F =  $\{\cup Q_i \mid x_i \in S\}$ 
      L = L - F
      If  $\|L\| == 0$  then
        Decompose and synthesize functions in set F
      else {
        Decompose F into basis and selector functions
        Multi_Output_Synthesis(basis,  $k$ )
        L = L  $\cup$   $\{selector\ functions\}$ 
      }
    }
  }
}

```

Figure A-3: Multi-output Synthesis algorithm

functions.

The second step of the algorithm is to perform a decomposition in the case where some variables are shared by all functions and some are not (lines 7-22). We create a list **L** of these functions and perform the next set of steps while the list is not empty. The list will shrink and grow as we synthesize functions, either because of the decomposition, or because the functions with support size less than k will be synthesized into a k -LUT and removed from the list (lines 9-10).

The next set of steps focuses on finding a subset of variables **S** that are used by the fewest number of functions in the set **L** (lines 11-12). We take these functions out of the set **L** (lines 13-14). If the set **L** is empty then all functions in **F** have the same support. We therefore run the variable

partitioning algorithm, decompose and synthesize these functions as shown in Chapter IV, Section 6.2 (line 16). Otherwise, we perform a decomposition where rows are indexed by variables in set **S** (lines 18-20). The decomposition proceeds as described in Chapter IV, Section 6.2, but is limited to the functions in set **F**. The decomposition (line 18) returns a set of basis and selector functions. The basis functions are synthesized separately by recursively calling the multi-output synthesis routine (line 19). The selector functions are placed back into set **L** (line 20) as they may still have variables in common with functions in set **L**.

Appendix B – References

- [ABC05] Berkeley Logic Synthesis Group, *ABC: A System for Sequential Synthesis and Verification*, December 2005 Release. URL: <http://www.eecs.berkeley.edu/~alanmi/abc>.
- [Anderson02] J. H. Anderson and F. N. Najm, "Power-Aware Technology Mapping for LUT-based FPGAs," Proceeding of Field-Programmable Technology Conference, Hong Kong, 2002, pp. 211 - 218.
- [Anderson03] J. H. Anderson and F. N. Najm, "Switching Activity Analysis and Pre-Layout Activity Prediction for FPGAs," ACM/IEEE Workshop on System Level Interconnect Prediction (SLIP), Monterey, CA, 2003, pp. 15 - 21.
- [Altera05a] Stratix Device Family Data Sheet, Altera Corporation, Online resource: http://www.altera.com/literature/hb/stx/stratix_section_1_vol_1.pdf. Last accessed in March 2005.
- [Altera05b] Stratix II Device Family Data Sheet, Altera Corporation, Online resource: http://www.altera.com/literature/hb/stx2/stx2_sii5v1_01.pdf. Last accessed in March 2005.
- [Anton94] H. Anton and C. Rorres, **Elementary Linear Algebra**, Applications Version, 7th Edition, Published by John Wiley & Sons Incorporated, 1994, ISBN 0-471058741-9.

- [Ashenhurst59] R. L. Ashenhurst, "The Decomposition of Switching Functions," Proceedings of an International Symposium on the Theory of Switching, April 2-5, 1957, Harvard University, pp. 74-116.
- [Bahar96] R. I. Bahar, M. Burns, G. D. Hachtel, E. Macii, H. Shin, and F. Somenzi, "Symbolic Computation of Logic Implication for Technology-Dependent Low-Power Synthesis," Proceedings of the International Symposium On Low Power Electronics and Design, Monterey, 1996, pp. 163-168.
- [Betz99] V. Betz, J. Rose, and A. Marquardt, **Architecture and CAD for Deep-Submicron FPGAs**, Kluwer Academic Publishers, 1999, ISBN 0-7923-8460-1.
- [Bryant86] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," IEEE Transactions on Computers, vol. C-35, August 1986, pp. 677-691.
- [Chandrakasan95] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey and R. Brodersen, "Optimizing Power Using Transformations," IEEE Transaction on Computer Aided Design of Integrated Circuits and System, Vol. 14, No. 1, January 1995, pp. 12-31.
- [Chen04] D. Chen, and J. Cong, "DAOmap: A Depth-optimal Area Optimization Mapping Algorithm for FPGA Designs," Proceedings of the IEEE International Conference on Computer Aided Design, San Jose, California, November 2004, pp. 752-759.
- [Chen06] D. Chen, J. Cong, and P. Pan, "FPGA Design Automation: A Survey," Foundations and Trends in Electronic Design Automation, Vol. 1, No. 3, November 2006, pp. 195-330.
- [Cheng07] L. Cheng, D. Chen, and M. D.F. Wong, "GlitchMap: An FPGA Technology Mapper for Low Power Considering Glitches," IEEE Proceedings of the 45th Design Automation Conference, San Diego, California, USA, June 2007, pp.318-323.
- [Cong94] J. Cong, and Y. Ding, "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table based FPGA Designs," IEEE Transactions on Computer Aided Design, Vol. 13, No. 1, Jan. 1994, pp. 1-12.

- [Cong96] J. Cong, J. Peck, and Y. Ding, "RASP: A General Logic Synthesis System for SRAM-based FPGAs," ACM/SIGDA Proceedings of the 4th International Conference of Field-Programmable Gate Arrays, Monterey, California, 1996, pp. 137-143.
- [Chou94] T. Chou, K. Roy, and S. Prasad, "Estimation of Circuit Activity Considering Signal Correlations and Simultaneous Switching," IEEE Proceedings of the International Conference on Computer Aided Design, San Jose, CA, 1994, pp. 300-303.
- [Chou96] T. Chou and K. Roy, "Accurate Estimation of CMOS Sequential Circuits," IEEE Transactions on VLSI, Vol.4, No.3, Sept. 1996, pp. 369-380.
- [Clarke93] E.M. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large boolean functions with applications to technology mapping," Proceedings of the 30th Design Automation Conference, June 1993, pp. 54-60.
- [Costa97] J. C. Costa, J. C. Monteiro, and S. Devadas, "Switching Activity Estimation Using Limited Depth Reconvergent Path Analysis", Proceedings of the International Symposium on Low Power Electronics and Design, 1997, pp. 184-189.
- [Curtis62] H.A. Curtis, "A New Approach to the Design of Switching Circuits," em Princeton, N. J., Van Nostrand, 1962.
- [Ding98] C. Ding, C. Tsui, and M. Pedram, "Gate-level power estimation using tagged probabilistic simulation," IEEE Transactions of Computer Aided Design, vol. 17, no. 11, November 1998, pp. 1099-1107.
- [Ghosh92] A. Ghosh, S. Devadas, K. Keutzer, and J. White, "Estimation of average switching activity in combinational and sequential circuits," Proceedings of the 29th IEEE Design Automation Conference, June 1992, pp. 253-259.
- [Farrahi94] A.H. Farrahi and J. Sarrafzadeh, "FPGA Technology Mapping for Power Minimization," International Workshop on Field-Programmable Logic, 1994, pp.66-77.

- [Hashimoto99] M. Hashimoto, H. Onodera and K. Tamaru, "A Practical Gate Resizing Technique Considering Glitch Reduction for Low Power Design," Proceedings Of the 36th ACM/IEEE Conference on Design Automation, 1999, pp. 446-451.
- [Hu05] F. Hu and V. D. Agrawal, "Dual-Transition Glitch in Probabilistic Waveform Power Estimation," Proceedings of ACM Great Lakes Symposium on VLSI, 2005, pp. 357-360.
- [Hurst85] S. L. Hurst, D. M. Miller, J. C. Muzio, **Spectral Techniques in Digital Logic**, Academic Press, London, 1985.
- [Hwang90] TT Hwang, R. M. Owens, and M. J. Irwin, "Exploiting Communication Complexity for Multilevel Logic Synthesis," IEEE Transactions of Computer Aided Design, vol. 9, No. 10, Oct 1990, pp. 1017-1027.
- [Karpovsky77] M. G. Karpovsky, "Harmonic analysis over finite commutative groups in linearization problems for systems of logical functions," Information and Control, vol. 33, no. 2, February 1977, pp. 142-165.
- [Kim02] K.W. Kim, T. Kim, T.T. Hwang, S.M. Kang, C.L. Liu, "Logic Transformation for Low Power Synthesis," ACM Transactions on Design Automation of Electronic Systems, 2002, pp. 265-283.
- [Kuon07] Ian Kuon and Jonathan Rose, "Measuring the Gap Between FPGAs and ASICs", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) , Vol. 26, No. 2, pp 203-215, Feb. 2007.
- [Lai93] Y. T. Lai, M. Pedram, S. Sastry, "BDD-based Decomposition of Logic Functions with Application to FPGA Synthesis," IEEE Proceedings of the 30th Design Automation Conference, 1993, pp. 642-647.
- [Leijten95] J. Leijten, J. van Meerbergen, J. Jess, "Analysis and reduction of glitches in synchronous networks," Proceedings of the European Design and Test Conference, Paris, 1995, pp. 398-403.
- [Li01] H. Li, W-K. Mak, and S. Katkooi, "LUT-Based FPGA Technology Mapping for Power Minimization with Optimal Depth," IEEE Computer Society Workshop on VLSI, 2001, pp.123-128.

- [Li03] F. Li, D. Chen, L. He, J. Cong, "Architecture Evaluation for Power-Efficient FPGAs," ACM/SIGDA International Symposium on FPGAs, Monterey, CA, 2003, pp.175-184.
- [Lin03] J. Lin, A. Jagannathan and J. Cong, "Placement-Driven Technology Mapping for LUT-Based FPGAs," ACM/SIGDA International Symposium on FPGAs, February 2003, Monterey, California, USA, pp. 121-126.
- [Lou99] J. Lou, W. Chen and M. Pedram, "Concurrent Logic Restructuring and Placement for Timing Closure," Proceedings of the 1999 ACM/IEEE International Conference on Computer Aided Design, November 1999, pp. 31-35.
- [Mahapatra00] N. R. Mahapatra, S.V. Garimella, A. Takeen, "Efficient techniques based on gate triggering for designing static CMOS ICs with very low glitch power dissipation", Proceedings of the International Symposium on Circuits and Systems, 2000, vol. 2, pp. 537-540.
- [Marculescu98] R. Marculescu, D. Marculescu, and M. Pedram, "Probabilistic Modeling of Dependencies During Switching Activity Analysis", in IEEE Transactions on Computer Aided Design, Vol.12, No.2, Feb. 1998, pp. 73-83.
- [Monteiro93] J. Monteiro, S. Devadas and A. Ghosh, "Retiming Sequential Circuits for Low Power," IEEE Proceedings of International Conference on Computer Aided Design, 1993, pp. 398-402.
- [Monteiro97] J. Monteiro, S. Devadas, A. Ghosh, K. Keutzer, and J. White, "Estimation of Average Switching Activity in Combinational Circuits Using Symbolic Simulation," IEEE Transactions on Computer Aided Design, Vol. 16, No. 1, Jan 1997, pp. 121-127.
- [Najm90] F. N. Najm, R. Burch, P. Yang, and I. N. Hajj, "Probabilistic simulation for reliability analysis of CMOS VLSI circuits," IEEE Transactions on Computer Aided Design, April 1990, vol. 9, pp. 439-450.
- [Najm93] F. Najm, "Transition Density: a new measure of activity in digital circuits," IEEE Transactions on Computer Aided Design of Integrated Circuits and

Systems, February 1993, Vol. 12, pp. 310-323.

- [Najm95] F. N. Najm, "Tutorial: Feedback, Correlation, and Delay Concerns in the Power Estimation of VLSI Circuits", IEEE Proceedings of the 32nd Design Automation Conference, 1995, Issue 32, pp. 612-617.
- [Perkowski94] M. Perkowski and S. Grygiel, "A Survey of Literature on Function Decomposition, " A Final Report for Summer Faculty Research Program, Wright Laboratory, Sponsored by Air Force Office of Scientific Research, Bolling Air Force Base, DC and Wright Laboratory, September 1994.
- [Pradhan96] D. K. Pradhan, M. Chatterjee, M. V. Swarna and W. Kunz, "Gate-Level Synthesis for Low-Power Using New Transformations," Proceedings of International Symposium on Low Power Electronics and Design, California, 1996, pp. 297-300.
- [QII51] Altera Quartus II version 5.1.
- [QII71] Altera Quartus II version 7.1.
- [QUIP06] Altera QUIP 5.0 package. URL: <http://www.altera.com/education/univ/research/unv-quip.html>. Last Accessed March 1, 2006.
- [Sasao94] T. Sasao and J. T. Butler, "A Design Method for Look-up Table Type FPGA by Pseudo-Kronecker Expansion," Proceedings of the 24th International Symposium On Multi-Valued Logic, 1994, pp. 97-106.
- [Sasao99] T. Sasao, **Switching Theory for Logic Synthesis**, Kluwer Academic Publishers, 1999, ISBN 0-7923-8456-3.
- [Schneider96a] P. H. Schneider, U. Schlichtmann, and B. Wurth, "Fast Power Estimation of Large Circuits", IEEE Design and Test of Computers, 1996, pp.70-77.
- [Schneider96b] P. H. Schneider and S. Krishnamoorthy, "Effects of Correlations on Accuracy of Power Analysis - An Experimental Study", Proceedings of the International Symposium on Low-Power Electronics and Design, 1996, pp. 113-116.
- [Shang02] L. Shang, A. Kavani, and K. Bathala, "Dynamic Power Consumption in Virtex-II FPGA Family," ACM/SIGDA International Symposium on FPGAs, Monterey, California, 2002, pp. 157-164.

- [Shen92] A. Shen, A. Ghosh, S. Devadas and K. Keutzer, "On average power dissipation and random pattern testability of CMOS combinational logic networks," IEEE International Conference on Computer Aided Design, 1992, pp. 402-407.
- [Somenzi241] F. Somenzi, *CUDD: CU Decision Diagram Package*, Release 2.4.1, URL: <http://vlsi.colorado.edu/~fabio/CUDD>.
- [Singh02] D. P. Singh and S. D. Brown, "Incremental Placement for Layout-Driven Optimizations on FPGAs," Proceedings of the 2002 ACM/IEEE International Conference on Computer Aided Design, November 2002, pp. 752-759.
- [Singh05] D. P. Singh, V. Manohararajah, and S. D. Brown, "Two-Stage Physical Synthesis for FPGAs," a double-length invited paper in the Proceedings of the IEEE Custom Integrated Circuits Conference, San Jose, California, September 2005, pp. 171-178.
- [SiS94] Software for the Synthesis of Synchronous and Asynchronous circuits. Source can be found at: <http://embedded.eecs.berkeley.edu/pubs/downloads/sis/index.htm>
- [Theodoridis00] G. Theodoridis, S. Theoharis, D. Soudris, and C. Goutis, "Switching Activity Estimation Under Real-Gate Delay Using Timed Boolean Functions", in IEE Proceedings on Computers & Digital Techniques, Vol. 147, No. 6, November 2000, pp. 444-450.
- [Tsai96] C. Tsai and M. Marek-Sadowska, "Multilevel Logic Synthesis for Arithmetic Functions," Proceedings of the 33rd Design Automation Conference, June 1996, pp. 242-247.
- [Tsui93] C. Tsui, M. Pedram and A. M. Despain, "Efficient Estimation of Dynamic Power Consumption under Real Delay Model," Proceedings of the IEEE International Conference on Computer Aided Design, 1993, pp. 224-228.
- [Vemuri02] N. Vemuri, P. Kalla, and R. Tessier, "BDD-based Logic Synthesis for LUT-based FPGAs," ACM Transactions On Design Automation of Electronic Devices, 2002, pp. 501-525.
- [Wan92] W. Wan and M. A. Perkowski, "A New Approach to Decomposition of

Incompletely Specified Multi-Output Functions Based on Graph Coloring and Local Transformations and its Applications to FPGA Mapping,” IEEE Proceedings of European Design Automation Conference, 1992, pp. 230-235.

[Wang01] Z-H. Hong Wang, E-C. Liu, J. Lai, and T-C. Wang, "Power Minimization in LUT-Based FPGA Technology Mapping," ACM/IEEE Asia South Pacific Design Automation Conference, 2001, pp. 635-640.

[Yang99] C. Yang, V. Singhal, and M. Ciesielski, “BDD Decomposition for Efficient Logic Synthesis,” Proceedings of the International Conference On Computer Design, 1999, pp. 626-631.

[Yang00] C. Yang, M. Ciesielski, and V. Singhal, “BDS: A BDD-Based Logic Optimization System,” Proceedings of the 37th International Conference on Computer Aided Design, 2000, pp. 92-97.

[Yeh99] C. Yeh, C.-C. Chang and J.-S. Wang, "Power-driven technology mapping using pattern-oriented power modeling," IEE Proceedings on Computers and Digital Techniques, Vol. 146, No. 2, March 1999, pp. 83-89.