

PRODUCT-TERM BASED
SYNTHESIZABLE EMBEDDED
PROGRAMMABLE LOGIC CORES

by

Andy Chee Wai Yan
B.A.Sc, University of British Columbia, 2002

A thesis submitted in partial fulfillment of the requirements for
the degree of

Master of Applied Science

in

The Faculty of Graduate Studies

Electrical and Computer Engineering

The University of British Columbia

January 2005

© Andy Chee Wai Yan, 2005

ABSTRACT

PRODUCT-TERM BASED SYNTHESIZABLE EMBEDDED PROGRAMMABLE LOGIC CORES

As integrated circuits become increasingly complex, the ability to make post-fabrication changes will become more important and attractive. This ability can be realized using programmable logic cores. Currently, such cores are available from vendors in the form of “hard” macro layouts. Previous work has suggested an alternative approach: vendors supply a synthesizable version of their programmable logic core and the integrated circuit designer synthesizes the programmable logic fabric using standard cells. Although this technique suffers increased delay, area, and power, the task of integrating such cores is far easier than the task of integrating “hard” cores into an ASIC or SoC. When implementing small amount of logic, this case of use may be more important than the increased overhead.

This thesis presents a new family of architectures for these “synthesizable” cores; unlike previous architectures which were based on lookup-tables, the new family of architectures is based on a collection of product-term arrays. Compared to lookup-table based architectures, the new architectures result in density improvements of 35% and speed improvements of 72% on standard benchmark circuits. In addition, we describe novel architectural designs to enhance synthesizable architectures to support sequential logic. We show that directly embedding flip-flops as is done in stand-alone programmable cores will not suffice. Consequently, we present two novel architectures employing our solution and optimize and compare them. Finally, we describe a proof-of-concept chip employing one of our proposed architectures.

TABLE OF CONTENTS

ABSTRACT	II
LIST OF FIGURES	VI
LIST OF TABLES	VIII
ACKNOWLEDGEMENTS	IX
CHAPTER 1 INTRODUCTION	1
1.1 MOTIVATION	1
1.2 RESEARCH GOALS	3
1.3 ORGANIZATION OF THIS THESIS	5
CHAPTER 2 BACKGROUND AND PREVIOUS WORK	7
2.1 SOC DESIGN METHODOLOGY	7
2.2 PROGRAMMABLE IP IN SOCS	9
2.3 OVERVIEW OF PROGRAMMABLE LOGIC ARCHITECTURES	11
2.3.1 <i>FPGA Architecture</i>	12
2.3.2 <i>CPLD Architecture</i>	16
2.4 COMPARISON BETWEEN CPLDS AND FPGAS	20
2.5 PROGRAMMABLE LOGIC COMPUTER-AIDED DESIGN FLOW	21
2.6 EMBEDDED PROGRAMMABLE LOGIC CORES	25
2.6.1 <i>Hard Programmable Logic Cores</i>	25
2.6.2 <i>Synthesizable Programmable Logic Cores</i>	26
2.7 FOCUS AND CONTRIBUTIONS OF THIS THESIS	31
CHAPTER 3 PRODUCT-TERM BASED ARCHITECTURES	33
3.1 COMBINATIONAL PRODUCT-TERM BLOCK ARCHITECTURE	33
3.1.1 <i>Logic Blocks</i>	33
3.1.2 <i>Interconnect Architecture</i>	36

3.2 SEQUENTIAL PRODUCT-TERM BLOCK ARCHITECTURE.....	39
3.2.1 Logic Blocks	39
3.2.2 Interconnect Architecture	40
3.3 ARCHITECTURAL PARAMETERS.....	44
3.3.1 High-Level Parameters.....	44
3.3.2 Low-Level Parameters.....	44
3.4 SUMMARY	45
CHAPTER 4 PLACEMENT AND ROUTING ALGORITHMS	47
4.1 OPTIMIZATION GOALS.....	47
4.2 PLACEMENT ALGORITHM	48
4.2.1 Simulated Annealing-Based Algorithm.....	49
4.2.2 Greedy-Based Algorithm	50
4.3 ROUTING ALGORITHM.....	57
4.4 SUMMARY	60
CHAPTER 5 LOW-LEVEL ARCHITECTURE PARAMETER OPTIMIZATION	61
5.1 EXPERIMENTAL FRAMEWORK	62
5.2 PRODUCT-TERM LOGIC BLOCK PARAMETER OPTIMIZATION.....	64
5.2.1 Number of Product Terms, p , per PTB.....	64
5.2.2 Number of Inputs, i , per PTB.....	66
5.2.3 Number of Outputs, o , per PTB	67
5.3 PRODUCT TERM BASED INTERCONNECT PARAMETER OPTIMIZATION	68
5.3.1 Value of r for Combinational Rectangular Fabrics.....	69
5.3.2 Value of α for Combinational Triangular Fabrics.....	70
5.3.3 Comparison of Triangular and Rectangular Combinational Architectures	72
5.3.4 Value of r for Sequential Rectangular Fabrics.....	72
5.3.5 Value of α for Sequential Triangular Fabrics.....	73
5.3.6 Comparison of Triangular and Rectangular Sequential Architectures	74

5.4 FLIP-FLOP PLACEMENT PARAMETER OPTIMIZATION	75
5.4.1 Value of v for Dual-Network Architecture.....	75
5.4.2 Value of d for Decoupled Architecture	77
5.4.3 Comparison of Dual-Network and Decoupled Architecture.....	78
5.5 LIMITATION OF EXPERIMENTAL RESULTS	79
5.6 SUMMARY	79
CHAPTER 6 COMPARISON AND IMPLEMENTATION.....	81
6.1 COMPARISON TO LUT-BASED ARCHITECTURE	81
6.2 PROOF-OF-CONCEPT IMPLEMENTATION	86
6.3 SUMMARY	89
CHAPTER 7 CONCLUSIONS AND FUTURE WORK.....	90
7.1 SUMMARY	90
7.2 FUTURE WORK.....	92
7.3 CONTRIBUTIONS OF THIS WORK	94
REFERENCES.....	96

LIST OF FIGURES

FIGURE 2.1: TYPICAL SOC DESIGN.....	8
FIGURE 2.2: CONCEPTUAL FPGA MODELS.....	12
FIGURE 2.3: FOUR-INPUT LOOK-UP-TABLE.....	13
FIGURE 2.4: A GENERIC FPGA LOGIC BLOCK.	14
FIGURE 2.5: AN ISLAND-STYLE FPGA ROUTING BLOCKS.....	15
FIGURE 2.6: THREE TYPES OF PROGRAMMABLE SWITCHES USED IN SRAM-BASED FPGAS.....	16
FIGURE 2.7: COMPLEX PROGRAMMABLE LOGIC DEVICE ARCHITECTURE.....	17
FIGURE 2.8: PRODUCT-TERM BLOCK.....	18
FIGURE 2.9: CPLD CROSSBAR SWITCH.	20
FIGURE 2.10: THE FPGA CAD FLOW.....	22
FIGURE 2.11: HARD PROGRAMMABLE LOGIC CORE INTEGRATION.....	25
FIGURE 2.12: SYNTHESIZABLE (SOFT) PROGRAMMABLE LOGIC CORE INTEGRATION.....	27
FIGURE 2.13: SYNTHESIZABLE PROGRAMMABLE LOGIC ARCHITECTURE – DIRECTIONAL ARCHITECTURE.....	29
FIGURE 2.14: SYNTHESIZABLE PROGRAMMABLE LOGIC ARCHITECTURE – SEGMENTED ARCHITECTURE.....	30
FIGURE 2.15: SYNTHESIZABLE PROGRAMMABLE LOGIC ARCHITECTURE – GRADUAL ARCHITECTURE.....	31
FIGURE 3.1: PRODUCT-TERM LOGIC BLOCK.....	34
FIGURE 3.2: SCHEMATIC LAYOUT OF PTB ARCHITECTURE.....	35
FIGURE 3.3: PRODUCT-TERM BLOCK ARCHITECTURES.....	37
FIGURE 3.4: DETAILED VIEW OF ROUTING FABRIC.....	38
FIGURE 3.5: SEQUENTIAL PTB LOGIC BLOCK.....	40
FIGURE 3.6: ADDING REGISTERS TO LOGIC BLOCKS.....	41
FIGURE 3.7: DUAL-NETWORK ARCHITECTURE.....	42
FIGURE 3.8: DECOUPLED ARCHITECTURE.....	43
FIGURE 4.1: PSEUDO-CODE FOR SIMULATED ANNEALING.....	49
FIGURE 4.2: HIGH-LEVEL DESCRIPTION OF PLACEMENT ALGORITHM.....	51
FIGURE 4.3: EXAMPLE OF REQUIRED TIME CALCULATION.....	53

FIGURE 4.4: DETAILED DESCRIPTION OF PLACEMENT ALGORITHM.....	55
FIGURE 4.5: EXAMPLE CIRCUIT MAPPING ONTO PROGRAMMABLE FABRIC.	56
FIGURE 4.6: EXAMPLE CIRCUIT PLACEMENT STEPS.	57
FIGURE 4.7: EXAMPLE PTB MULTIPLEXER SIGNAL CONNECTION.....	58
FIGURE 4.8: EXAMPLE PTB MULTIPLEXER SIGNAL ROUTING.	60
FIGURE 5.1: PTB ARCHITECTURE EXPERIMENTAL METHODOLOGY	63
FIGURE 5.2: NUMBER OF PRODUCT TERMS PER PTB.	65
FIGURE 5.3: NUMBER OF PRODUCT TERMS PER PTB (SMALL CIRCUITS).....	66
FIGURE 5.4: NUMBER OF PRODUCT TERMS PER PTB (LARGE CIRCUITS).....	66
FIGURE 5.5: NUMBER OF INPUTS PER PTB.	67
FIGURE 5.6: NUMBER OF OUTPUTS PER PTB.	68
FIGURE 5.7: PTB RECTANGULAR ARCHITECTURE PARAMETER, r	69
FIGURE 5.8: VALUE OF PTB RECTANGULAR ARCHITECTURE PARAMETER, r	70
FIGURE 5.9: PTB TRIANGULAR ARCHITECTURE PARAMETER, α	71
FIGURE 5.10: VALUE OF PTB TRIANGULAR ARCHITECTURE PARAMETER, α	71
FIGURE 5.11: VALUE OF PTB RECTANGULAR ARCHITECTURE PARAMETER, r , FOR SEQUENTIAL FABRICS.	73
FIGURE 5.12: VALUE OF PTB TRIANGULAR ARCHITECTURE PARAMETER, α , FOR SEQUENTIAL FABRICS.	74
FIGURE 5.13: DUAL-NETWORK ARCHITECTURE PARAMETER, v	76
FIGURE 5.14: VALUE OF DUAL-NETWORK ARCHITECTURE PARAMETER, v	77
FIGURE 5.15: DECOUPLED ARCHITECTURE PARAMETER, d	78
FIGURE 5.16: VALUE OF DECOUPLED ARCHITECTURE PARAMETER, d	78
FIGURE 6.1: AREA AND DELAY PLOTS FOR THE PTB AND LUT-BASED ARCHITECTURE.	85
FIGURE 6.2: GROWTH TRENDS OF THE PTB AND LUT-BASED ARCHITECTURE.	85
FIGURE 6.3: FLOORPLAN OF PROOF-OF-CONCEPT CHIP.....	86
FIGURE 6.4: CLOCK NETWORK OF PROOF-OF-CONCEPT CHIP.....	87
FIGURE 6.5: SYNTHESIS RESULTS OF PROOF-OF-CONCEPT CHIP.....	88

LIST OF TABLES

TABLE 2.1: PROGRAMMABLE DEVICE SUMMARY	21
TABLE 3.1: HIGH-LEVEL ARCHITECTURAL PARAMETERS	44
TABLE 3.2: LOW-LEVEL ARCHITECTURAL PARAMETERS	45
TABLE 5.1: SUMMARY OF OPTIMIZED LOW-LEVEL PARAMETER VALUES	80
TABLE 6.1: COMPARISON OF PRODUCT-TERM ARCHITECTURE TO LUT-BASED ARCHITECTURE.....	83
TABLE 6.2: SYNTHESIZABLE PROGRAMMABLE ARCHITECTURE IMPLEMENTATION DETAILS.....	84
TABLE 6.3: AREA AND DELAY SUMMARY.....	88

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor, Dr. Steven Wilton for the guidance, patience, and knowledge that he provided throughout my Masters, as well as my professional development. I attribute my Masters degree to his expert advice and encouragement; without him, completion of this thesis would not have been possible. One simply could not have wished for a better supervisor.

I would also like to thank my colleagues and friends: Jess Chia, Ken Chow, Noha Kafafi, Andy Kuo, Zion Kwok, Julien Lamoureux, Ernie Lin, Martin Ma, Kara Poon, and James Wu. Their constant source of cheerfulness and insightful advice about my research and are much appreciated and always remembered.

This research project was funded by the Natural Science and Engineering Research Council of Canada (NSERC), Altera Corporation, and Micronet R&D. Their support is greatly appreciated.

Finally, I would like to especially thank my family for enduring this long process with me, while always offering endless concern, support, and encouragement. Their advice has always been cherished and highly respected. Special thanks to my sister, Karen, for waking up at unearthly hours to drive me to and from the airport and not complaining one bit.

Chapter 1

INTRODUCTION

1.1 Motivation

Recent years have seen an impressive improvement in the achievable density of integrated circuits. In order to utilize this excess capacity, while maintaining reasonable design and test costs, the System-on-a-Chip (SoC) design methodology has emerged. In this methodology, pre-designed and pre-verified blocks, often called cores, are obtained from internal sources or third-parties, and combined onto a single chip. Although this technique partially alleviates some of the complexity, the design and test of a correctly-functioning integrated circuit is still a difficult task.

The design costs of manufacturing these chips are also rapidly increasing. A 0.13 μ m technology mask set costs approximately \$700,000 USD; a 90nm set averages about \$1.7M USD; and a 65nm set is estimated to cost over \$3.0M USD [1, 2]. This problem is further aggravated by the fact that ensuring a design is error-free has become virtually impossible. There will always be some chips that are designed, fabricated, and then deemed unsuitable. This may be due to design errors not detected during development and manufacturing or simply due to changes in the design requirements.

A partial solution to this problem is to reduce the number of re-spins (redesigns due to errors) per design thereby reducing the costs. This can be accomplished by providing post-fabrication flexibility through the use of embedded programmable logic cores. A programmable logic core is a flexible logic fabric that can be customized to implement any digital circuit after fabrication [3, 4, 5, 6, 7, 8]. Before fabrication, the designer embeds a programmable fabric consisting of many uncommitted gates and programmable interconnects between the gates. After fabrication, the designer can then program these gates and the connections between them.

Post-fabrication flexibility is attractive for a number of reasons. First, it may be possible to postpone some design details until late in the design cycle. Secondly, as products are upgraded, or as standards change, it may be possible to incorporate these changes using the programmable part of the chip, without fabricating an entirely new device. Finally, it may be possible to fabricate a single chip for an entire family of devices. The characteristics that differentiate each member of the family can be implemented using the programmable logic. The advantages of post-fabrication flexibility are described in more detail in the following chapter. Several integrated circuits containing programmable logic cores have been described [9, 10, 11, 12, 13].

Despite these compelling advantages, the use of programmable logic cores has not become mainstream. In fact, many companies that develop these cores have either changed focus or gone out of business. There are a number of reasons for this. One reason is that designers often find it difficult to identify a subsystem that can be implemented in programmable logic. A second reason is that embedding a core with an unknown function makes timing, power distribution, integration, and verification difficult. Another disadvantage is that embedded

programmable logic cores often come in fixed sizes and thus may lead to a waste of chip area. Lastly, embedded programmable logic cores must address physical connection and placement issues. This is difficult when the regions of fixed and programmable logic are tightly-coupled together and/or when there are a number of small programmable pieces distributed over the entire SoC. This extra design complexity limits the use of programmable logic cores to only the very best VLSI designers.

In [14], an alternative technique is described which addresses the last three concerns by shifting the burden from the SoC designer to mature standard-cell synthesis tools. In this technique, core vendors supply a synthesizable version of their programmable logic core (a “soft” core) and the integrated circuit designer synthesizes the programmable logic fabric using standard cells. A “soft core” is one in which the designer obtains a description of the behaviour of the core, written in a hardware description language. Note that this is distinct from the behaviour of the circuit to be implemented in the core, which is determined after fabrication. Here, we are referring to the behaviour of the programmable logic core itself. Although this technique suffers increased delay, area, and power overhead, the task of integrating such cores is far easier than the task of integrating “hard” cores into a fixed-function chip. For very small amounts of logic, this ease of use may be more important than the increased overhead. The techniques for creating synthesizable programmable logic cores have also been patented [15] and licensed by Altera Corporation.

1.2 Research Goals

In this thesis, we present a new family of architectures for a synthesizable embedded programmable logic core (PLC). Unlike the architectures in [14], which are based on lookup-

tables (LUT's), our new family of architectures is based on a collection of product-term array blocks. It is well-known that product-term array blocks can result in density and speed improvements for small size circuits [16, 17]; in this thesis, we show that the small combinational circuits envisaged for these synthesizable cores are very suitable for product-term based architectures. More specifically, the objectives of this research are as follows:

1. To propose a new architectural family of synthesizable embedded programmable logic cores, one that uses the product-term array block as the basic logic element for programmability.
2. To provide enhancements to synthesizable architectures to support sequential circuit implementations.
3. To develop the accompanying CAD tools required to map user circuits onto the proposed novel architectures.
4. To describe a proof-of-concept chip employing a fabric from our new architectural family.

The first goal is to describe a family of product-term based architectures. These novel architectures must satisfy a number of requirements. Firstly, the design should focus on area and delay minimization due to inherent inefficiencies in standard-cell designs compared to hand layout. The next requirement is that the architecture must satisfy any existing constraints imposed on synthesizable architectures as described in [14]. Lastly, the new architectural family should be flexible to modify and scalable in size.

The synthesizable architectures presented in [14] can only implement combinational logic circuits. Our second goal is to extend existing synthesizable architectures to provide support for sequential circuits. A straightforward embedding of flip-flops within each logic block, as is done for stand-alone FPGAs, will not suffice. We explain why simply embedding flip-flops in each logic block will not suffice, and outline two solutions. The product-term based synthesizable architecture family will be used as the baseline for modifications.

The development of any novel architecture must be accompanied by supporting CAD tools. Our third goal is to create placement and routing tools required to program user circuits onto our product-term based fabrics.

The final goal of this research is to outline a proof-of-concept chip employing our novel architecture. We also compare the performance of product-term based architectures with the performance of LUT-based architectures from [14].

1.3 Organization of this Thesis

This thesis is organized as follows: Chapter 2 describes the SoC design methodology, which is the base application for synthesizable embedded programmable logic cores. A description of both stand-alone and embedded programmable logic architectures and their CAD follows shortly. Chapter 3 introduces the new product-term based architecture family, and shows how it differs from standard commercial product-term based architectures. In Chapter 4 we present the placement and routing algorithms created to support the novel architectures. Chapter 5 describes the experimental results aimed at optimizing several architectural parameters. The CAD algorithms presented in Chapter 4 are used to help facilitate the evaluation. Next, Chapter 6

compares the new architecture to the LUT-based architecture from [14] and presents a proof-of-concept chip implementation. Finally, this thesis concludes with a brief summary of the work presented, possible future research directions, and a list of contributions made. Parts of this work have been published in [18] and [19].

Chapter 2

BACKGROUND AND PREVIOUS WORK

This chapter begins with a brief overview of the SoC design methodology and explains how programmable IP blocks can help alleviate design complexities in SoCs. It will also discuss programmable logic architectures for both stand-alone and embedded devices. In addition, a brief description of the CAD flow used for mapping user circuits into programmable logic devices is presented. Finally, the chapter concludes with a description of previous work on synthesizable programmable architectures and outlines the focus and contributions of this thesis.

2.1 SoC Design Methodology

As semiconductor technology continues to improve, designers are finding it increasingly difficult to keep pace. In order to take advantage of the improvement provided by advanced process technologies, designers need new techniques to handle the increased complexities inherent in their chips. One such emerging technique is the System-On-a-Chip (SoC) design methodology. In this methodology, pre-designed and pre-verified blocks, often called cores or Intellectual Property (IP), are obtained from internal sources or third parties, and combined onto a single chip. A single SoC can implement an entire system instead of using multiple integrated circuits (ICs) on a printed circuit board. The advantages of this design methodology over its corresponding multi-chip approach include better performance, lower power consumption, and smaller volume and weight. Although there are a variety of definitions of SoC, chips designed

using the SoC methodology are generally more application-oriented and customer-specific compared to standard ICs because of the high levels of integration.

Figure 2.1 shows a typical SoC application containing multiple embedded IP cores. As the word “system” in SoC implies, SoC designs often contain one or more embedded programmable processors, as well as embedded memory blocks and other application specific cores. The main motivation for using embedded IP cores is to shorten development time through design reuse. Unlike traditional IC designs where reuse is limited to only standard-cell libraries containing basic logic gates, SoC designs allow for reuse on a larger scale. Another advantage of the use of embedded IP cores is it enables design specialization. SoC designers can import IP cores, such as CPUs, memories, and analog modules, from other companies. The SoC designer, who would have only limited knowledge of the structure of these cores, could then combine them to implement complex systems.

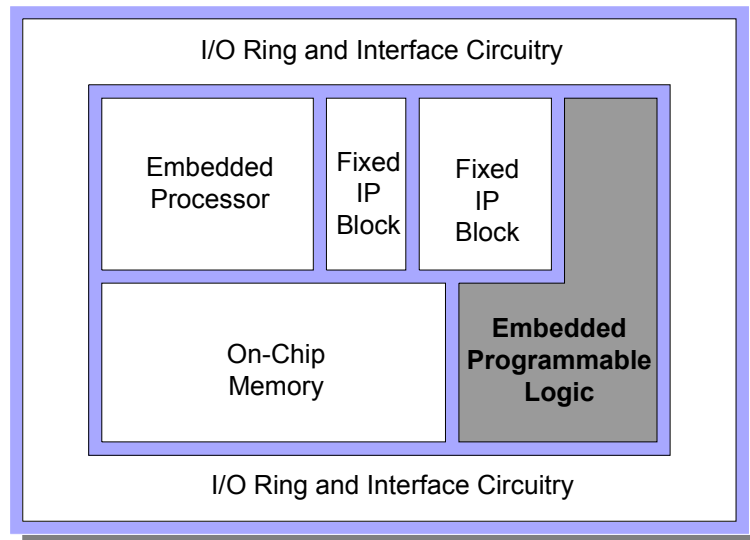


Figure 2.1: Typical SoC design.

Embedded IP cores can be characterized in a variety of ways with each method offering certain advantages and disadvantages. One of the techniques is through a register-transfer level (RTL) description of the IP core, commonly called a *soft* core. In this approach, a high-level description language, such as VHDL or Verilog, is used to describe the behaviour of the IP core. Soft cores leave much of the implementation to the physical design tools, but they are flexible and process-independent. Another technique to describe embedded IP is through a technology-dependent custom layout, commonly called a *hard* core. The advantage of this method is the design can be optimized for predictable performance, efficient area utilization, and/or low power consumption, but lacks flexibility. Finally, embedded IP cores can be described as a gate-level netlist, commonly called a *firm* core. Firm cores offer a compromise between soft and hard cores [20, 21].

2.2 Programmable IP in SoCs

As SoC designs grow in complexity, designers are finding it increasingly important to provide post-fabrication flexibility in their SoC's. There are several ways post-fabrication flexibility can ease the SoC design process:

- 1) Design decisions can be postponed to later stages in the development cycle. As an example, the development of a communications application chip can proceed to manufacturing while standards are still being finalized. Once the standards are set, they can be incorporated in the programmable portion of the SoC.
- 2) Embedded Programmable IP allows for multiple-use or field-upgradable products. As standards or requirements change, they can be easily integrated to the SoC without fabricating a new device and thereby extending the useful lifetime of the chip.

- 3) Programmable IP makes it possible to fabricate a single chip for an entire family of devices. The characteristics that differentiate each member of the family can be implemented using the programmable logic. This would amortize the cost of developing the SoC over several products.
- 4) On-chip testing can be performed. One of the greatest challenges in building a SoC is testing them. The embedded programmable core can be used to implement various test scenarios. Also as testing proceeds, new tests can be devised and the on-chip test circuitry implemented in the programmable core.
- 5) Reduce the risk of error for developers by allowing them to move critical areas of the design or possibly correcting errors by using the programmable IP core.

There are three ways to provide programmability into a SoC. The first is through software. In this method, programmability is achieved by embedding a processor into the SoC. Post-fabrication design changes are accomplished by modifying the software that is executed by the processor. This method is straightforward, but may not provide sufficient performance for some applications. The second method is through the use of programmable registers that configure different modes or options for the desired functionality of the SoC. The advantage of this method is its simplicity, but generally only offers limited flexibility. The last method of embedding programmability is through hardware. In this method, programmability is achieved by embedding a programmable logic IP core into the SoC. The programmable IP core is a flexible logic fabric that can be customized to implement any digital circuit after fabrication. Before fabrication, the designer embeds a programmable fabric (consisting of many uncommitted gates and programmable interconnects between the gates). After fabrication, the

designer can then program these gates and the connections between them. This method generally offers better performance than software programmability or register configuration.

2.3 Overview of Programmable Logic Architectures

Embedded programmable logic cores typically inherit their architectures from stand-alone Field-Programmable Gate Arrays (FPGA's). In this section we discuss the architecture of programmable logic devices.

Every programmable logic device has at least four components:

- 1) I/O resources to provide a mechanism for transferring and receiving data to and from the programmable device.
- 2) Logic elements to store and implement the required Boolean functions.
- 3) Routing resources to connect the logic and I/O elements together to implement a digital circuit.
- 4) Configuration bits to store the state of the logic and routing resources that defines the user circuit.

There are two classes of programmable logic devices: Field-Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs). The main difference between the two is the type of logic element used to implement the digital function and their *logic retention capability* (whether upon power-up of the device if the configuration bits recall their last known state). The two types of architectures will be described in more detail in the next two sections.

2.3.1 FPGA Architecture

Figure 2.2 represents both legacy and modern field-programmable gate arrays (FPGAs). Both types of FPGAs contain lookup-table (LUT) based logic elements; modern FPGAs also contain embedded components such as memories, DSP blocks, and ALUs. The organization of most FPGAs is known as *island-style* [22, 23]; commercial examples of such are Xilinx’s Virtex FPGA family [24] and Altera’s Stratix FPGA family [25]. In this architecture, the “islands” are the logic resources and they are separated by a “sea” of routing resources. A user circuit can be implemented by configuring logic functions in each LUT, and configuring their routing resources.

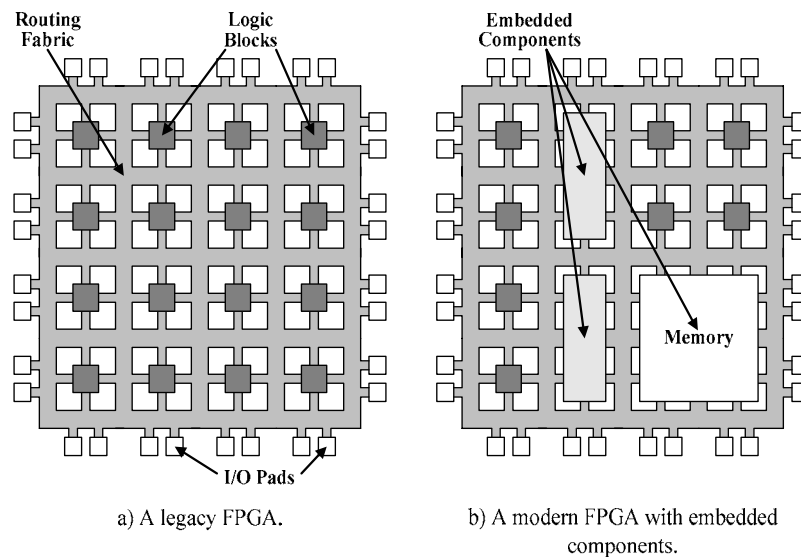


Figure 2.2: Conceptual FPGA models.

Logic blocks

FPGAs use look-up-tables (LUTs) as the programmable logic element to implement Boolean functions. A LUT is a 1-bit wide memory array in which the function inputs to the LUT are the address lines and the LUT output corresponds to the one bit output of the memory. The number

of inputs defines the size of a LUT. A k -input LUT, commonly called a k -LUT, contains 2^k memory bits. These memory bits store the truth-table of the k -input function. As a result, k -LUTs can implement any Boolean function with up to k inputs. Figure 2.3 illustrates a 4-input LUT, consisting of a 16×1 memory block connected to a tree of multiplexers. There is an area-delay-routability tradeoff for different LUT sizes. Large k -LUTs simplify circuit routing, but have a higher area and delay penalty compared to smaller k -LUTs. Previous work has shown that 4 or 5-LUTs results in the densest and fastest FPGAs [26, 27]; although recent work has suggested that significantly larger LUTs may be suitable [28, 29].

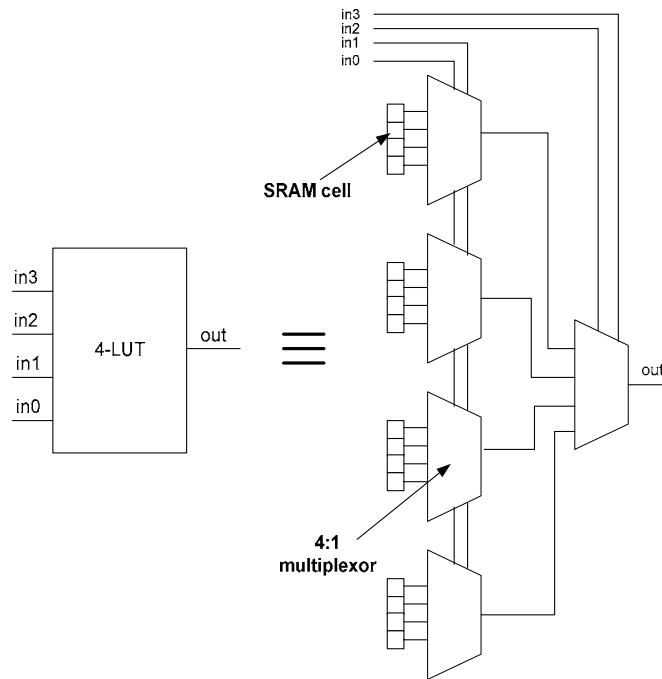


Figure 2.3: Four-input Look-up-Table.

To support sequential logic, a flip-flop and a 2-input multiplexer with a configuration bit are cascaded to the output of LUT as shown in Figure 2.4. The resulting circuit is commonly called a basic logic element (BLE). Depending on the value of the configuration bit, the LUT output can be either registered or non-registered. To reduce the number of BLEs and the amount of

routing required to connect them, between four and sixteen BLEs are combined together with some local interconnect to form a configurable logic block (CLB) or cluster, as shown in Figure 2.4. CLBs help reduce the reliance on global routing resources and increase circuit speed by providing short, fast paths between closely-knit BLEs. There is an area-delay tradeoff in CLB size; a CLB that is too large will result in area-delay inefficiencies in its local interconnect, while a CLB that is too small will not be able to encapsulate enough connections. [26] found a CLB containing between 4 to 10 BLEs is the most efficient.

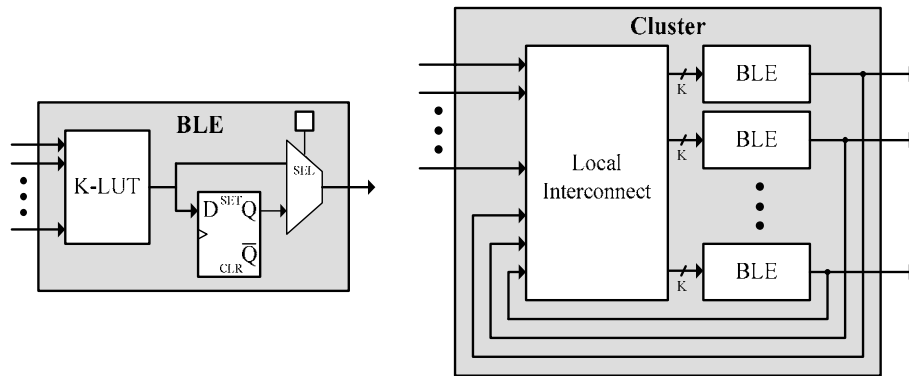


Figure 2.4: A generic FPGA logic block.

Routing Resources

Routing resources are used to connect logic blocks to other logic blocks and to I/O blocks. They account for the majority of area, delay, and power dissipation of the FPGA. The architecture of the routing resources can be separated into three categories: wire segments, switch blocks, and connection blocks. Figure 2.5 illustrates these routing blocks.

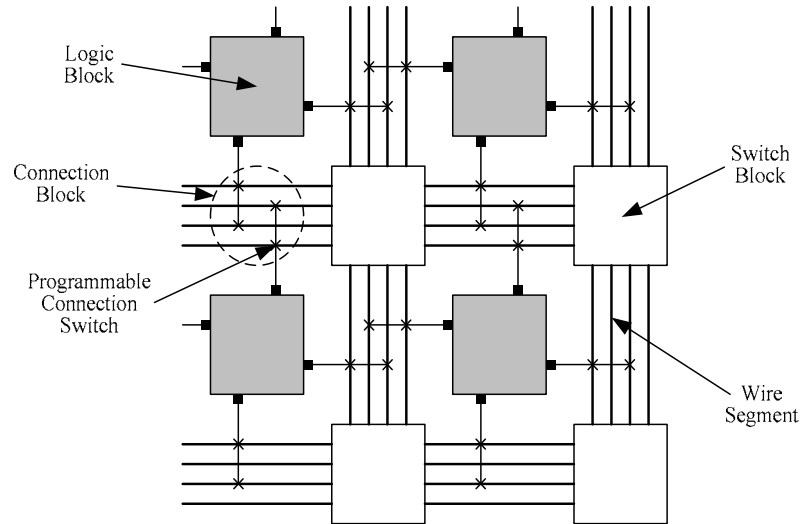


Figure 2.5: An island-style FPGA Routing Blocks.

Routing segments are wires that run across the chip horizontally and vertically. An FPGA typically contains wires of several different lengths.

Switch blocks are located at the intersection of horizontal and vertical routing channels, allowing routing segments to connect to other routing segments. A fully connected switch block (one that allows all possible horizontal and vertical routing channels connections) is not feasible due to the high area and delay overhead. Several reduced-connectivity switch block topologies have been studied and found to offer similar routability to a fully-connected switch block [23, 30, 31, 32, 33, 34].

Connection blocks are located around the logic blocks. They are used to connect logic block inputs and outputs to the routing segments. Similar to switch blocks, a fully connected connection block (one that allows all possible connections between logic block inputs/outputs to

routing tracks) is impractical. Sparsely connected connection blocks are often used with little degradation in routability [.

Programmable connections within switch blocks and connection blocks can be implemented using pass-transistors, multiplexers, or tri-state buffers as shown in Figure 2.6. SRAM cells are commonly used to control the gates and enable lines of pass-transistors and tri-state buffers, and the select lines of multiplexers. Recent work [35] has shown that architectures in which only a single driver driving each routing segment results in improved speed and reduced area, compared to architectures in which each segment can be driven from one of several sources.

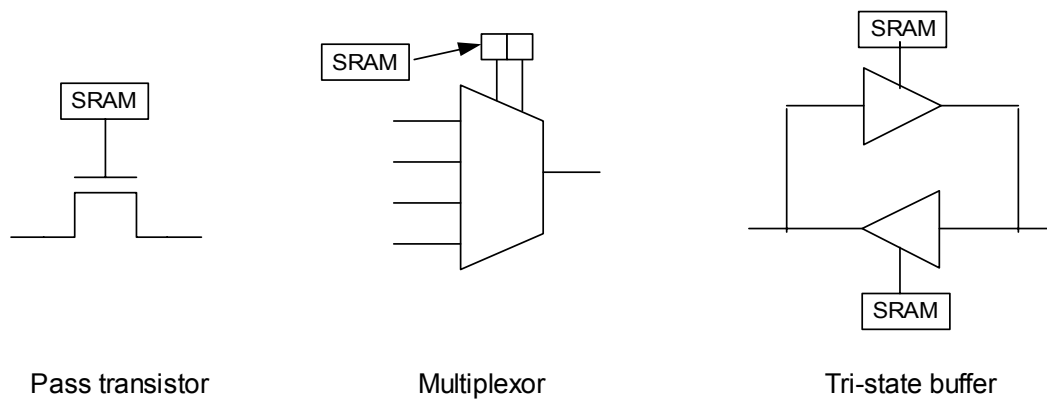


Figure 2.6: Three types of programmable switches used in SRAM-based FPGAs.

2.3.2 CPLD Architecture

Figure 2.7 shows the general architecture of a complex programmable logic device (CPLD). Similar to FPGAs, CPLDs are comprised of an array of logic blocks and interconnect wires with configuration memory distributed throughout the chip. These components will be described in the following subsections.

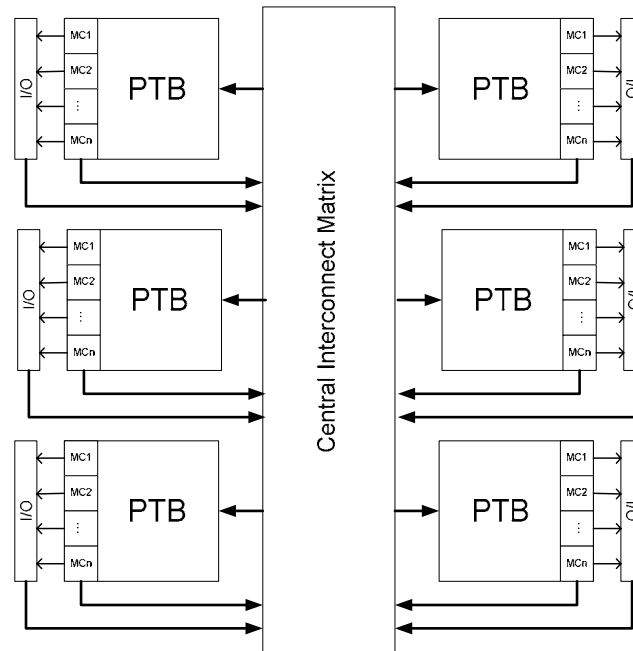


Figure 2.7: Complex programmable logic device architecture.

Logic Blocks

Most CPLDs use product-term based cells as their logic blocks (recently, however, Altera has proposed a LUT-based style CPLD [36]). Product-term blocks (PTBs), also commonly called programmable logic arrays (PLAs), implement Boolean functions in a *sum-of-product form*. A PTB consist of two planes – the AND plane and the OR plane. The AND plane generates a set of product terms that consist of any number and combination of PTB inputs or their complements. The OR plane is then used to “sum” the product terms to create the desired Boolean function. The outputs of the OR plane drive one or more *Macrocells*. Each Macrocell can optionally register an output (controlled by a programming bit) and usually contains additional circuitry to allow efficient implementation of specialized functions such as adders, multipliers, or parity checkers. The size of a PTB can be scaled by altering the number of inputs, the number of product terms (the number of AND gates in the AND array) and the number of

outputs (which is often equal to the number of OR gates in the OR array). In this thesis, we will refer to the size of a PTB using the tuple (i,p,o) where i is the number of inputs, p is the number of product terms (p-terms), and o is the number of outputs. The structure of a PTB is illustrated in Figure 2.8.

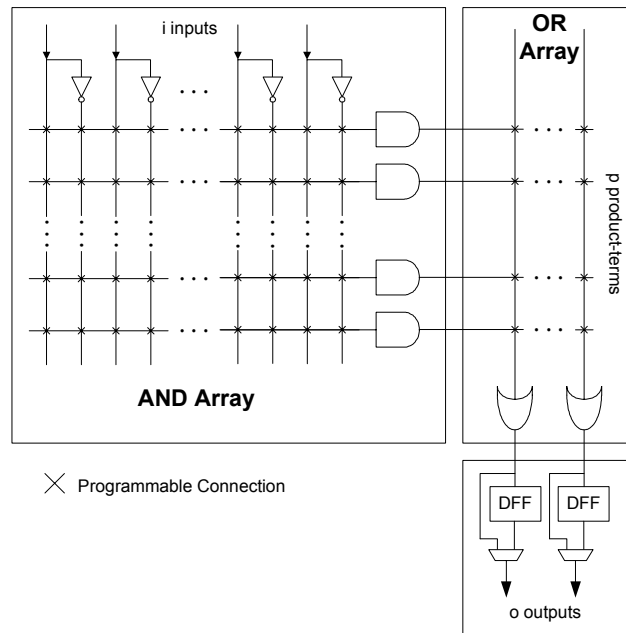


Figure 2.8: Product-Term Block.

Compared to LUT-based architectures, a PTB architecture may provide better density and speed for some logic functions. Although a LUT can implement any function of its inputs, [16] has shown that most functions mapped to a LUT only contain a few product terms, and thus can be mapped to PTB architectures efficiently. [16] also investigated the impact of PTB size on area and delay, and found that a PTB should be much larger than 4-input or 5-input LUTs commonly found in FPGAs. PTBs with (i,p,o) values of (8 to 10), (12 to 13), and (3 to 4), respectively, results in a programmable core that uses up to 25% fewer wiring tracks and 40% fewer logic levels than a LUT-based device. Commercial CPLDs, however, are much larger; they generally have (i,p,o) values of $(i=36, p=48 \text{ to } 80, o=16)$ [37, 38]. Overall, PLDs are generally faster but

less dense than FPGAs [39, 40]. The efficiency of PTB blocks can be further increased by employing *product-term sharing*, in which unused product-terms in one PTB can be distributed to other PTBs [37, 38].

Routing Resources

The CPLD routing fabric is used to connect a PTB block to another PTB block or to an I/O block. Typically, a central interconnect switch matrix surrounded by PTB blocks is employed. Although such a switch would be infeasible in a large FPGA, it is sufficient for a CPLD, since the large PTBs encapsulate most logic, meaning only a small number of connections between PTBs are required. The advantage of a central switch matrix is that most signal connections travel through similar paths, thereby eliminating skew and making timing performance easy to predict. Also the central switch matrix allow CPLDs to achieve low power designs compared to the segmented routing architecture in FPGAs [37].

The most routable switch topology is a fully-connected crossbar switch, in which all inputs can connect to all possible outputs. Such a crossbar is shown in Figure 2.9 (a). A disadvantage of a fully-connected switch is that many programmable connections (switch points) are required, leading to area and speed inefficiencies. As a result, most commercial CPLD's switch matrices are not fully-connected. Typically, a switch matrix is implemented using a combination of programmable switch points and multiplexers, as shown in Figure 2.9 (b). The multiplexers help reduce the number of switch points and the size of the multiplexers determine the flexibility of the switch matrix [41].

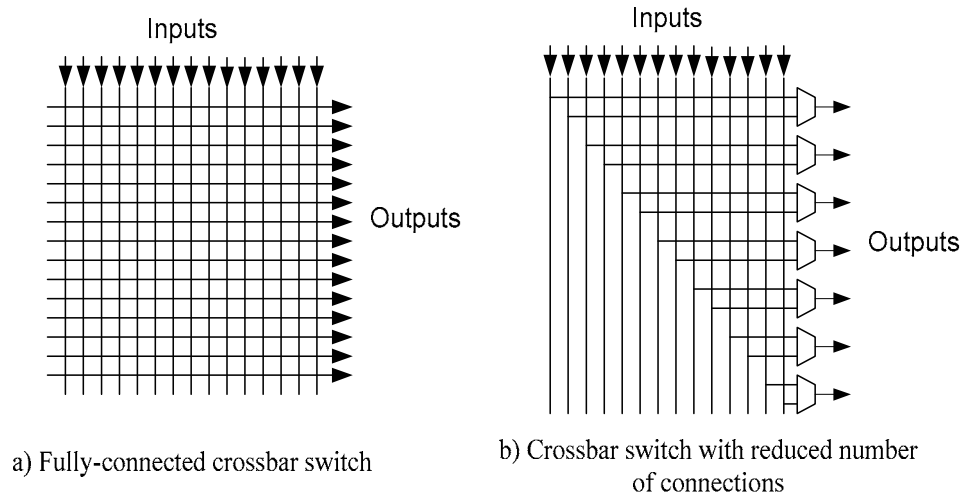


Figure 2.9: CPLD crossbar switch.

The programmable connections in the switch block and logic blocks of CPLDs are often realized using EEPROM transistors or FLASH memory [42]. Unlike SRAM cells, which are typically employed in FPGAs, EEPROM transistors and FLASH memory are *non-volatile*. The programming of the CPLD is kept intact even when the device is turned off and is instantly available upon power-up. A disadvantage of FLASH and EEPROM memory, however, is they are more expensive to fabricate.

2.4 Comparison Between CPLDs and FPGAs

This section will compare the difference between FPGAs and CPLDs that arise due to the different design realizations described previously. Table 2.1 contains a summary of the main differences.

	CPLDs	FPGAs
Application Scope	Narrow	Wide
Logic Block Density	Product-Term Block coarse-grain	Look-Up Table fine-grain
Speed	Fast	Moderate
Logic Capacity	Medium-sized circuits	Large circuits
Interconnect	Continuous	Segmented
Power Dissipation	Low	High
Architecture and Timing	Predictable and Simple	Complex
Logic Retention	Non-Volatile	Volatile

Table 2.1: Programmable Device Summary.

CPLDs and FPGAs are useful for a wide assortment of applications, from implementing random glue logic to prototyping complex designs. However, the high logic densities in FPGAs tend to make them more suitable for implementing large complex systems. CPLDs are generally limited to circuits that can exploit wide AND/OR gates and do not require a large number of flip-flops [40].

2.5 Programmable Logic Computer-Aided Design Flow

Programmable logic device computer-aided design (CAD) tools are used to convert a conceptual description of a circuit into a physical implementation on the device. The CAD flow is similar to the standard ASIC design flow. The design entry method for both flows is typically a behavioural or register-transfer level (RTL) circuit description. The output of the flow is a physical layout of transistors and wires for an ASIC, and a stream of configuration bits for a programmable logic fabric.

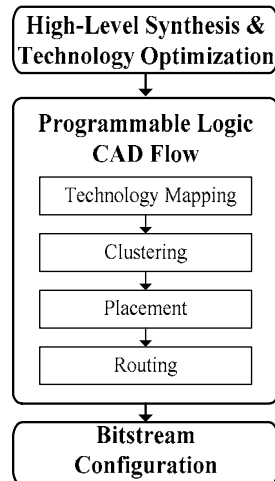


Figure 2.10: The FPGA CAD Flow.

Figure 2.10 illustrates the various stages in the programmable logic CAD flow. The first step, known as high-level synthesis, is to transform the high-level description language (HDL) of the target circuit into basic Boolean logic equations. The next step (technology optimization) is to optimize the circuit to minimize area, delay, and/or power, using techniques such as logic optimization and state assignment. The optimizations depend on the target architecture. For example, combinational logic destined for a CPLD will be minimized in a sum-of-product form, while combinational logic that is to be implemented in an FPGA will optimize the logic functions in a multi-level form. State assignment also depends on the target architecture. For a CPLD, state machines are encoded using the fewest number of flip-flops as possible. Whereas for an FPGA, flip-flops are abundant so one-hot encoding is often used. The resulting output of this stage is a network of basic logic gates, called a *netlist* [39].

The next stage is technology mapping, in which the netlist is implemented using either LUTs or PTBs for FPGAs and CPLDs, respectively. The goal of this process is to group gates in the netlist into as few logic blocks as possible and to minimize the circuit depth. There have been

extensive studies on LUT-based mapping algorithms [43, 44, 45, 46]. In contrast, limited work has been targeted for PTB-based mapping technologies. Most CPLD mapping algorithms only focus on logic block minimization [47, 48, 49, 50], however a timing-driven algorithm was presented in [51].

After technology-mapping, the logic cells are clustered into configurable logic blocks (CLBs) for FPGAs. CPLDs do not require clustering because the logic blocks do not have local interconnect.

The next step in the CAD flow is placement, in which, logic blocks (CLBs for FPGAs and PTBs for CPLDs) are assigned physical locations on the device. This is usually an easy task in CPLDs since there are relatively few logic blocks. However, architectural variations such as shared product terms can complicate this process [52]. The main goals in placement are to minimize routing demands and critical path delays. Routing resources are reduced by placing highly connected logic blocks close together. Path delays are minimized by giving timing critical nets priority so that logic blocks connected by timing-critical nets are placed close to each other. There are three common placement algorithmic approaches, *mincut* [53, 54, 55], *analytic* [56, 57, 58], and *simulated annealing* [22, 23, 59, 60, 61, 62]. Although most commercial CAD tools often use a combination of these algorithms, simulated annealing has been shown to yield reasonable results given a fixed time constraint. Simulated annealing-based placers are also more adaptable to new architectural changes and optimization goals.

The next phase of the CAD flow is routing, in which a path for each net is determined. The goal of this task is to route all required connections while minimizing critical path delay and avoiding congestion. This is achieved by assigning critical paths a higher priority for routing, selecting the shortest possible routes, and balancing the usage of wires in the device. There are two main approaches to routing, two-step *global and detailed routing* [63, 64, 65, 66, 67] and single-step *combined global-detailed routing* [68, 69, 70, 71]. The latter is commonly used in FPGAs. In CPLDs, this task is simpler, since the flexible central switch matrix results in easy identification of routes.

Most routers use Dijkstra's algorithm [72] to find the shortest possible path between source and sink. However, solely relying on this approach often yields unroutable solutions because of contention due to the limited number of routing resources. Instead, most routers use some sort of *rip-up and re-route* [73] approach. In this method, established routes may be removed and rerouted to accommodate other signals. As a result, routing is less dependent on the order in which signal nets are routed. Most FPGA CAD flows use a more advanced version of rip-up and re-route algorithms, called *negotiated congestion-delay*. This algorithm is based on the Pathfinder algorithm [69]. The algorithm performs several iterations. During early iterations, routes are allowed in which more than one net shares a routing resource. As the routing progresses, the cost of this "over-utilization" of routing resources is increased, eventually leading to a solution in which no routing resources are over-used.

The final stage of the CAD flow involves generating the configuration bitstream to program the user circuit onto the programmable device.

2.6 Embedded Programmable Logic Cores

Embedded Programmable Logic cores can either be “hard” cores, in which the SoC designer receives a layout of the core optimized for a particular technology, or a “soft” core, in which the SoC designer receives a synthesizable description of the core, written in a hardware description language such as VHDL or Verilog. This section differentiates between the two approaches, and describes several architectures for each.

2.6.1 Hard Programmable Logic Cores

Figure 2.11 shows how “hard” embedded programmable logic cores can be integrated into an SoC design. A pre-fabricated programmable core, usually obtained from third party vendors [5, 6, 7, 8], is introduced during the floorplanning stage. The programmable fabric is in the form of a transistor layout description (such as a GDSII hard-macro).

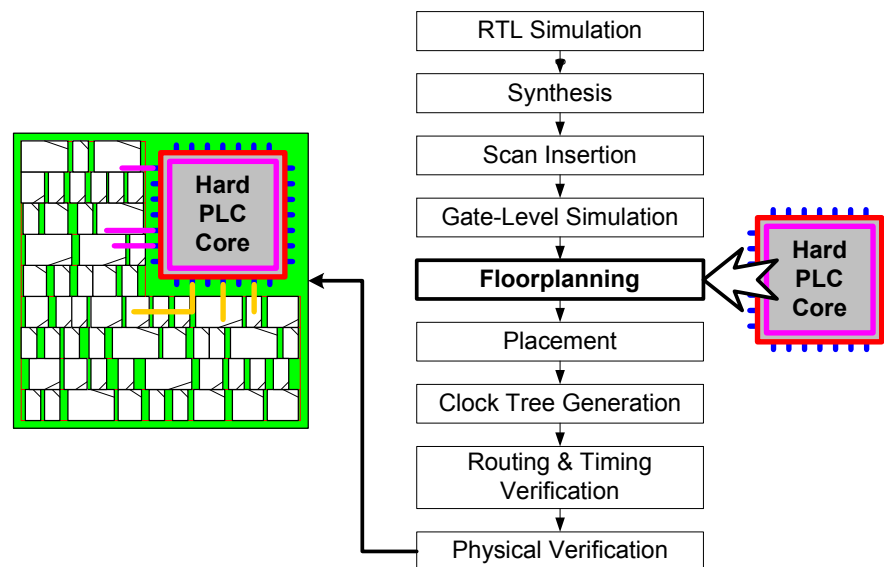


Figure 2.11: Hard programmable logic core integration.

There have been two recent works that describe hard product-term based embedded programmable logic cores. [74, 75, 76] described a multi-level PTB-based programmable fabric

called Glacier PLA in which multiple PTBs are stacked together. The PTBs are connected unidirectionally and routed using river routing. This architecture supports only combinational circuits. The stacked PTBs and the relatively sparse routing results in high circuit regularity and predictable area and delay performance. However, the focus on circuit regularity and that the architecture's interconnect is fixed prevent them from achieving high performance and flexibility.

[77, 78] proposes two architectures for a domain-specific product-term based programmable fabric for SoCs. The first architecture is a PLA (PTB)-based fabric where the product terms are shared among the OR gates in the OR plane; and the second a PAL-based fabric where the product terms are not shared, but directly connected to the OR gates. The programmable core consists of a single logic block and thus mapped circuits pass through only one logic level, resulting in predictable performance. Density improvements are obtained by optimizing away redundant programmable connections in the logic block and implementing the PLA/PAL logic blocks using pseudo-nMOS logic. However, a disadvantage in this approach is the programmable fabric is limited to domain-specific circuits due to the removal of programmable connections. Also, the use of pseudo-nMOS logic results in static power dissipation.

2.6.2 Synthesizable Programmable Logic Cores

In [14], an alternative technique is described. In this technique, core vendors supply a synthesizable version of their programmable logic core (a “soft” core) in a hardware description language and the integrated circuit designer synthesizes the programmable logic fabric using standard cells. This simplifies the task of embedding a programmable logic core in an SoC.

Since the designer receives only a description of the behaviour of the core, he or she must use synthesis tools to map the behaviour to gates. These synthesis tools can be the same ones that are used to synthesize the fixed (ASIC) portions of the chip. Figure 2.12 illustrates the design flow. Instead of inserting a hard programmable logic core during floorplanning, the embedded programmable logic fabric is inserted during RTL simulation and synthesis stage. The resulting layout contains a combination of both fixed and programmable logic implemented in standard-cells.

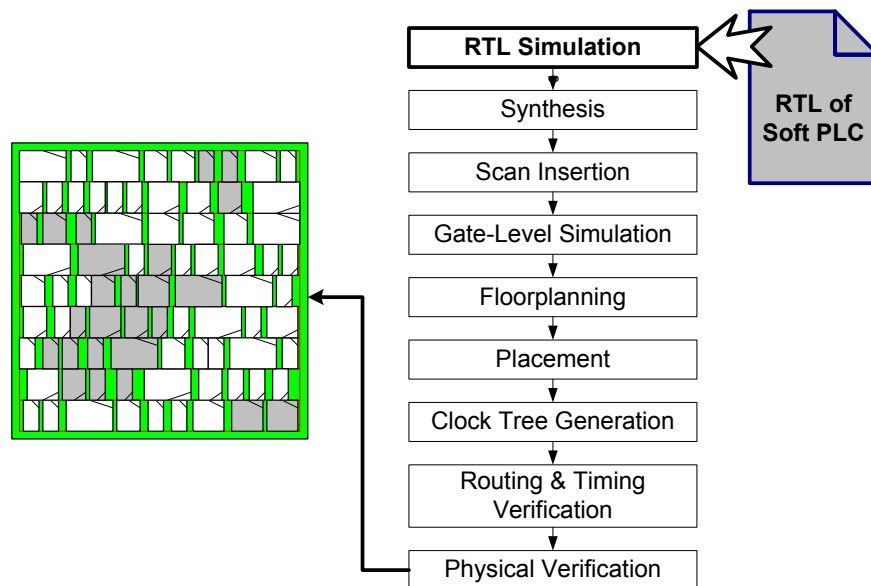


Figure 2.12: Synthesizable (soft) programmable logic core integration.

The primary advantage of this method is that existing ASIC tools can be used to implement the chip. No modifications to the tools are required, and the flow follows a standard integrated circuit design flow that designers are familiar with. Implementing a synthesizable embedded programmable logic core is essentially the same as implementing a digital circuit through an RTL level description. This will significantly reduce the design time of chips containing these cores. A second advantage is that this technique allows small blocks of programmable logic to

be positioned very close to the fixed logic that connects to the programmable logic. The use of a “hard core”, however, requires that all the programmable logic be grouped into a small number of relatively large blocks. A third advantage is that the new technique allows users to customize the programmable logic core to support his or her needs precisely. This is because the description of the behaviour of the programmable logic core is a text file which can be edited and understood by the user. Finally, it is easy to migrate the circuit to new technologies; new programmable logic cores from the core vendors are not required.

The primary disadvantage of the proposed technique is that the area, power, and speed overhead will be significantly increased, compared to implementing programmable logic using a hard core. Thus, for large amounts of circuitry, this technique would not be suitable. It only makes sense if the amount of programmable logic required is small. An envisaged application might be the next state logic in a state machine.

Synthesizable programmable logic cores have been described before. [79] proposed the design of domain-specific FPGAs using standard cells. Their architecture involves generating a structural Verilog representation of an FPGA and then creating a standard-cell layout of the fabric. To increase efficiency they created custom layouts of standard FPGA components, such as LUTs, SRAM cells, and multiplexers. They also removed some flexibility in the FPGA (logic blocks and routing resources) and added coarse-grained blocks such as adders and multipliers. The resulting design is a high performance reconfigurable core, but targeted only to specific application domains, namely arithmetic intensive computations. This is different from our work

in that we are not only targeting arithmetic computations in a specific application domain, but instead we are proposing a general-purpose programmable core.

[14, 80] presented three synthesizable FPGA architectures. LUTs were used as the basic logic element to provide programmability. The first architecture, called Directional, mimics a standard FPGA, as shown in Figure 2.13 (a). The basic logic element is a 3-LUT. Flip-flops are not present, since the core is aimed at small combinational circuits. To further simplify the routing structure and because of timing complexities in synthesis tools when dealing with combinational loops, a “directional” interconnect architecture, in which signals flow from left to right and up and down, but never backwards, was imposed. The switch block is shown in Figure 2.13 (b).

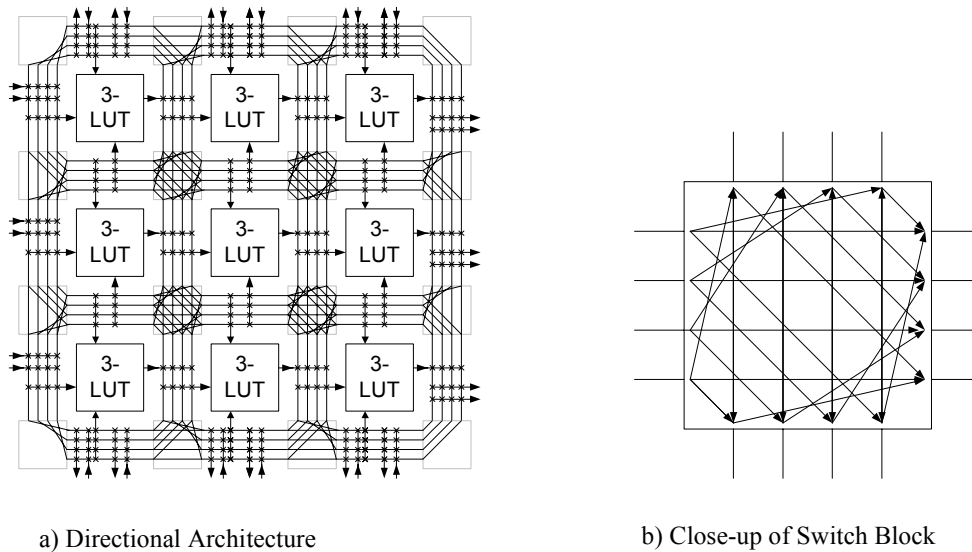


Figure 2.13: Synthesizable programmable logic architecture – Directional Architecture.

The second architecture described in [80] is shown in Figure 2.14. This architecture is called the Segmented Architecture. Multiplexers are used to transfer signals to and from the logic blocks and primary inputs and outputs. They are also used to provide a unidirectional signal flow from

left to right. Similar to the Directional Architecture, the Segmented Architecture uses 3-LUTs. Two of the inputs to the LUTs are driven from horizontal tracks below the LUT and the third input is driven from the horizontal track above the LUT. The outputs of the LUTs are connected to vertical tracks. Vertical and horizontal tracks can be connected together using additional multiplexers.

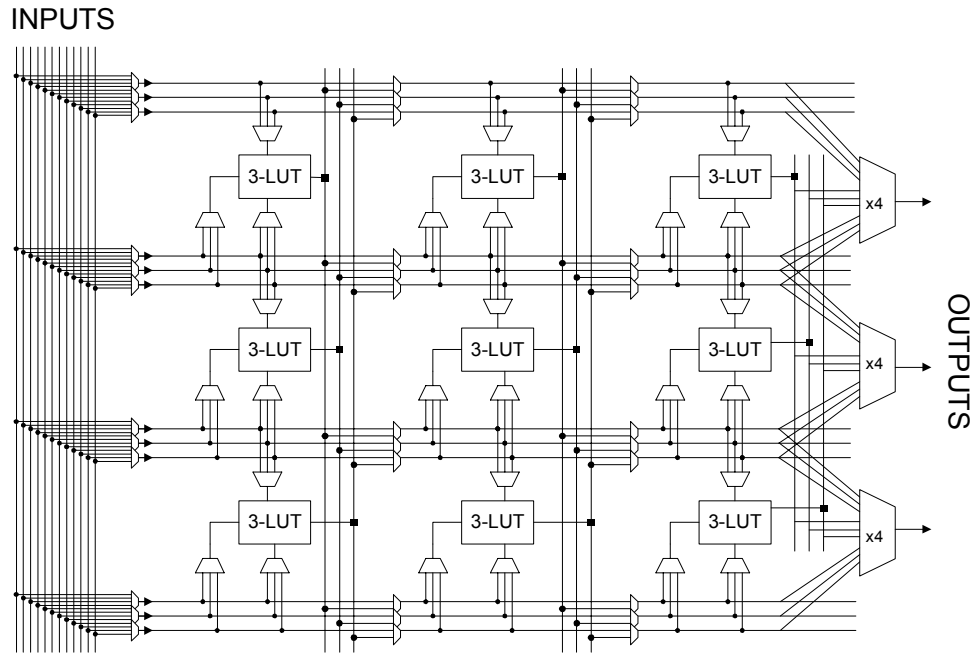


Figure 2.14: Synthesizable programmable logic architecture – Segmented Architecture.

The third architecture from [14, 80], called the Gradual Architecture, is shown in Figure 2.15. The name of this architecture derives from the gradual increasing number horizontal tracks from left to right. Similar to the previous architectures, the logic resources consist of 3-LUTs and signals can only flow from left to right. The outputs of the LUTs are only accessible through vertical tracks. The inputs of the LUTs, however, can connect to horizontal channels and to vertical channels in previous columns. Horizontal routing multiplexers are used to connect vertical channels to LUTs in nonadjacent columns. Comparing the density of the three

architectures, the Gradual architecture is the most area efficient, requiring 18-25% less area than the previous two architectures described above.

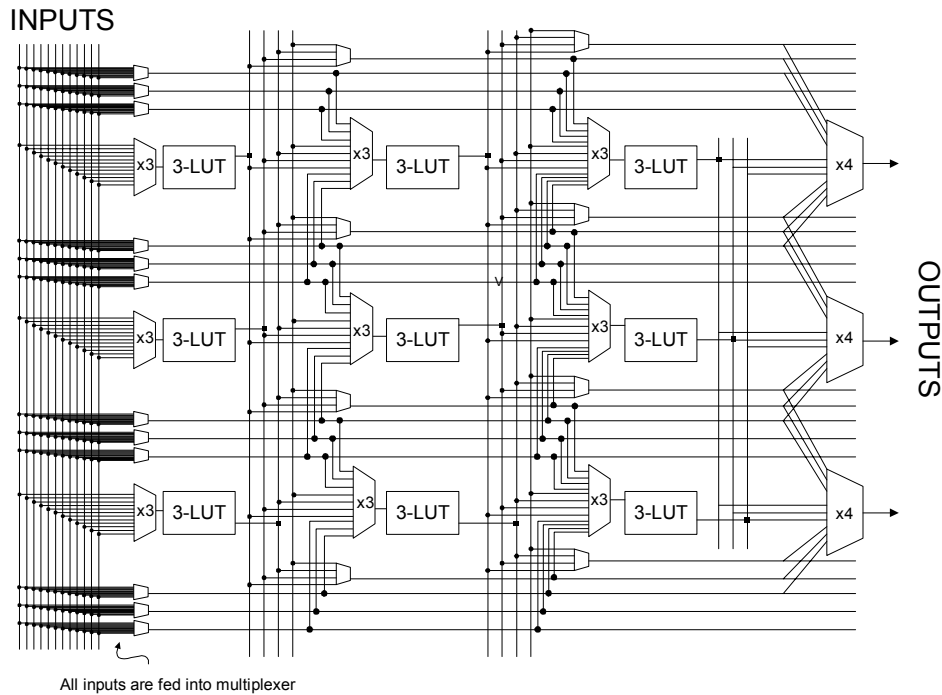


Figure 2.15: Synthesizable programmable logic architecture – Gradual Architecture.

2.7 Focus and Contributions of this Thesis

The goal of this research is to evaluate the feasibility of using product-term blocks for the design of synthesizable programmable logic cores in an SoC environment. As described in the previous section, lookup-table based synthesizable cores have already been evaluated. However, the small combinational circuits that are the target of these cores may be more efficiently implemented using product-term blocks. In Chapter 3, we propose a novel product-term based synthesizable architecture and describe parameters that characterize the architecture. In Chapter 4, we present a new placement algorithm used to map user circuits to our new programmable logic architecture. In Chapter 5, we experimentally optimize our architecture; finally in Chapter

6, we compare our product-term based architecture with the look-up table based architectures from [14] and present a proof-of-concept chip that illustrates our approach.

The contributions of this thesis are as follows:

1. A novel product-term based synthesizable architecture and the identification and optimization of architectural parameters.
2. A method to provide sequential logic support for synthesizable architectures.
3. A placement algorithm that maps to the new architecture.
4. A comparison of the proposed architecture with the look-up table based architectures in [14]
5. A proof-of-concept chip implementation.

Chapter 3

PRODUCT-TERM BASED ARCHITECTURES

In this chapter, we present a new family of architectures for synthesizable programmable logic cores. Unlike previous architectures, which were based on lookup-tables [14, 80], the new family of architectures is based on a collection of product-term arrays. Two programmable fabrics are introduced. The first supports only combinational circuit designs and the second supports both combinational and sequential circuit designs. We present two routing network architectures for the sequential fabric. Also, two interconnect structures are described. The first consists of a rectangular arrangement of the product-term arrays and the second consists of a triangular arrangement. Finally, this chapter concludes with a discussion of the architectural parameters that a designer would specify to choose a specific core from our programmable fabric library.

3.1 Combinational Product-Term Block Architecture

3.1.1 Logic Blocks

Similar to CPLD architectures described in Chapter 2, we employ product-term blocks as the basic programmable logic element. PLA-style logic blocks (called PTBs) are used instead of PAL-type logic blocks because their flexibility allows for more efficient logic implementation and because of the availability of PLA-based technology mapping algorithms. A small

programmable logic core may consist of a single PTB. The behaviour of the single PTB can be written in a hardware description language, synthesized, and combined with the rest of the integrated circuit. This works well for small cores, but as the number of inputs, product terms, and outputs grow, the size of the synthesized fabric will grow. The number of programming bits (which are implemented using flip-flops in the synthesized fabric) is proportional to the product of the number of inputs and the number of product terms added to the product of the number of product terms and outputs. For large cores, this becomes unwieldy. Because of this, product-term based devices usually contain a collection of smaller PTBs connected using a very flexible interconnect switch matrix.

The basic logic block for the combinational architecture is illustrated in Figure 3.1. Compared to PTBs in commercial devices, there is no macrocell attached to each output of our PTB; this results in improved area and delay for combinational circuits. As in [14], this architecture targets only small combinational circuits (such as the next state logic in a state machine). Therefore, the logic blocks do not contain any registered outputs. However, registers can be attached to the periphery of the programmable core if desired.

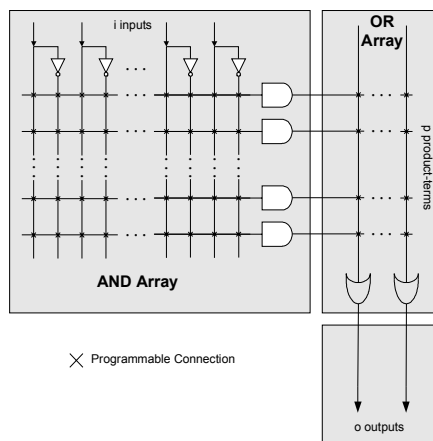


Figure 3.1: Product-Term logic block.

Figure 3.2 shows the schematic layout of the synthesizable PTB. The programming bits are realized using flip-flops that are daisy-chained together, similar to a scan chain. Although it would be more efficient to use SRAM cells to implement the programming bits, SRAM cells were not considered due to complexities with the physical design tools. To program the PTB, the configuration bit stream is sent into the “scan in” input port. Faults in the configuration chain can be detected by probing the “scan out” output port. The flip-flops control the state of the logic gates that make up the AND and OR plane. A “0” in the flip-flop means the gate is disabled and, conversely, a “1” enables the gates in the plane.

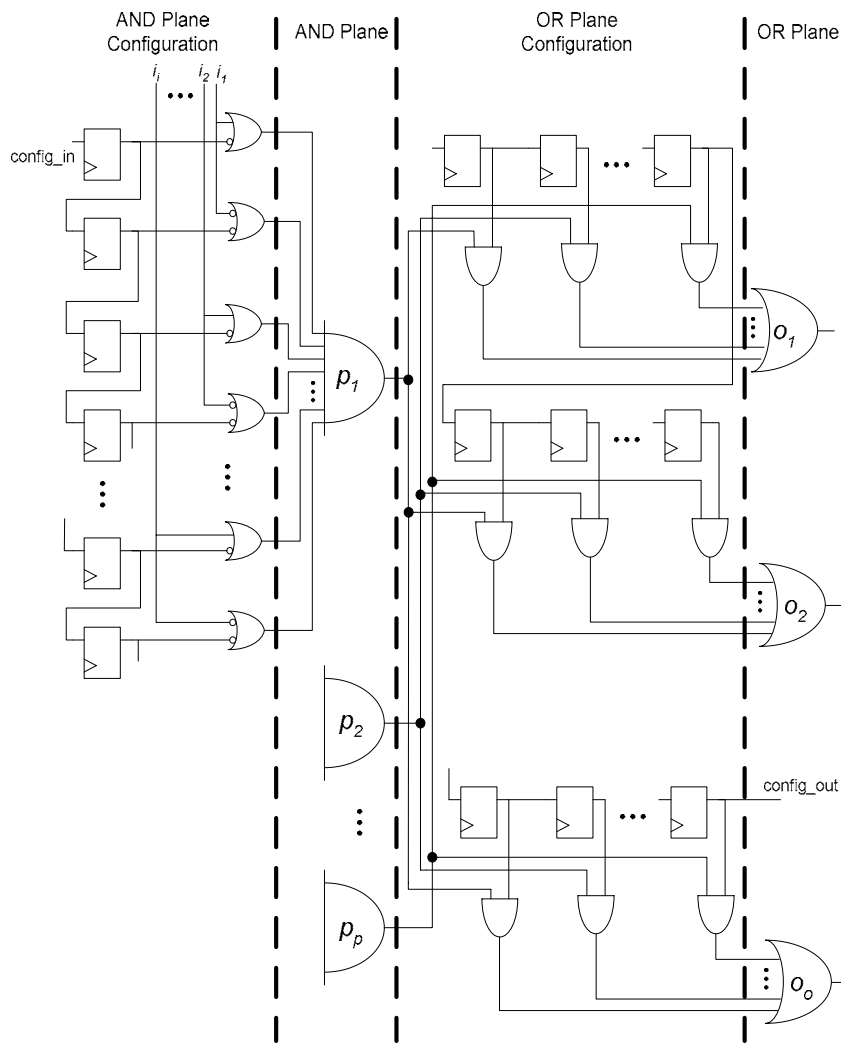


Figure 3.2: Schematic layout of PTB architecture.

3.1.2 Interconnect Architecture

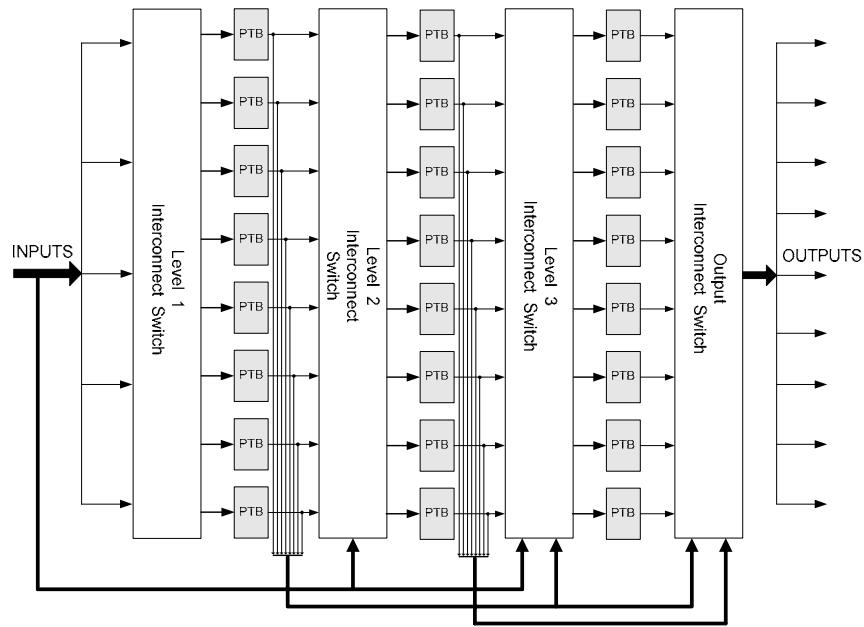
Design Constraints

Unfortunately, the single global interconnect switch matrix found in commercial CPLD devices is not appropriate for a synthesizable PTB-based architecture. This is because it allows PTB outputs to be connected to any other PTB input (or a large fraction of them). This can create combinational loops in the unprogrammed fabric (for example, if the output of a PTB is connected to the input of the same PTB). These combinational loops are normally not a problem, since it is up to the user to configure the fabric in such a way that these combinational loops do not occur. In our case, however, we wish to synthesize the fabric itself using standard synthesis tools. Many of these tools (such as those performing timing verification) have difficulty optimizing circuits containing combinational loops. Thus, the synthesizable interconnect fabric needs to be designed without combinational loops.

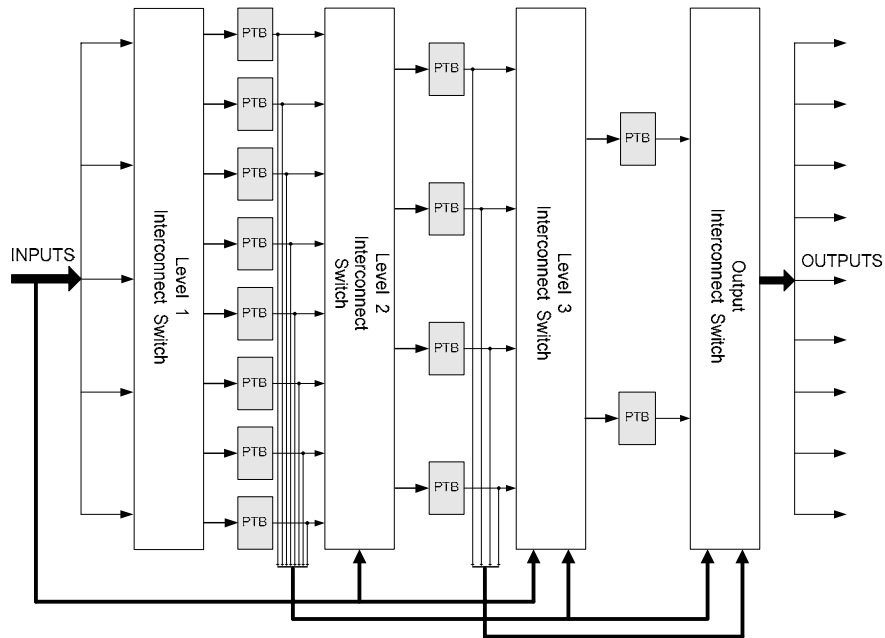
General Structure

Figure 3.3 shows our novel routing architecture that can be used to connect the PTBs. This figure is only a conceptual view since standard ASIC tools are used to automatically create the layout of the fabric. We have considered two interconnect strategies. In the first strategy, shown in Figure 3.3 (a), the PTBs are arranged in a rectangular grid, while in the second strategy, shown in Figure 3.3 (b), the PTBs are arranged in a triangular shape. From [81], logic circuits often have a triangular shape. However, the logic blocks in commercial devices are arranged in a square or rectangular fashion. This “unmatched placement” is normally not a problem, since the routing resources are flexible enough that signals can be routed such that the circuit can efficiently fit in the shape of the core. In a synthesizable architecture, however, because there are constraints on signal flows, a rectangular grid may lead to unused logic blocks. This problem

can be alleviated by creating a programmable logic core that is not square, for example triangular as shown in Figure 3.3 (b).



a) Rectangular PTB Architecture



b) Triangular PTB Architecture

Figure 3.3: Product-Term Block Architectures.

In both routing architectures, the embedded core contains PTBs arranged in several levels. The outputs of PTBs in one level can be connected to the inputs in all subsequent levels (levels to the right) but cannot drive any PTBs in the same level, or preceding levels (levels to the left). This results in a directional architecture, and eliminates the possibility of combinational loops, as described above. Thus, this architecture can only support combinational circuits. Enhancements to the interconnect architecture to support sequential circuits will be described in the next section.

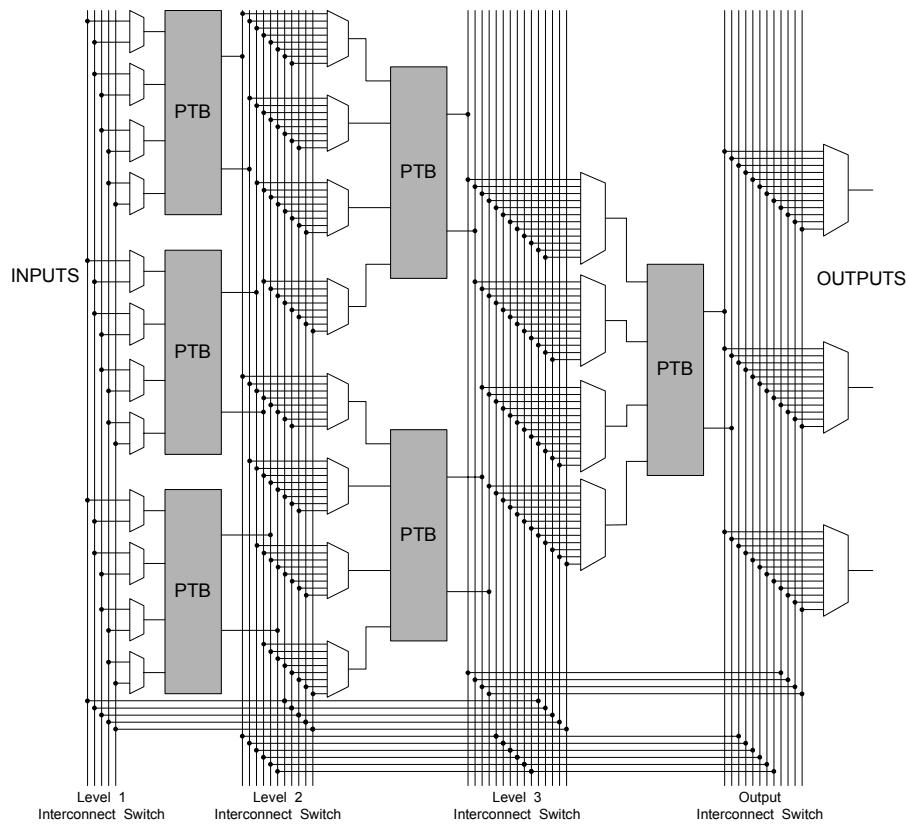


Figure 3.4: Detailed view of routing fabric.

Figure 3.4 shows a detailed view of our proposed directional routing architecture. In place of a central single interconnect matrix, we use multiplexers at each input of the PTB block. The multiplexers allow the PTB inputs to connect to the outputs of all PTBs in previous levels, as

well as the primary inputs. We are able to reduce the size of the multiplexers by taking advantage of the fact that all input connections to a PTB block are equivalent. For a circuit with n primary inputs, to be mapped into a (i,p,o) PTB block, the size of a multiplexer at level l is:

$$Mux\ Size_l = n + \sum_{s=1}^{s=l-1} C_s \cdot o - (i - 1)$$

where C_s denotes the number of PLA blocks at level s . The size of a routing multiplexer is calculated by summing the number of primary inputs with the number of outputs from PTBs in preceding levels, and subtracting the number of inputs in the PTB block.

Allowing a “full-connect” fabric may not seem to be very area efficient, especially since the multiplexers grow in size with the depth of the core. Because of this, [14, 80] suggested a sparsely populated routing fabric used in their synthesizable LUT-based architecture. However, in our case, the number of PTB blocks and the depth of the fabric are comparatively small, meaning the size of the multiplexers do not become too unwieldy. An advantage of a fully connected architecture is that the placement and routing tasks are simplified. Place and route algorithms to support PTB-based fabrics are described in Chapter 5.

3.2 Sequential Product-Term Block Architecture

3.2.1 Logic Blocks

Figure 3.5 illustrates the logic block that supports both combinational and sequential circuits. The structure of this logic block differs slightly from the PTBs used in commercial devices (Figure 2.8). In commercial CPLDs either the registered or non-registered signal is available at one time via a two-input multiplexer. However in our architecture, both registered and non-

registered outputs are available since the multiplexer is removed. This is because of constraints imposed on a synthesizable fabric which will be described in the next subsection.

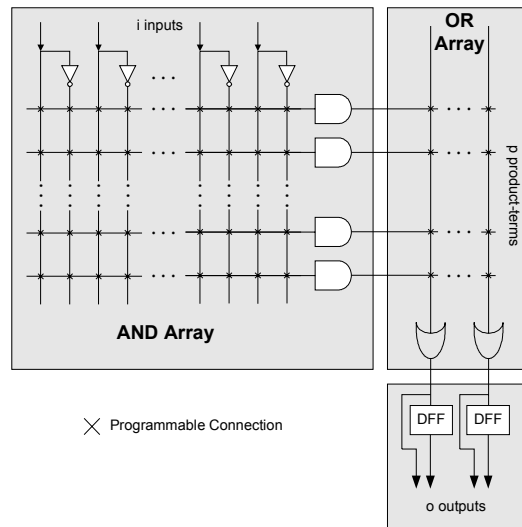


Figure 3.5: Sequential PTB logic block.

The schematic layout of a sequential capable PTB is similar to the schematic shown in Figure 3.2. The main difference is the addition of flip-flops and its accompanying clock tree to provide registered outputs. In this architecture, synthesis and physical design tools will need to deal with two clock domains, one to program the fabric and the other to provide sequential circuit implementations.

3.2.2 Interconnect Architecture

Design Constraints

Standard FPGA architectures support sequential logic through the use of one or more flip-flops embedded in each logic block, as shown in Figure 3.6 (a). Each output of each logic block output can be programmably registered or unregistered. The problem with this approach for a synthesizable programmable logic architecture lies in the interconnect. Applying a directional interconnect to an architecture containing flip-flops as in Figure 3.6 (a) will not suffice, since

there will be no way to implement registered feedback loops; such loops are an important part of most sequential circuits such as state machines (where the state variables must be used as inputs to the next state logic). However, simply adding feedback signals to the architecture to support these registered feedback loops will re-introduce combinational loops in the unprogrammed fabric (through the combinational output of the logic block).

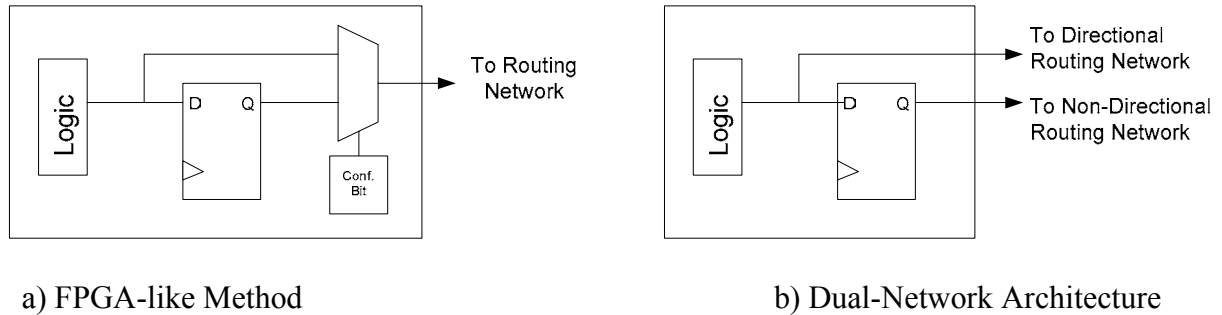


Figure 3.6: Adding Registers to Logic Blocks.

General Structure

To satisfy the design constraints, we modified the logic block as shown in Figure 3.6 (b). Our approach is to provide both the registered output and combinational output signals outside of the PTB, and connect the registered outputs and unregistered outputs to the other PTBs using two separate routing networks. The unregistered network is “directional”, while the registered network can connect an output to a PTB in any level. In this way, the unprogrammed fabric contains loops, but each loop contains at least one flip-flop, meaning the synthesis tools will be able to process the circuit effectively.

We present two methods to support sequential logic. In either method, the directional routing network is still based on the interconnect strategies as described in Section 3.1.2. The PTBs are

still arranged in a rectangular or triangular fashion, with all non-registered PTB outputs in one level directly connected to PTB inputs in subsequent levels via multiplexers. The difference between the two sequential architectures lies in the non-directional routing network.

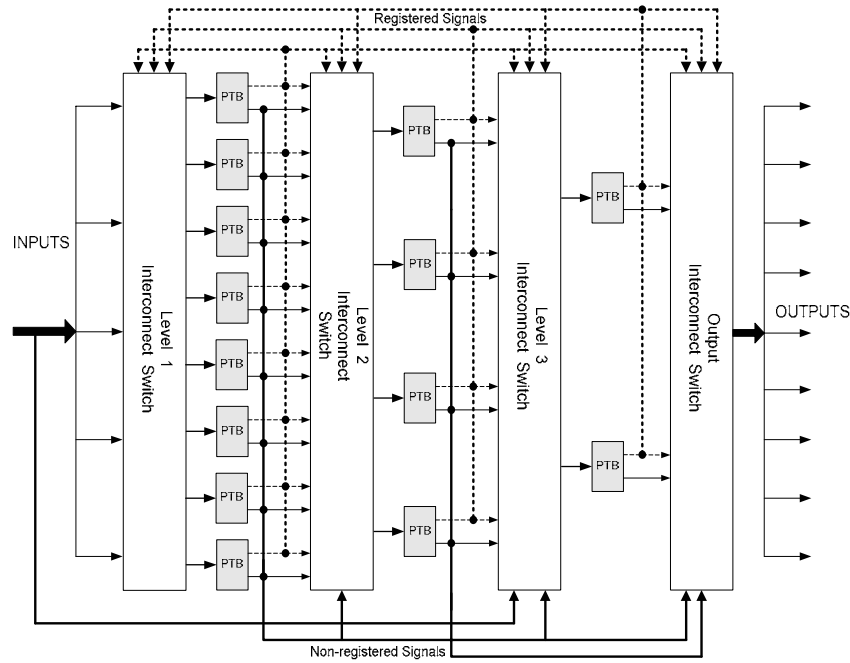


Figure 3.7: Dual-Network Architecture.

Figure 3.7 presents the first architecture called the Dual-Network Architecture. As shown in the figure, the combinational output of each PTB can drive any PTB input in subsequent columns, while the sequential output can drive any PTB in any column. Although the size of this “full-interconnect” architecture will increase quadratically with the number of PTBs, in practice the number of PTBs is small, and this interconnect does not become unwieldy. Unlike the approach in stand-alone FPGA’s, this technique requires two routing networks, which significantly increases the size of the fabric. This area can be reduced by only including flip-flops for some logic block (PTB) outputs. In Chapter 5, we will present experiments to determine how many logic blocks (PTB) should contain registered outputs.

Figure 3.8 illustrates our second architecture, which we will refer to as the Decoupled Architecture. Depopulating the flip-flops as described for the Dual-Network Architecture reduces the flexibility in PTB placement and routing. The Decoupled Architecture combats this by replacing the flip-flops within the logic blocks with a global array of registers. All logic block outputs connect to a number of “general” registers through an array of multiplexers. The outputs of the registers then feed back into the logic block inputs. By not associating any particular register to any specific logic block output, utilization of the registers can be greatly increased. This added flexibility, however, comes at a cost of increase area for signal routing. In Chapter 5, we will determine whether this decoupling leads to an overall improvement in density, and determine how many global registers are required.

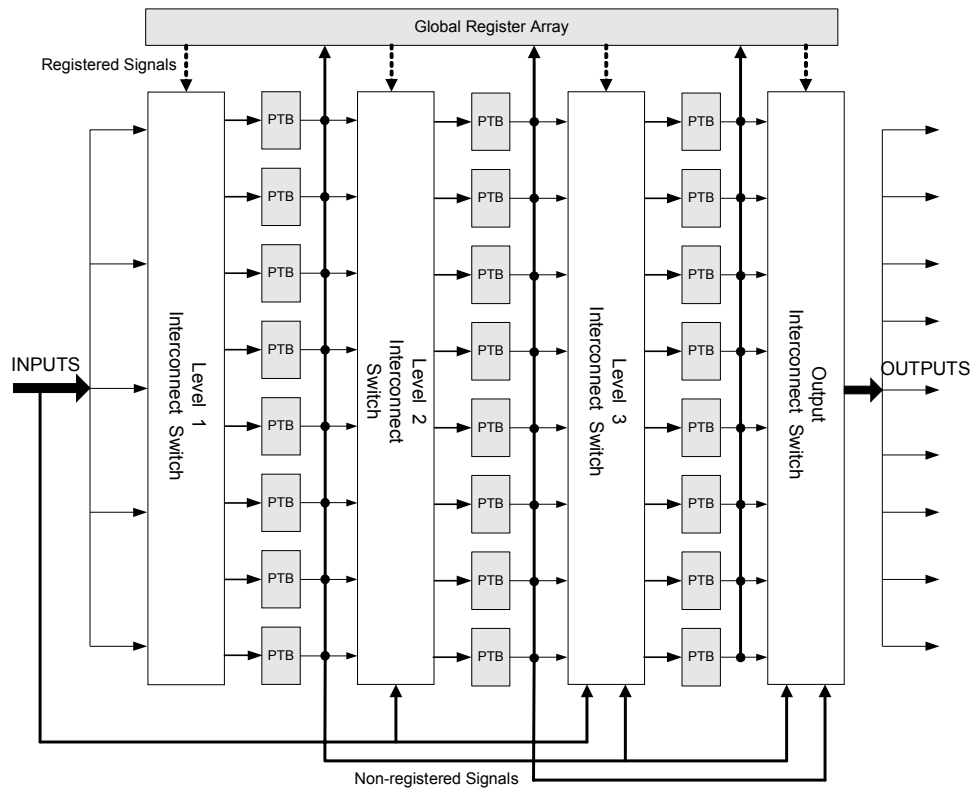


Figure 3.8: Decoupled Architecture.

3.3 Architectural Parameters

3.3.1 High-Level Parameters

There are two classes of parameters we use to describe a specific design within our architectural family: high-level parameters and low-level parameters. Consider a VLSI designer who wishes to employ one of our cores. The designer would have a rough idea of how much logic should fit in the core (and hence, the size of the core in terms of number of LUTs or PTBs) as well as the number of inputs and outputs of the core. The designer can also provide information on the type of circuits, combinational or sequential, to be implemented on the fabric. The designer would use these quantities, which we refer to as high-level parameters, to choose a specific core from a library or a core generator. The high-level parameters are summarized in Table 3.1. Of course, any additional information the VLSI designer can provide would be welcome. The designer of the programmable core can take into account such data and tailor the fabric for a specific user.

Parameter	Symbol
Number of Primary Input Pins	PI
Number of Primary Output Pins	PO
Number of PTBs (or LUTs) in the first level	y
Sequential or Combinational circuit only	s / c

Table 3.1: High-Level Architectural Parameters.

3.3.2 Low-Level Parameters

Low-level parameters, on the other hand, would not normally be specified by the VLSI designer. These parameters describe the details of the core layout. The designer of the core library itself (as opposed to the VLSI designer who uses the library) would like to use optimum values for these parameters in the design of each core in the library. The low-level parameters for our synthesizable PTB-based cores are listed in Table 3.2. Chapter 5 will seek optimum values for

these parameters. The first three parameters, (i,p,o) , characterize the size of the PTB logic blocks. The next two parameters, (r,α) , details the interconnect between the PTBs. r determines the depth of the core for rectangular fabrics and α determines the “drop-off” rate of PTBs for triangular fabrics. The last two parameters describe how many flip-flops should be incorporated into the core. The parameter, v and d , are for the Dual-Network and Decoupled architecture, respectively. Further details about r , α , v , and d are given in Chapter 5.

Parameter	Symbol
Inputs per PTB	i
Product-Terms per PTB	p
Outputs per PTB	o
Rectangular Core: Ratio of levels to number of PTBs in first level	r
Triangular Core: Ratio of PTBs in neighbouring levels	α
Sequential Fabric, Dual-Network Architecture: Number of registered PTBs in each core level	v
Sequential Fabric, Decoupled Architecture: Ratio of global registers to number of PTBs	d

Table 3.2: Low-Level Architectural Parameters.

3.4 Summary

In this chapter we have presented synthesizable product-term based architectures for implementation in SoCs. We have described the constraints standard synthesis tools impose on a synthesizable interconnect, and have presented routing structures that satisfy these constraints. We have presented two types of programmable fabrics. The first supports only combinational circuits and the second supports both combinational and sequential circuits. We have also proposed two interconnect strategies in which the placement of the PTBs is arranged in a rectangular or triangular fashion. Two architectural enhancements to the sequential fabric have also been presented. The sequential Dual-Network Architecture contains routing networks

driven from only PTB outputs, while the Decoupled Architecture's interconnect is driven from PTB outputs and a global array of registers. Finally, we have described the parameters that characterize these architectures. The parameters are separated into two classes, high-level parameters that are specified by the user of the programmable core and low-level parameters that are specified by the designer of the fabric itself. In Chapter 5, we find optimal values of these parameters.

Chapter 4

PLACEMENT AND ROUTING ALGORITHMS

The ability of CAD tools to place and route user circuits on the programmable fabric plays an important role in the density and speed that can be achieved by the programmable core. An efficiently designed programmable architecture is useless without CAD tools that can take advantage of its features. In the previous chapter we presented novel PTB-based synthesizable architectures. These architectures, however, require CAD tools, particularly placement and routing algorithms, to support them. Note, these place and route CAD tools are distinct from the place and route tools used to create the programmable fabric itself. Here, we are referring to the CAD algorithms used after fabrication of the programmable core to implement the user circuits. In this chapter we present novel place and route algorithms used to implement circuits on our PTB-based programmable architectures. We first describe the optimization goals for placement and routing. Next, we will show that, although a simulated annealing-based placement algorithm could be used, such an algorithm does not work well with our architecture. We then present a new, more efficient, and simpler placement algorithm. Finally, we describe the routing algorithm for our architectures.

4.1 Optimization Goals

The goal of placement and routing algorithms is to assign locations to each logic block in the user circuit such that:

- a) Each logic block is assigned to a *unique* PTB in the fabric. If a logic block requires a flip-flop at its output, it is assigned to a PTB with a flip-flop, otherwise, it is assigned to any PTB.
- b) The connections required to implement a user circuit can *likely* be made using available routing resources. In standard FPGA architectures, this may mean attempting to place blocks which are connected to each other close together. In our architecture, it means ensuring that non-registered signals are never connected from right to left. Indeed, if we find a placement that satisfies this constraint, we are *guaranteed* that the circuit will be routable.

In practice, our placement algorithm will be used to map a circuit to an embedded FPGA with a fixed number of PTBs. In our architectural experiments, however, we wish to optimize our architectures, and thus must map each circuit to as small a fabric as possible. Thus, we add the following optimization goal:

- c) The size of the fabric contains as few PTBs as possible, while meeting the aspect ratio and connectivity properties described in the previous section. Rather than minimize the number of PTBs directly, we minimize *the number of PTB levels*. In addition to minimizing the size of the fabric, the fewer levels in a fabric, the faster the fabric will be.

4.2 Placement Algorithm

In the following subsection, we describe two possible placement algorithms for the PTB-based architectures described in Chapter 3.

4.2.1 Simulated Annealing-Based Algorithm

Programmable logic placement algorithms often employ simulated annealing-based algorithms. Simulated annealing is based on the process of industrial annealing used to cool molten metal. The pseudo-code for a generic simulated annealing-based placer is shown in Figure 4.1. At the heart of the algorithm is the calculation of a *cost* of swapping the positions of two logic blocks. The cost function evaluates the quality of a possible placement over the previous placement. If the quality, such as wire length, congestion, and/or delay improves, then the swap is always taken. If not, there is still a chance the move is accepted even though it makes the placement worse. The probability of accepting a bad move is primarily dependant on *temperature*. At first, the temperature is very high and almost all moves are accepted. However, as the algorithm progresses, the temperature gradually decreases so the probability of accepting a bad move is lower. The reason why bad moves are initially accepted is to prevent the algorithm from getting stuck in a local minimum in the cost function.

```
set placement to some random placement;
set initial temperature;

while (some exit criteria = false)
{
    while (some inner loop criteria = false)
    {
        new placement = swap two logic blocks in placement;
        cost = cost in new placement - cost in placement;
        accept = random (0,1);
        if (accept <  $e^{-cost / temperature}$ )
            placement = new placement;
    }
    decrease temperature;
}
```

Figure 4.1: Pseudo-code for simulated annealing.

A simulated annealing-based placement algorithm can be employed in the PTB-based architecture. The cost function needs to be modified so it never allows placement that violates the directional constraints. The sparse number of flip-flops in the architecture can also be accounted for by applying a penalty for placing a logic block that does not require a registered output onto a PTB with a flip-flop.

There are a number of disadvantages to a simulated annealing-based placement approach, however. First, simulated annealing-based algorithms typically have long run times. Randomly swapping pairs of logic blocks and calculating the cost of each swap in an architecture that contains only a few logic blocks is inefficient. Since our architecture has only a small number of PTBs (compared to stand-alone FPGAs), it would be tempting to simply reduce the number of iterations of the algorithm. This, however, would increase the risk of getting stuck in a local minima. Therefore, we consider an alternative placement algorithm described in the next section.

4.2.2 Greedy-Based Algorithm

An alternative placement algorithm is presented in this section. Although the placement algorithm is based on a greedy algorithm, results have been shown to be very close, or exactly, the same as optimal results (obtained by manually placing each logic block into the programmable fabric).

High-Level Description

The algorithm starts by initially assigning the netlist of PTBs to logic blocks in the programmable core in a breadth-first manner, based on the signal dependencies in the circuit. Such an assignment may result in more PTBs assigned to a level than there are physical PTBs in

that level, or may lead to a mismatch between registered PTBs and registers in the architecture. To resolve these conflicts, levels are visited in order (starting from the inputs), and the excess PTBs are demoted to subsequent levels. The selection of PTBs from within a level to demote is done on the basis of the PTB's *required arrival time*. A flowchart illustrating the process is shown in Figure 4.2.

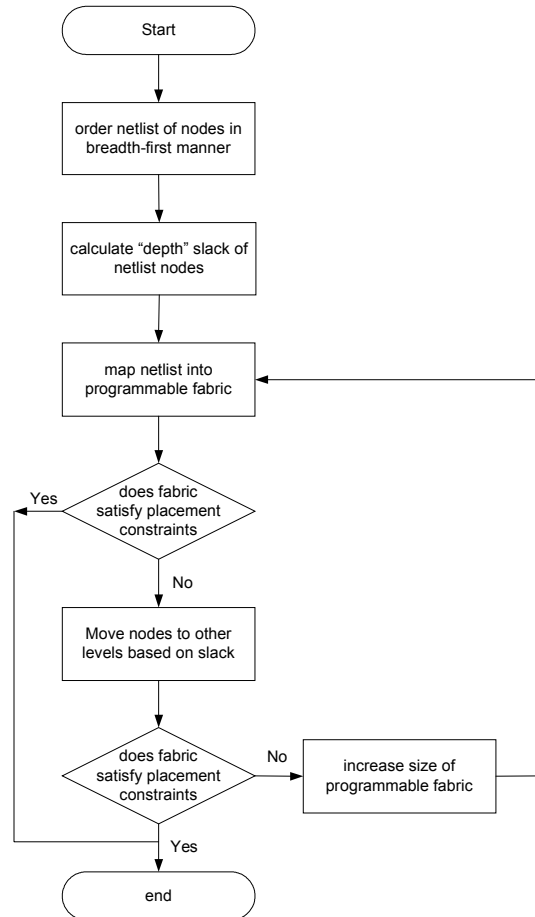


Figure 4.2: High-level description of placement algorithm.

Detailed Description

In this section, we describe our placement algorithm in greater detail. The pseudo-code for the algorithm is shown in Figure 4.4. The algorithm has two phases. The first phase begins with a

breadth-first ordering of the logic blocks in the netlist. The nodes are ordered topologically so that a *level* can be assigned to each node. The level of each node is defined as:

$$level(v) = 1 + \max_{u \in fanin(v)} (level(u))$$

where $fanin(v)$ is the set of all nodes that are inputs to node v . Assuming each node has a delay of 1 time unit, the largest level in the network will then be equal to the critical path length of the circuit.

Next, the *required time* of each node is calculated. The required time is the time before which a signal must arrive to avoid increasing the critical path delay. We start by assigning all the primary output nodes a required time equal to the critical path length. From there, the rest of the nodes are visited recursively and the required time is calculated as:

$$R(v) = \min_{u \in fanout(v)} (R(u) - D(u))$$

where $R(v)$ is the required time for node v , $fanout(v)$ is the set of nodes that are outputs of v , and $D(u)$ is the delay of node u , which is always 1. The required time of a node can also be viewed as the maximum level a node can be placed without violating the unidirectional constraint of the fabric. An example of the required times is shown in Figure 4.3.

Most placement algorithms, however, are based on *timing slack* instead of required time [59]. Timing slack of a node is calculated by subtracting the node's required time from its *arrival time*. Arrival time is the latest time input signals can arrive at node v without violating timing. It is calculated as:

$$A(v) = \max_{u \in fanin(v)} (A(u) + D(u))$$

where $A(v)$ is the arrival time for node v , $fanin(v)$ are the fanins nodes of v , and $D(u)$ is the delay of the node. Our algorithm does not use timing slack because of the directional placement constraints. The algorithm requires information about the logic depth that a node can be placed and, unlike timing slack, required time contains such data. Another reason why timing slack is not used is to avoid repeated slack calculations once a node is promoted to higher levels.

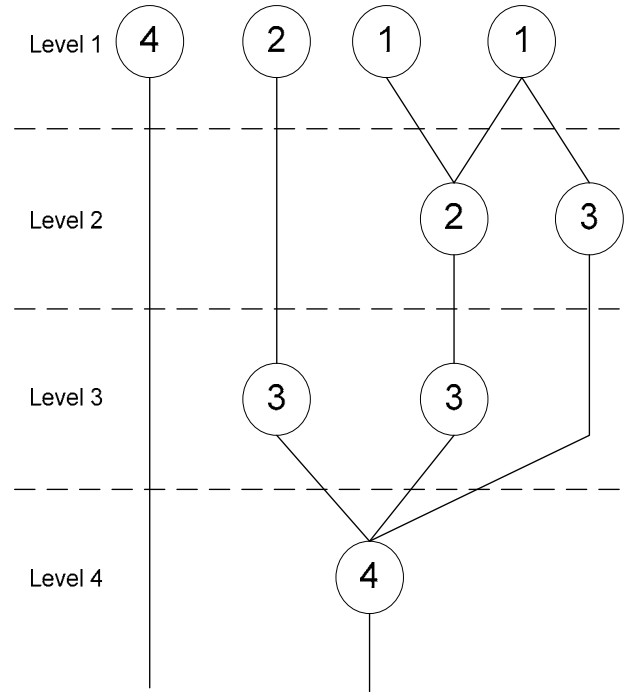


Figure 4.3: Example of required time calculation.

In the second phase of the algorithm, the nodes are assigned specific sites on the fabric, based on their required time values. This requires determining a fabric size and shape. As described in Section 4.1, in our algorithm, we wish to find the minimum sized fabric for a given user circuit. Thus, we start with a fabric with “just enough” logic blocks to implement the user circuit. Each logic block is assigned a specific level in the fabric, based on its topological level (so, a node with a level n is assigned to level n in the fabric). Of course, this may result in some levels being “overcommitted”; that is, more logic blocks may be assigned to a level than there are PTBs in

that level. In addition, it may result in logic blocks with flip-flops being assigned to levels where there are insufficient flip-flops.

To resolve this, we visit each level, starting from the inputs, and resolve the contention at that level. We deal with registered and non-registered nodes separately, starting with registered nodes. Suppose we are visiting level l , which has n_l available sites with flip-flops. Further, suppose that, initially, n_c registered logic blocks are assigned to level l . If $n_c \leq n_l$, then there is no contention. On the other hand, if $n_c > n_l$, we *promote* the $(n_c - n_l)$ registered logic blocks with the highest required time value from level l to level $l+1$. This frees up the contention for flip-flops in level l , but of course increases the number of flip-flops in level $l+1$. We do the same thing with non-registered logic blocks; if there are more logic blocks assigned to a level than there are sites available, the excess logic blocks are promoted to the next level. This is repeated for each level.

During the mapping of the last level, if no logic blocks are promoted, we have found a valid solution. On the other hand, if the last level is not sufficient to implement the logic blocks assigned to that level (and promoted from previous levels), we start the process again, with a larger fabric. Note that this is preferable than simply adding another level, since doing so might result in very deep but small fabrics, which may lead to slow circuit implementations.

```

/* Phase 1 */
order nodes breadth-first;
for each node in the network
{
    calculate required time;
}

/* Phase 2 */
find minimum size fabric based on number of network nodes;
directly map circuit into fabric to create initial mapping;
if (mapping does not result in connection for resources)
{
    done!
}
else
{
    for each level l in mapping
    {
        while (contention for resources in level l)
        {
            if (contention of PTBs only)
            {
                for each node in level l
                {
                    target node = choose node with highest required time,
                    followed by nodes in highest level,
                    and then followed by nodes that
                    do not require registered outputs;
                }
            }
            if (contention of register elements)
            {
                for each node in level l
                {
                    target node = choose node with highest required time
                    and requires registered outputs;
                }
            }
            move target node to level l+1 of mapping;
            decrease number contentions for resources in level l;
            increase number of resources used in level l+1;
        } //while contention for resources in level l

        check level l for violations in placement constraints;
        if (violations present)
        {
            move offending nodes to level l+1 of mapping;
            decrease number of resources used in level l;
            increase number of resources used in level l+1;
        }
    } // for each level l in mapping

    if ( mapping satisfies directional constraints
        and contains no contention for resources )
    {
        done!
    }
    else
    {
        increase fabric size;
        try placement again with new fabric size;
    }
} //else

```

Figure 4.4: Detailed description of placement algorithm.

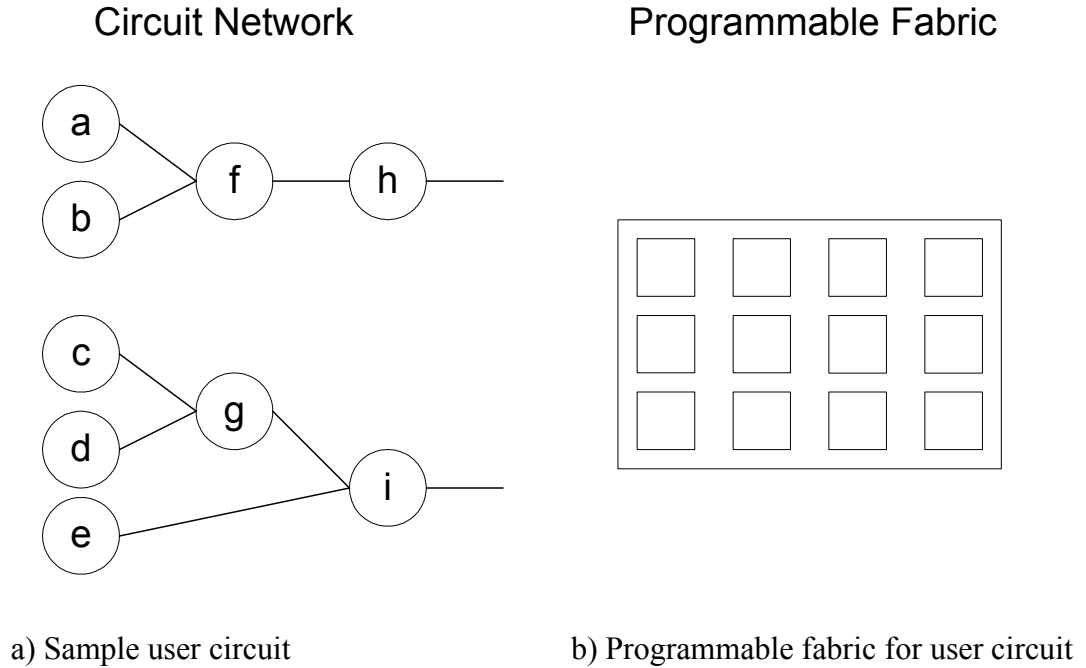


Figure 4.5: Example circuit mapping onto programmable fabric.

This algorithm, while conceptually simple, may lead to invalid solutions. Consider the example circuit shown in Figure 4.5 (a) to be placed in a PTB-based fabric consisting of an array of 3x4 logic blocks as illustrated in Figure 4.5 (b). Each placement step is illustrated in Figure 4.6. In the first iteration, the algorithm assigns logic blocks to PTBs based on the node's required time value, as shown in Figure 4.6(a). Visiting each level in order, we see that the first level has been assigned five logic blocks, yet there are only three available sites. Thus, the algorithm moves two nodes to the second level. Since node e has the highest slack, it is moved, along with one of the other nodes (in this example, we arbitrarily assume a is chosen). The result is shown in Figure 4.6 (b). We then visit level 2 and see that this level now has one more logic block than there are sites available, so the node with the highest slack, which is node e, is moved to level 3, as shown in Figure 4.6 (c). This creates an invalid placement, since node *f* requires input from node *a*, but node *f* is located in the same level as node *a*. If this occurs, our algorithm will

promote node f to the next level, as shown in Figure 4.6 (d). The placement algorithm then continues onto the next level, level 3. Again, it finds contention for resources so it shifts the node with the highest slack as shown in Figure 4.6 (d) (for this example, assume that it arbitrarily chooses node h). It then finds the directional constraint has been broken once more, so it promotes the violating node, i , to the next level (as shown in Figure 4.6 (f)). Finally, a satisfactory placement is obtained (as shown in Figure 4.6 (f)).

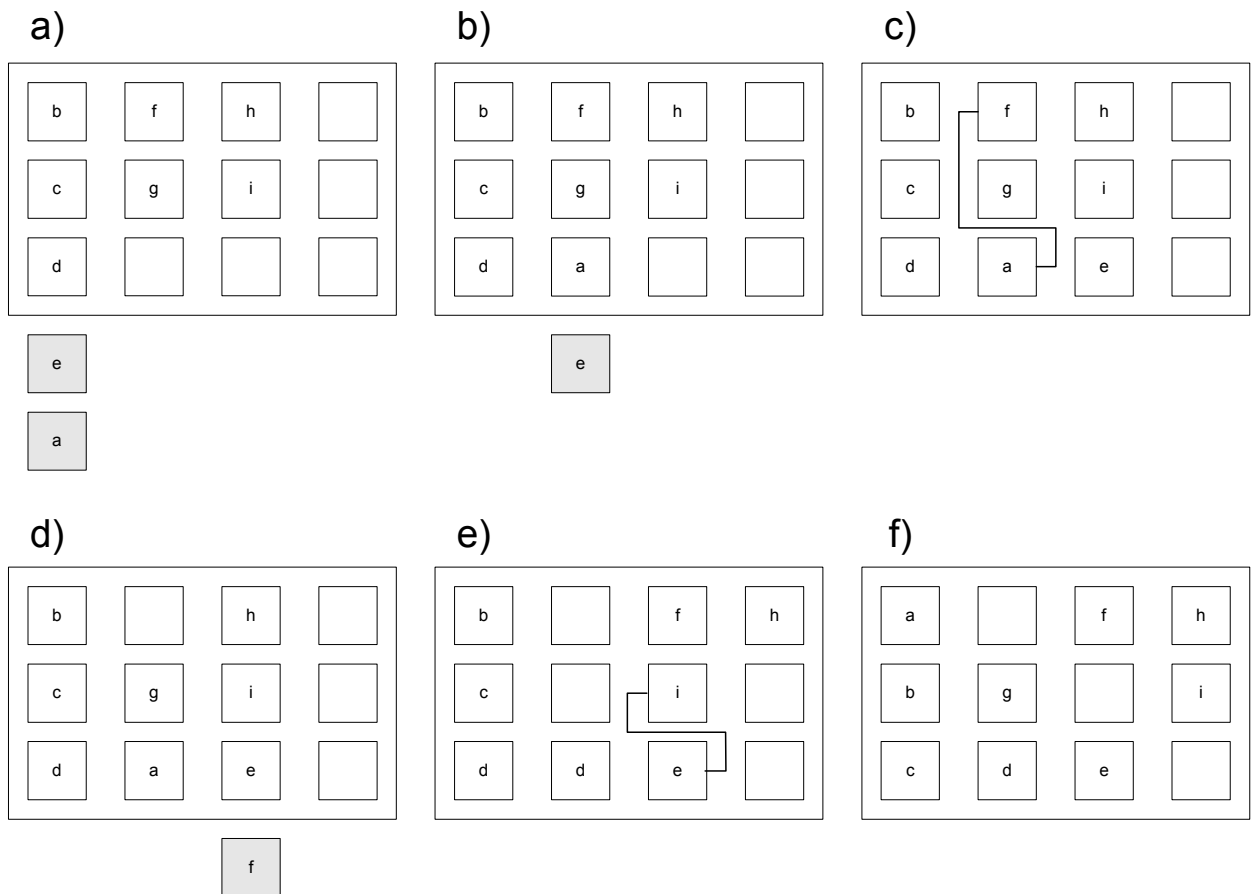


Figure 4.6: Example circuit placement steps.

4.3 Routing Algorithm

Because of the limited interconnect found in most commercial stand-alone FPGAs, the routers that target these FPGAs typically use complex negotiated-congestion based routing algorithms.

In our architecture, the interconnect fabric is fully-connected; that is, any logic block output can be connected to any logic block in subsequent levels without using any shared resources. Thus, the routing task is much simpler.

Although each PTB output can be connected to any other PTB in a subsequent level, it can *not* necessarily be connected to every *pin* on each of these target PTBs. As described in Chapter 3, by taking advantage of the fact that each input pin to a given PTB is equivalent, the multiplexers at the inputs of each input pin can be simplified somewhat. Consider the example in Figure 4.7. This example shows a four-input PTB which can be driven with seven inputs (each input coming from a different PTB in a preceding level). An architecture in which each signal could be connected to every input pin would require four 7-input multiplexers (one per input pin). On the other hand, by taking advantage of the fact that all the input pins are equivalent, Chapter 3 shows that four 4-input multiplexers are sufficient to implement all required connections.

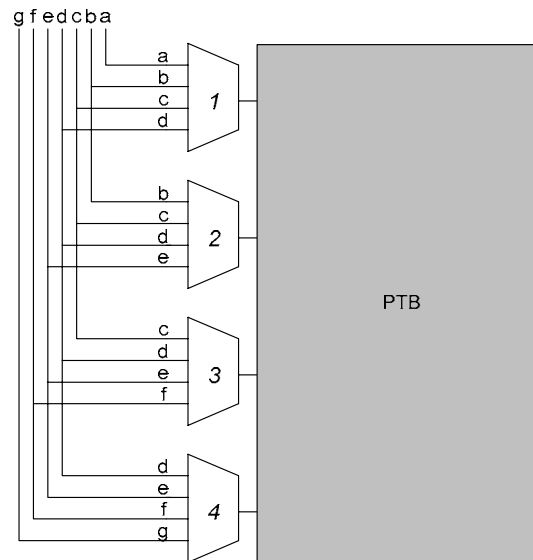


Figure 4.7: Example PTB multiplexer signal connection.

Depopulating the input multiplexers in this way does not decrease the overall routability of the core, but it does place constraints on which signals can be assigned to each input pin. Thus, for each PTB, we need to solve the following pin assignment problem:

Given: a PTB with n_p input pins ($p_0, p_1, \dots, p_{n_p-1}$) and a circuit in which n_s signals ($s_0, s_1, \dots, s_{n_s-1}$) are to be connected to this PTB ($n_s \leq n_p$),

Find: an assignment of each signal s_k ($0 \leq k < n_s$) to an input pin p_j ($0 \leq j < n_p$) such that all n_s connections can be made simultaneously.

Given the architecture described in Chapter 3, this problem can be solved by finding an appropriate ordering of the signals ($s_0, s_1, \dots, s_{n_s-1}$). Consider the example of Figure 4.8. There are seven potential input signals to a four-input PTB. Suppose signals b, e, f , and c are to be connected to this PTB. If we connect signal b to pin 1, signal e to pin 2, and signal f to pin 3, we can not connect signal c to the PTB, as shown in Figure 4.8 (a). However, if the connections were re-ordered, such that signal b connects to pin 1, signal c connections to pin 2, signal e connections to pin 3, and signal f connects to pin 4, then all connections can be made, as shown in Figure 4.8 (b). In general, a correct assignment is guaranteed to be found by:

- a) Ordering the input signals s such that, for all $0 < i < j < n_s-1$, $m(s_i) < m(s_j)$, where $m(s_k)$ is the number of the lowest-number (i.e., highest in the diagram of Figure 4.8) multiplexer which has signal s_k as an input.
- b) For each signal in order, assign the signal to the pin corresponding to the lowest-number available multiplexer that has s_k as an input. Once a pin has been assigned a signal, the associated multiplexer is no longer available for future signals in the ordered list.

This algorithm is applied for each PTB independently. The order in which the PTBs are processed does not matter.

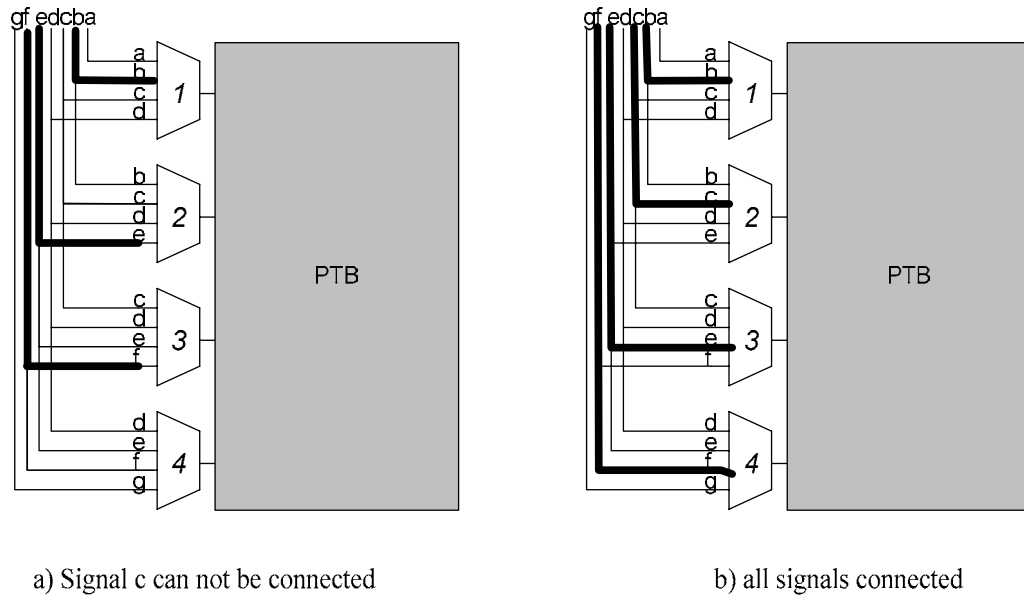


Figure 4.8: Example PTB multiplexer signal routing.

4.4 Summary

In this chapter, we have presented place and route algorithms for the product-term based architectures in Chapter 3. We started by describing the optimization goals of our algorithms. We then discussed why a simulated annealing-based placement algorithm is not suitable for our architectures. We then described a novel greedy-based placement algorithm. Finally, we described our routing algorithm and determined that the rich interconnect architecture results in an easy implementation. In the next chapter, we apply our place and route algorithms to determine optimal architectural parameters.

Chapter 5

LOW-LEVEL ARCHITECTURE PARAMETER OPTIMIZATION

In this chapter, we present our experimental methodology to investigate area and delay efficiencies of the Product-Term Based architectures described in Chapter 3. The CAD algorithms presented in Chapter 4 will be used to help facilitate our evaluation. This chapter finds optimum values of *low-level* parameters presented in Chapter 3. We do not attempt to find optimum values for the *high-level* parameters, since these are parameters that would usually be determined by the VLSI designer depending on the applications expected to be mapped to the programmable logic core. This chapter investigates each low-level parameter and identifies their impact on the area and delay of the resulting programmable logic core. In particular, this chapter focuses on the following:

1. We first determine optimal product-term logic block parameters to be used in our architecture. Such parameters include the number of inputs, product-terms, and outputs in each product-term block (PTB).
2. We then proceed to calculate the optimal interconnect parameters for combinational-only PTB architectures. More specifically, we determine the optimal interconnect arrangement for rectangular and triangular fabrics. A comparison of the two interconnect strategies follows next.

3. After we derive our proposed interconnect parameters for the combinational PTB fabrics, we investigate whether these values still hold true for sequential PTB architectures. We then establish optimal flip-flop placement parameters for sequential PTB fabrics. In particular, we reveal the best ratio of flip-flops to logic blocks for the Dual-Network and Decoupled sequential PTB architectures. A comparison of the two flip-flop assignment methods is presented afterwards.
4. Finally, we discuss the limitations of the experimental results that we obtained.

Before we present our experimental findings, however, a description of our experimental framework and evaluation techniques is presented. This chapter ends with a summary of our results in Section 5.6.

5.1 Experimental Framework

Figure 5.1 illustrates the experimental methodology we used to evaluate PTB-based architectures. We used benchmark circuits from the Microelectronics Center of North Carolina (MCNC). These circuits are separated into two categories, combinational and sequential. Fifty-eight combinational circuits with sizes ranging from 10 to 300 equivalent four input LUTs (4-LUTs) were used to evaluate combinational-only PTB architectures. Forty-seven MCNC sequential benchmark circuits, each containing between 14 and 296 equivalent 4-LUTs, were used for sequential-capable PTB-based fabrics. Generally small circuits were chosen since they are the types of circuits envisioned for synthesizable architectures. Larger sized circuits would likely be implemented using pre-fabricated hard programmable logic cores.

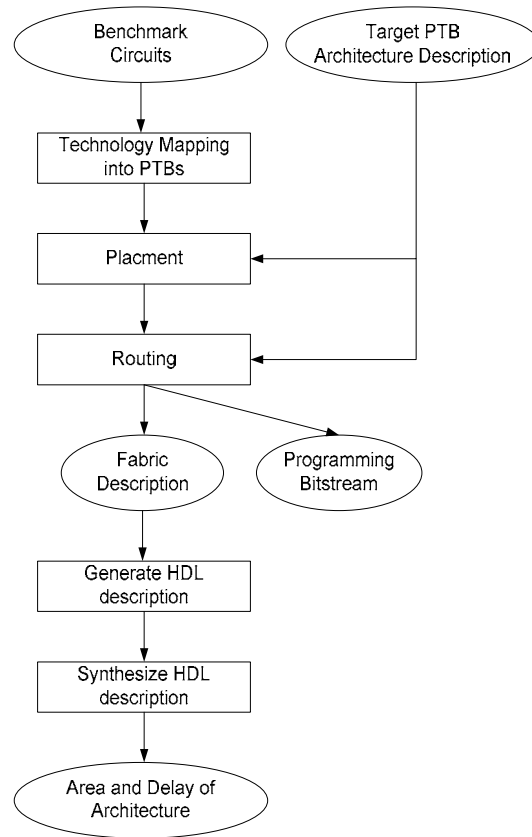


Figure 5.1: PTB Architecture Experimental Methodology

Each benchmark circuit was first mapped into PTBs using PLAmmap [51]. The PTB-based circuit netlist, along with a description of the target PTB architecture, was then placed and routed using the algorithm described in Chapter 4. The output of the placement and routing tool is a programming bit stream file and the size, area, and depth of the target PTB architecture.

After the target PTB-based fabric characteristics were obtained, a HDL description (Verilog) of the architecture was generated. Next, a synthesized behavioural description of the core was created using Synopsys Design Compiler with 0.18 μ m technology wire load models from TSMC and standard cell libraries from Virtual Silicon. Area and delay numbers were then gathered to evaluate the architecture. Although more accurate area and delay values can be obtained from a

physical layout, we and [80] have determined that there is a good fidelity between synthesized results and physical chip implementation.

5.2 Product-term Logic Block Parameter Optimization

In this section, we seek the optimal number of inputs, i , product-terms, p , and outputs, o , of a product-term block (PTB). Optimization of the three parameters was performed in a specific order. The number of product terms was optimized first, since product terms account for a large amount of total PTB area and was found to greatly influence the utilization efficiency of the PTB blocks [Error! Bookmark not defined.]. After obtaining the optimal number of PTB product terms, we focused on finding the optimal number of PTB inputs and outputs. The number of inputs was believed to have a larger impact on area and delay compared to the number of outputs, so it (number of inputs) was optimized next.

5.2.1 Number of Product Terms, p , per PTB

To find the optimal number of PTB product terms, p , we performed an experimental parameter sweep ranging from 6 to 21 terms. The baseline PTB architecture was initially fixed with the following values: PTB logic block parameters $i=12$, and $o=3$; and triangular interconnect structure with $\alpha=0.5$. Experimentally, we confirmed these initial parameter assumptions were a good choice. Next, each combinational MCNC benchmark circuit was mapped onto the PTB architecture and the minimum core size recorded.

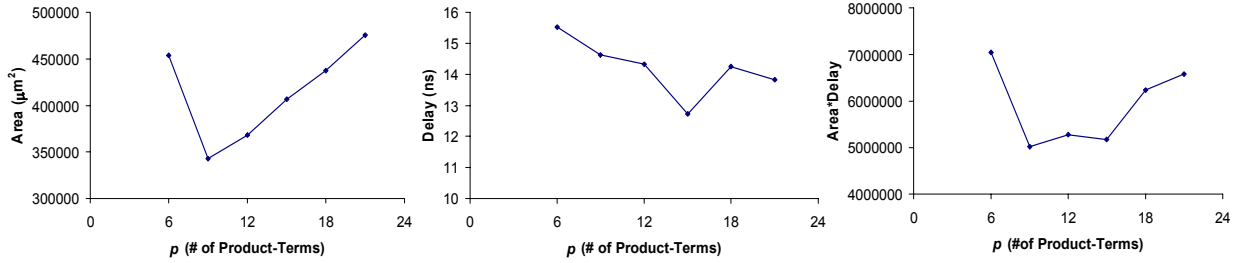


Figure 5.2: Number of product terms per PTB.

For each value of p , we measured the area and delay after synthesis and plotted the geometric average. Figure 5.2 shows the area, delay, and area*delay product of the results. The best area*delay value for p is 9. However, we have observed small circuits tend to prefer a smaller p , while larger circuits tend to prefer a larger p . We repeated our experiments, but partitioned the benchmark circuits into two sets – one set for small circuits (less than 50 equivalent 4-LUTs) and the other set for larger circuits. The results are shown in Figures 5.3 and 5.4. We see small circuits prefer $p=9$, with $p=12$ or 15 resulting in a 12-13% degradation. Breaking down the graph into area and delay components, $p=12$ or 15 results in a 23% degradation in area compared to 9 product terms. For larger circuits, we see $p=18$ or 15 provides the best area and delay results. When $p=9$, we get a 11% increase in area*delay product. As a result, we propose that cores aimed at small circuits use PTBs with $p=9$, and cores aimed at larger circuits use PTBs with $p=18$. This combination results in a 5% improvement in area*delay product than would be obtained by just setting p to 9 for all cores.

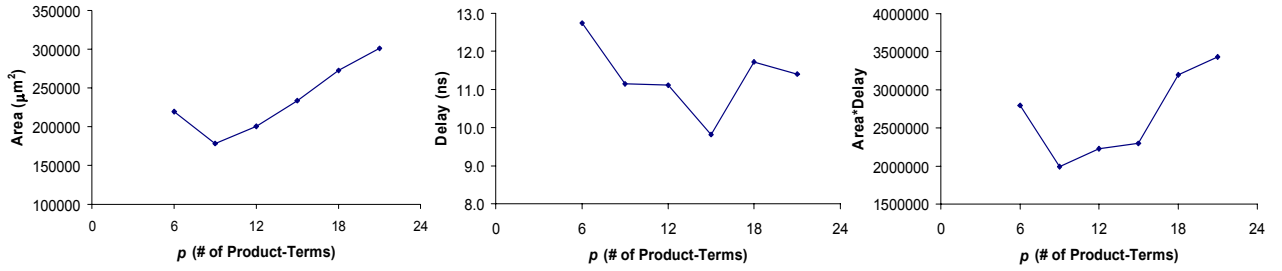


Figure 5.3: Number of product terms per PTB (small circuits).

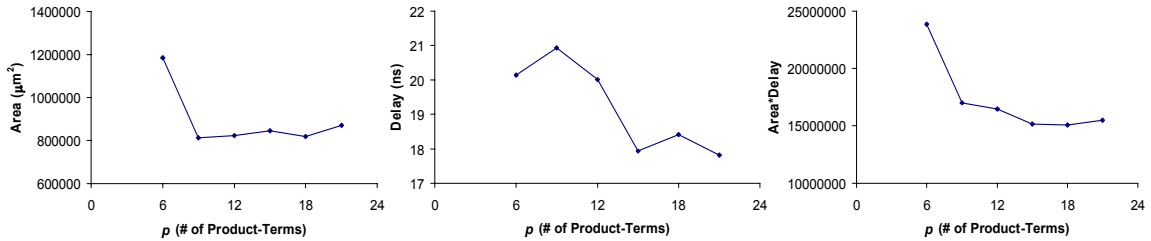


Figure 5.4: Number of product terms per PTB (large circuits).

5.2.2 Number of Inputs, i , per PTB

We next find the optimal number of inputs per PTB. To perform this experiment and subsequent experiments, we substitute the architectural parameter assumptions with the optimized values we determined previously. For example, in this experiment the number of product terms per PTB was set to $p=9$ or 18 , as was obtained in Section 5.2.1. The baseline PTB architecture consists the following parameters: PTB product terms $p=9$ or 18 , PTB outputs $o=3$, and PTB triangular interconnect arrangement with $\alpha=0.5$.

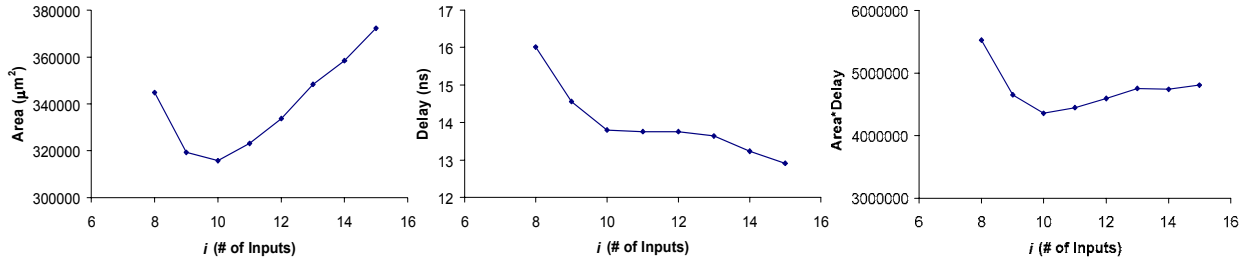


Figure 5.5: Number of inputs per PTB.

This experiment consists of sweeping the parameter, i (number of PTB inputs), from 8 to 15 inputs; recording the respective area and delay values; and calculating and plotting the geometric average over all the combinational benchmark circuits. Figure 5.5 shows the results. We see delay decreases as more inputs are added to the PTB. This is because increasing the number of inputs allows more logic to be packed into a single PTB. By increasing the capacity of the PTB, logic depths of mapped circuits gets shorter and consequently the delay decreases. In terms of area, experimental results show a minimum at $i=10$. There are two competing trends in the area graph. If the number of inputs per PTB is too low, then more PTBs are required to map user circuits. However, if there are too many inputs per PTB, then some of the inputs may not be used. Overall, $i=10$ results in the best area*delay product. Interestingly, this finding is similar with results obtained from [Error! Bookmark not defined.] but for a different architecture.

5.2.3 Number of Outputs, o , per PTB

To find the optimal number of PTB outputs, o , we used a similar procedure. The number of outputs was swept from 1 to 5, while fixing the PTB block parameter i to 10, p to 9 or 18, and assuming a triangular interconnect structure with $\alpha=0.5$. The experimental results are summarized in Figure 5.6. Again there are two competing trends for area. Area increases if 1) there are too few outputs because this leads to increased demand for logic blocks, and 2) there

are too many outputs because this leads to inefficiencies in logic block packing. These two factors lead to an area minimum at $o=3$. Delay increases after $o=3$ because increasing the number of outputs increases the fanout of the AND gates in the PTB to drive the additional OR gates. Adding additional inputs, as was done in Section 5.2.2, however, does not affect the fanout of the AND gates in the PTBs. As a result, the delay does not increase when more inputs per PTB are added. In summary, setting the number of outputs per PTB to be $o=3$ results in the best area*delay product; the next best point $o=4$ results in a 7% area*delay degradation. Coincidentally, this result is also similar to results published by [Error! Bookmark not defined.], but again for a different architecture.

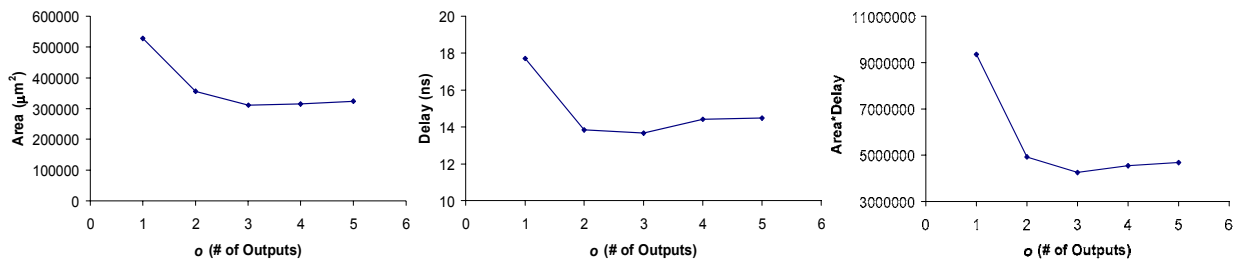


Figure 5.6: Number of outputs per PTB.

5.3 Product Term based Interconnect Parameter Optimization

In the previous section optimal product term logic block parameters were determined. This section uses seeks optimal interconnect parameter values. The following subsections focuses on determining the optimal value of r and α for rectangular and triangular fabrics, respectively. Both combinational and sequential programmable logic cores are evaluated. In all experiments, the PTB logic block parameters found in Section 5.2 are used.

5.3.1 Value of r for Combinational Rectangular Fabrics

In this experiment we seek the optimal value of r for rectangular cores. The parameter r is defined as the aspect ratio between the number of logic levels to the number of PTBs in each level. The depth of a rectangular core is determined as, $Depth = r * n_y$, where n_y is the number of PTBs per column and r is the parameter we wish to optimize. Figure 5.7 illustrates how r affects the depth of the programmable core; the square grey boxes are the PTBs.

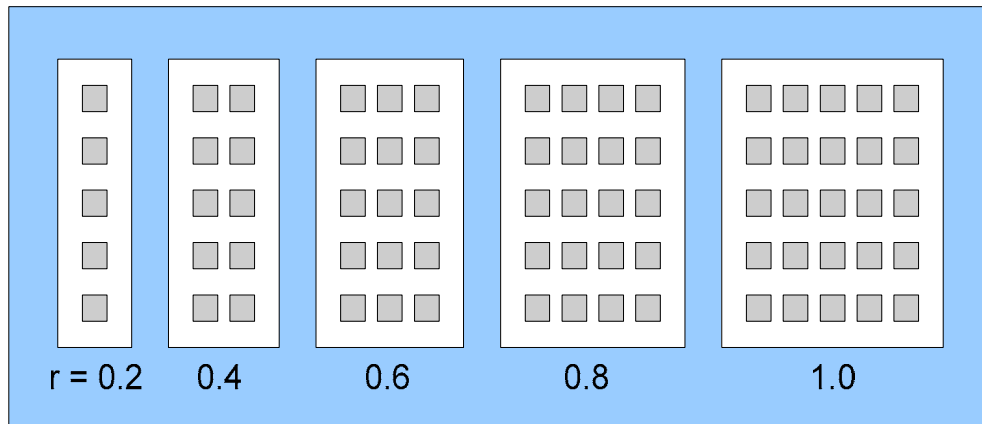


Figure 5.7: PTB rectangular architecture parameter, r .

This experiment used a similar methodology to that in Section 5.2. The same set of combinational benchmark circuits were used, and PTBs of either $(10,9,3)$ or $(10,18,3)$ were used (depending on the core size, as described previously). One difference, however, is we have used depth instead of estimated delay as a metric value. Since these experiments (and all following experiments) all used the same sized PTB logic cells and because PTBs contribute to the majority of delay in the programmable fabric, depth values are well correlated with delay results.

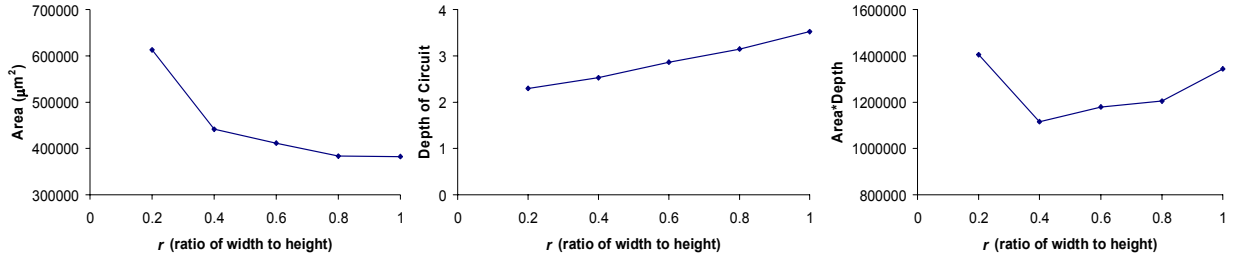


Figure 5.8: Value of PTB rectangular architecture parameter, r .

Figure 5.8 shows the impact of r on area, circuit depth, and area*depth product averaged over our benchmark circuits (again, geometric average is used). As the graph shows, as r increases, the number of levels in the core increases, leading to a longer delay. On the other hand, the area decreases as r increases. A shallow core (small r) places more constraints on the placement of logic, since more stringent precedence relationships must be obeyed (recall, each PTB can drive PTBs in subsequent levels only). All the benchmark circuits, however, were placed and routed successfully by increasing the size of the programmable fabric until it supported the design. Overall, a value of $r=0.4$ gives the best area*depth result.

5.3.2 Value of α for Combinational Triangular Fabrics

We next investigate the effect α has on triangular cores. The parameter α is defined as follows: Given a triangular core that has y PTBs in the first level, there are $n_i = \alpha^{i-1}y$ (rounded to nearest integer) PTBs in the i^{th} level. The number of levels in a triangular core depends on α indirectly. All triangular cores have a minimum of 2 levels. For $y > 3$, there are x levels, where x is the smallest value for which n_x is less than or equal to 3. Figure 5.9 illustrates how α affects the structure of the programmable core; the square grey boxes are the PTBs.

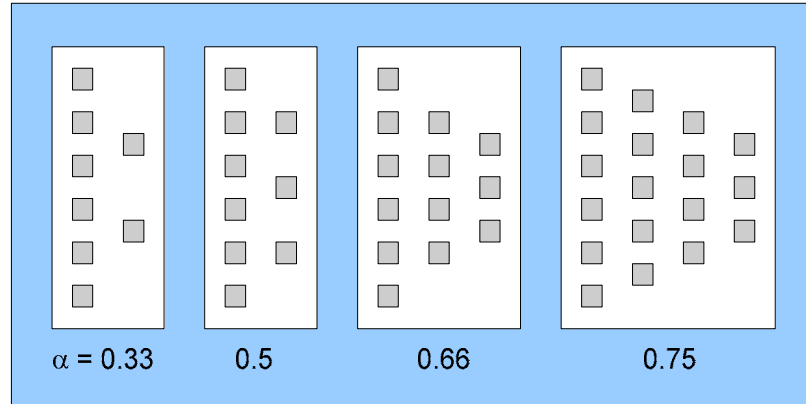


Figure 5.9: PTB triangular architecture parameter, α .

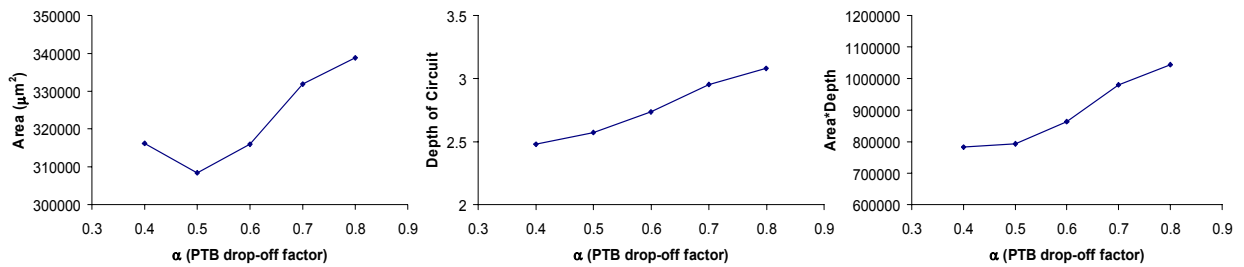


Figure 5.10: Value of PTB triangular architecture parameter, α .

Figure 5.10 shows the impact of α on area, depth, and area*depth product of our benchmark circuits. As the graphs show, a value of $\alpha=0.4$ to 0.5 is a good choice, with a 1.3% *relative difference* in area*depth product. Relative difference is defined as the percentage of improvement between two data points. A high relative difference value indicates the two data points have large performance variations; while a low relative difference value indicates similar improvements can be obtained with either data values.

We see that both delay and area increases for α larger than 0.5 . A large α implies a larger depth and more PTBs, thereby increasing both delay and area. When α is less than 0.5 , mapping depth decreases, but area increases. This is because there are not enough levels to map the benchmark

circuits for small α , so thus the number of PTBs in the first level must increase to provide the required mapping depth. Increasing the number of PTBs in the first level selects a larger size programmable fabric.

5.3.3 Comparison of Triangular and Rectangular Combinational Architectures

Comparing the results from Figures 5.8 and 5.10, we can see that the best triangular core leads to 30% smaller area but a 2% larger depth (and hence delay) than a rectangular core, on average. The best area*depth product is 29% lower in a triangular core than in a rectangular core. The result that triangular-shaped programmable cores are more efficient than rectangular-shaped cores has been observed for lookup-table based architectures [14]. Also, [81] observed logic circuits often have a triangular-shape. In standard FPGAs, this is not a problem, since the routing resources are flexible enough that the physical implementation of a circuit need not match the shape of the circuit. In synthesizable architectures, however, such as that described in [14] and ours, the signal flow is directional (signals can only flow from left to right). As a result, non-rectangular fabrics may lead to a more efficient implementation.

5.3.4 Value of r for Sequential Rectangular Fabrics

The remainder of this section revisits optimization of the parameters r and α for rectangular and triangular cores containing sequential elements. We use a sequential PTB fabric employing the Dual-Network interconnect architecture described in Chapter 3. In this architecture flip-flops are inserted into every logic block. The registered outputs of the PTBs are then connected to every possible PTB in the programmable core to provide sequential circuit capability. The size of the PTB remains the same; PTBs of size $(10,9,3)$ or $(10,18,3)$ were used.

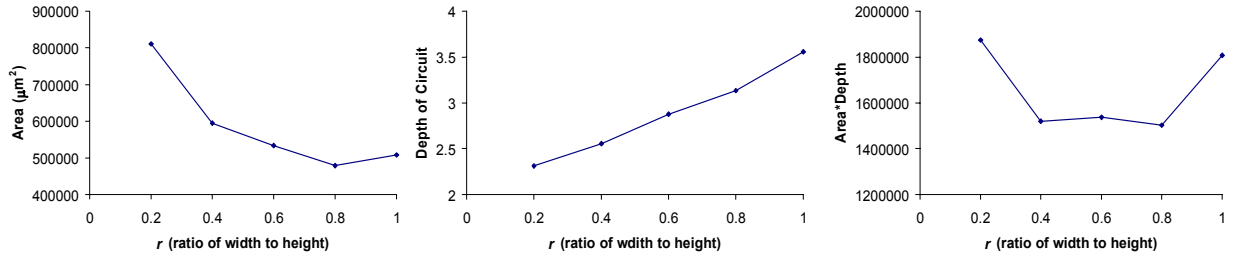


Figure 5.11: Value of PTB rectangular architecture parameter, r , for sequential fabrics.

Figure 5.11 shows the results of this experiment, in terms of area, depth, and area*depth. Results show similar trends to graphs shown in Section 5.3.1 in which r was optimized for combinational fabrics. Intuitively, we expect delay to increase as r increases since r is directly proportional to mapping depth of the fabric. Also, similar to Section 5.3.1, area decreases with increasing r because a deeper fabric allows for easier placement of the PTBs in the unidirectional architecture. Overall, the best area*depth value for r is between $r=0.4$ to 0.8 with a 1 to 2% relative difference between these two values.

As described in [82], it is important to analyze results for their sensitivity to experimental assumptions. As a byproduct of this experiment, we were able to observe how r behaves to benchmark circuit variations. Overall, we see the conclusion for optimal r in both cases is similar. According to the methodology in [82], we classify this experiment as *slightly sensitive* to benchmark circuits.

5.3.5 Value of α for Sequential Triangular Fabrics

In this experiment, we optimize α for sequential PTB architectures. Similar to the previous experiment, the baseline architecture consists of the Dual-Network architecture with PTB sizes of (10, 9, 3) or (10, 18, 3). The results of this experiment are shown in Figure 5.12.

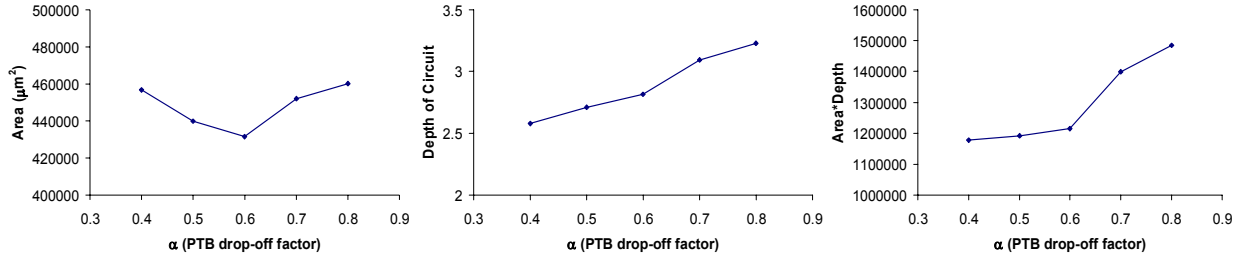


Figure 5.12: Value of PTB triangular architecture parameter, α , for sequential fabrics.

We observe similar behaviour to the α parameter optimization experiment for combinational circuits. In this experiment the best value for area is $\alpha=0.6$, but the relative difference between $\alpha=0.5$ and 0.6 is less than 2%. Again, we see two trends in the area graph. If α is too small, then the placement tool finds it more difficult to place circuits and compensates by choosing a larger fabric; and if α is too large then some PTBs are left unused. Worst case depth, and consequently delay, of the circuit increases with α since they are directly proportional to each other. Overall, $\alpha=0.4$ to 0.5 is a good choice, with a 1% relative difference in area*depth product. In summary, we see the conclusion for optimal α for both combinational and sequential circuits are the same. We would classify the sensitivity to benchmark circuits for α in triangular fabrics as *slightly sensitive* according to [82].

5.3.6 Comparison of Triangular and Rectangular Sequential Architectures

We see a triangular core with $\alpha=0.5$ has a 21.6% lower area*depth product compared to a rectangular core with $r=0.4$ for sequential circuits. Although the depth for triangular fabrics is larger (5.6% more on average) compared to rectangular fabrics, triangular cores are generally more area efficient (approximately 26% smaller). When comparing these findings with results from Section 5.3.3, we see many similarities. Not only is a triangular fabric most efficient for

both combinational and sequential circuits, it also has the same optimized α parameter value of 0.5. Also, the relative percentage difference between optimal and near optimal values in both experiments are very similar. Finally, we note the interconnect parameters, r and α , are only *slightly sensitive* to combinational and sequential benchmark circuit variations according to the methodology in [82].

5.4 Flip-flop Placement Parameter Optimization

As described in Chapter 3, the interconnect for sequential PTB architectures consists of two routing networks, one to drive combinational signals and the other for sequential signals. Both routing networks are *fully connected*. The combinational network can connect to all PTB inputs in subsequent logic levels of the core; and the sequential network can drive any PTB input in the fabric. Because of the fully connected routing networks, the size of the fabric can increase significantly. This additional area can be reduced by removing some flip-flops and placing remaining ones strategically. In turn, this decreases the number and size of the routing multiplexers that make up the sequential network. In the following subsections, we perform experiments to find the optimal flip-flop arrangement for sequential architectures. In particular, we focus on optimizing the parameters v and d for the Dual-Network and Decoupled sequential PTB architectures, respectively. For both experiments, we used a triangular PTB fabric with $\alpha=0.5$ and PTB sized (10, 9,) or (10, 18, 3) as the baseline architecture.

5.4.1 Value of v for Dual-Network Architecture

Figure 5.13 (a) shows the base interconnect structure for the Dual-Network architecture. By examining benchmark circuits mapped on an architecture with registers in every PTB, we observed that the utilization of registers in lower levels (near the left side of Figure 5.13 (a)) was

significantly lower than the utilization of registers in higher levels (near the right side of Figure 5.13 (a)). Intuitively, this makes sense; combinational values may need to be computed using several levels of PTBs before they are registered. Thus, rather than depopulating the flip-flops uniformly, we assume that the number of PTBs with registered outputs in level i is equal to $r_i = n_i * v^{(l-i+1)}$ where n_i is the number of PTBs in level i , l is the number of levels, and v is a constant that we optimize in this section. Figure 5.13 (b) shows an example interconnect structure after removing some flip-flops. The figure illustrates six registered outputs are remaining (one output in the first column, two in the second, and three in the last column).

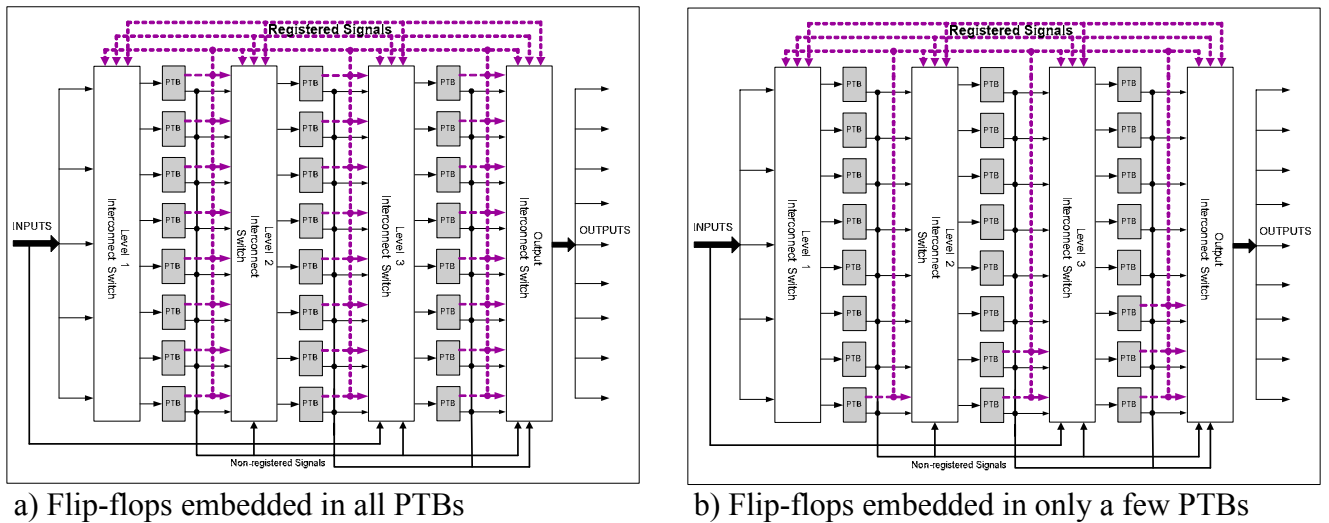


Figure 5.13: Dual-Network architecture parameter, v .

Intuitively, as v increases, the number of flip-flops and the size of the non-directional network increases, so we would expect the area to rise. On the other hand, if v is too small, the placement tool will find it more difficult to find a legal mapping for circuits with many sequential elements, and thus will compensate by choosing a larger core. Thus, we would expect these two competing trends to cause a minimum. The depth drops slightly as v increases; this is because the placement tool finds it easier to find legal placements, meaning, in some cases, it can get by

with a smaller core. The product of area*depth as a function of ν is shown in Figure 5.14.

Overall, the area*depth shows a minimum at $\nu=0.8$.

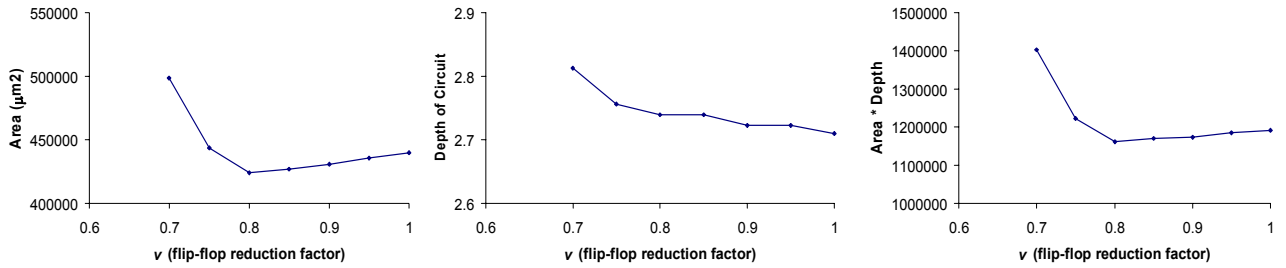
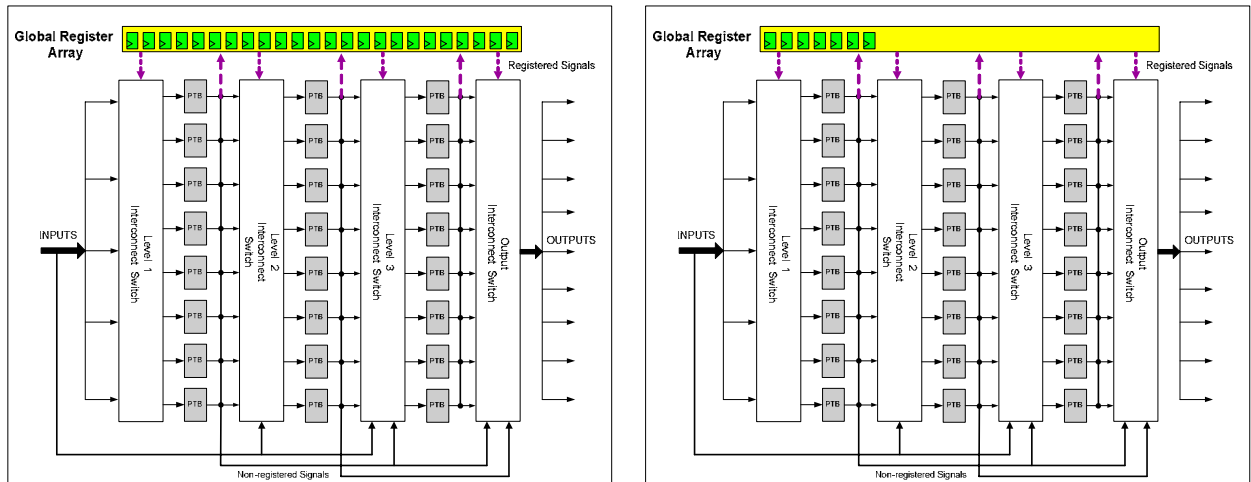


Figure 5.14: Value of Dual-Network architecture parameter, ν .

5.4.2 Value of d for Decoupled Architecture

In this experiment, we find the optimal ratio of flip-flops to total number of PTBs for the Decoupled architecture. We will use the parameter, d , to denote the number of global registers as a fraction of the total number of PTB output signals. Thus, if there are n PTBs, each with o outputs, there are $r = n*(d*o)$ flip-flops in the register file. Figure 5.15 illustrates this graphically. As d decreases, the number of flip-flops present in the global register array decreases, and consequently the size and number of routing multiplexers that drive signals to and from the global register array shrinks.



a) Equal number of flip-flops to PTBs in global register array

b) Reduced number of flip-flops to PTBs in global register array

Figure 5.15: Decoupled architecture parameter, d .

Intuitively, if d is too low, the placement tool will not find enough flip-flops to implement the benchmark circuits, and will increase the size of the fabric to compensate. On the other hand, if d is too large, some flip-flops will remain unused. The depth of the fabric stays constant for $d > 0.5$ because the additional flip-flops can not be taken advantage of by the placement tool. As shown in Figure 5.16, a compromise of $d = 0.67$ provides the best area*depth tradeoff.

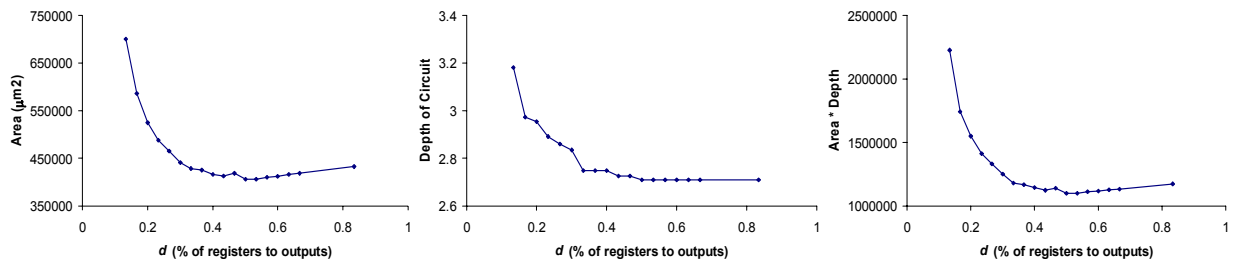


Figure 5.16: Value of Decoupled architecture parameter, d .

5.4.3 Comparison of Dual-Network and Decoupled Architecture

Comparing the minimums of the two curves in Figures 5.14 and 5.16, it is clear that a well-optimized Decoupled architecture is more efficient than a well-optimized Dual-Network

architecture. The area*depth product of the best Decoupled architecture is roughly 8% lower than that of the best Dual-Network architecture.

5.5 Limitation of Experimental Results

Although we have tried to present work that is complete, we acknowledge that there are limitations that need to be considered. The main limitation is the experimental parameter sweep while holding certain baseline architectural assumptions. As mentioned in [82], assumptions made to perform experiments can greatly influence conclusions. As a result, these conclusions may not be entirely accurate. In our work, we kept the number of architectural assumptions to a minimum. For example, optimal values of PTB logic parameters were later used in subsequent experiments. Also, the experiments to originally find the optimal logic block values used optimal or near-optimal architectural assumptions of the remaining parameters. Nevertheless, more accurate values can be obtained through repeated iteration of the derived optimal values. This method, however, is extremely time-consuming. We believe the results obtained in this chapter are already optimal or close to optimal that repeated experimental iterations are not necessary.

5.6 Summary

In this chapter we have presented our experimental results. We have determined the best size of a PTB contains 10 inputs, 9 or 18 product-terms, and 3 outputs. We have also compared the efficiency of rectangular and triangular PTB based architectures. We have found triangular PTB architectures to be generally more efficient (approximately 26% better in area*delay product) than rectangular cores. We have also determined optimal parameter values for the Dual-Network and Decoupled sequential PTB architectures. Finally, we compared the Dual-Network

and Decoupled architectures and found the Decoupled architecture has a slight area*delay advantage (about 8%) over the Dual-Network architecture. Table 5.1 shows a summary of the optimal parameter values we obtained.

Parameter	Symbol	Value
Inputs per PTB	i	10
Product-Terms per PTB	p	9 or 18
Outputs per PTB	o	3
Rectangular Core: Ratio of levels to number of PTBs in first level	r	0.4
Triangular Core: Ratio of PTBs in neighbouring levels	α	0.5
Sequential Fabric, Dual-Network Architecture: Number of registered PTBs in each core level	v	0.8
Sequential Fabric, Decoupled Architecture: Ratio of global registers to number of PTBs	d	0.67

Table 5.1: Summary of optimized low-level parameter values.

Chapter 6

COMPARISON AND IMPLEMENTATION

In the previous chapter we determined optimal low-level parameters for PTB-based architectures. In this chapter we compare the PTB-based architecture with the Gradual lookup-table based architecture from [14]. We then describe a proof-of-concept chip employing the PTB-based architecture.

6.1 Comparison to LUT-Based Architecture

In this section, we compare the area and delay efficiency of our synthesizable core with that of the LUT-based architecture in [14]. Based on the results from Chapter 5, we used a combinational triangular core with $\alpha=0.5$ and PTBs with $i=10$ inputs, $p=9$ or 18 product terms, and $o=3$ outputs.

The results are shown in Table 6.1. Overall, the PTB-based architecture is 35% smaller and 72% faster than the LUT-based architecture. In the architecture of [14], most of the area and delay was due to large routing multiplexers. [83] determined that about 53% of total area was due to routing multiplexers and the configuration bits that control these multiplexers. In our architecture, since we have larger, and hence fewer, product-term blocks, the routing fabric is simpler and faster. Analysis of our experimental results showed that interconnect accounts for 24% of the total area.

Further details of the PTB implementation of the benchmark circuits are shown in Table 6.2. The high PTB utilization in Table 6.2 may not entirely reflect logic block efficiency. Recall a PTB contains many product-terms (and also OR gates) and it is up to the technology mapping algorithm to be as efficient as possible in logic packing. Often full utilization of the product-terms (and OR gates) in the PTBs is difficult to achieve. Because of this, certain benchmark circuits may not perform very well when mapped to PTBs.

Circuit	Area (μm^2)			Delay (ns)		
	LUT fabric	PTB fabric	Improvement	LUT fabric	PTB fabric	Improvement
cm138a	79 450	68 009	14.4 %	20.9	7.1	66.1 %
cm42a	79 670	64 228	19.4 %	20.9	4.9	76.5 %
cm163a	80 365	80 926	-0.7 %	20.9	10.8	48.4 %
cm85a	109 995	72 071	34.5 %	26.3	7.3	72.4 %
rd53	107 263	85 012	20.7 %	26.3	6.4	75.6 %
cm150a	216 144	134 449	37.8 %	34.4	10.3	70.0 %
mux	302 770	134 449	55.6 %	41.5	10.3	75.1 %
cmb	163 693	80 938	50.6 %	32.8	10.9	66.7 %
x2	114 248	70 815	38.0 %	26.3	7.7	70.6 %
cm162a	80 023	78 730	1.6 %	20.9	9.3	55.8 %
pm1	122 989	101 335	17.6 %	26.3	11.1	57.6 %
decod	85 597	103 051	-20.4 %	20.9	6.7	68.2 %
cc	537 513	187 250	65.2 %	46.3	10.4	77.5 %
i1	179 248	116 297	35.1 %	32.8	10.6	67.5 %
misex1	163 588	69 920	57.3 %	32.8	6.3	80.7 %
pcl	120 500	180 362	-49.7 %	26.3	11.9	54.8 %
sct	180 436	206 614	-14.5 %	32.8	12.2	62.9 %
cu	228 576	100 168	56.2 %	34.4	9.8	71.6 %
sqrt8	300 204	105 303	64.9 %	41.5	9.0	78.4 %
sqrt8ml	377 658	164 543	56.4 %	45.7	9.0	80.4 %
lal	340 068	340 425	-0.1 %	41.5	16.3	60.6 %
squar5	218 648	66 302	69.7 %	34.4	5.6	83.9 %
ldd	331 367	147 931	55.4 %	41.5	9.2	77.9 %
pcler8	336 547	346 406	-2.9 %	41.5	18.8	54.7 %
c8	1 124 829	346 504	69.2 %	66.5	18.1	72.8 %
count	421 721	498 829	-18.3 %	45.7	17.8	61.1 %
comp	496 800	353 452	28.9 %	46.3	18.1	60.9 %
b9	817 867	375 121	54.1 %	62.6	16.7	73.3 %

unreg	339 868	160 713	52.7 %	41.5	10.5	74.8 %
frg1	1 062 808	334 953	68.5 %	66.5	17.7	73.4 %
misex2	420 493	215 676	48.7 %	45.7	13.5	70.4 %
f51m	749 837	201 085	73.2 %	62.6	10.1	83.8 %
cht	3 139 111	601 530	80.8 %	112.2	17.7	84.2 %
b12	902 504	143 963	84.0 %	67.7	10.9	84.0 %
5xpl	756 187	166 027	78.0 %	62.6	9.4	85.0 %
inc	388 765	130 689	66.4 %	45.7	8.3	82.0 %
vg2	507 168	707 288	-39.5 %	46.3	21.6	53.3 %
ttt2	1 300 072	354 500	72.7 %	77.4	13.1	83.0 %
rd73	732 367	459 375	37.3 %	62.6	16.4	73.7 %
term1	2 210 396	938 399	57.5 %	100.4	18.5	81.6 %
9symml	3 266 584	304 526	90.7 %	117.3	10.6	91.0 %
apex7	3 460 155	977 454	71.8 %	117.3	20.7	82.4 %
bw	1 556 027	470 369	69.8 %	85.3	14.4	83.1 %
clip	2 662 875	464 458	82.6 %	107.1	16.9	84.2 %
rd84	1 923 606	1 013 640	47.3 %	94.2	21.0	77.8 %
alu2	2 394 192	1 149 793	52.0 %	102.0	23.0	77.4 %
Average	771 539	292 910		52.3	12.5	
Geomean	411 950	203 951	35 %	45.8	11.6	72 %

Table 6.1: Comparison of Product-term architecture to LUT-based architecture.

Circuit	Product-Term Based Architecture					LUT Based Architecture	
	# of Primary Inputs	# of Primary Outputs	# of PTBs Available	PTB utilization	Ratio of Routing to Logic Area	# of 3-LUTs Available	LUT utilization
cm138a	6	8	4	0.75	0.08	25	0.68
cm42a	4	10	4	1.00	0.02	25	0.80
cm163a	16	5	4	1.00	0.24	25	0.64
cm85a	11	3	4	1.00	0.16	36	0.50
rd53	5	3	5	1.00	0.13	36	0.50
cm150a	21	1	6	0.83	0.34	64	0.53
mux	21	1	6	0.83	0.34	81	0.38
cmb	16	4	4	1.00	0.24	49	0.55
x2	10	7	4	0.75	0.12	36	0.69
cm162a	14	5	4	1.00	0.22	25	0.72
pm1	16	13	5	1.00	0.23	36	0.64
decod	5	16	6	1.00	0.09	25	0.80
cc	21	20	8	0.88	0.32	121	0.31
i1	25	13	5	1.00	0.31	49	0.49
misex1	8	7	4	0.75	0.11	49	0.65
pcl	19	9	8	1.00	0.32	36	0.69

sct	19	15	9	1.00	0.32	49	0.63
cu	14	11	5	1.00	0.22	64	0.48
sqrt8	8	4	6	1.00	0.16	81	0.54
sqrt8ml	8	4	9	0.89	0.22	100	0.38
lal	26	19	13	0.92	0.41	81	0.59
squar5	5	8	4	1.00	0.06	64	0.66
ldd	9	19	8	0.88	0.16	81	0.59
pcler8	27	17	13	0.85	0.42	81	0.52
c8	28	18	13	0.92	0.43	225	0.27
count	35	16	17	0.82	0.48	100	0.59
comp	32	3	13	1.00	0.45	121	0.52
b9	41	21	13	1.00	0.46	169	0.37
unreg	36	16	6	1.00	0.40	81	0.62
frg1	28	3	13	0.92	0.44	225	0.31
misex2	25	18	9	1.00	0.36	100	0.60
f51m	8	8	6	1.00	0.08	169	0.44
cht	47	36	13	0.92	0.32	529	0.20
b12	15	9	4	1.00	0.13	196	0.39
5xp1	7	10	5	1.00	0.06	169	0.48
inc	7	9	4	0.75	0.04	100	0.84
vg2	25	8	17	0.88	0.28	121	0.64
ttt2	24	21	9	1.00	0.20	256	0.42
rd73	7	3	13	0.77	0.16	169	0.66
term1	34	10	21	0.86	0.33	400	0.32
9symml	9	1	9	0.78	0.13	576	0.24
apex7	49	37	26	1.00	0.36	576	0.22
bw	5	28	13	0.77	0.13	289	0.52
clip	9	5	13	0.92	0.17	484	0.36
rd84	8	4	26	0.92	0.26	361	0.54
alu2	10	6	29	0.93	0.28	441	0.57
Average	17.89	11.13	9.57			156.00	
Geomean	14.31	8.07	7.98	0.92	0.24	100.81	0.57

Table 6.2: Synthesizable programmable architecture implementation details.

Figure 6.1 shows the area and delay plots of the PTB-based and LUT-based architecture over the benchmark circuits. Figure 6.2 shows area and delay as a function of the number of logic blocks (LUTs or PTBs). The results show that the area of the LUT-based architecture increases faster than the area of the PTB-based architecture. This is because there are more logic blocks in the LUT architecture than PTBs in the product-term architecture for a given circuit. Fewer logic blocks means fewer routing multiplexers are required, and thus, area increases more slowly. We

see a similar trend with delay; because the product-term architecture uses larger blocks, fewer logic levels are required compared to the LUT-based architecture.

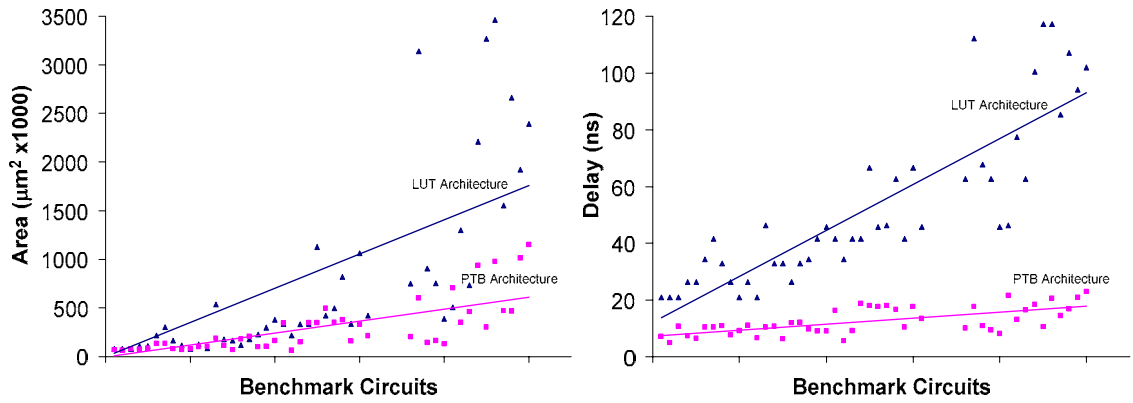


Figure 6.1: Area and Delay plots for the PTB and LUT-based architecture.

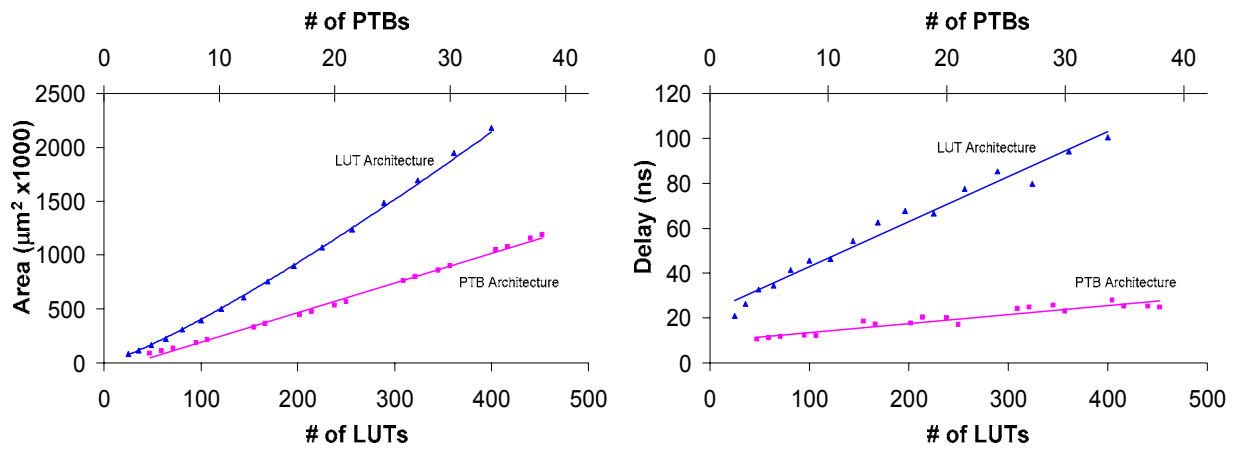


Figure 6.2: Growth trends of the PTB and LUT-based architecture.

From the results in Table 6.1, we also observe that larger circuits tend to result in higher area improvements. Again, this is because the LUT-based architecture in [14] contains large routing multiplexers; the sizes of these multiplexers grow as the core increases. Although our multiplexers also grow as the core size increases, our multiplexers form a smaller portion of the overall fabric than the corresponding multiplexers in the LUT-based architecture.

6.2 Proof-of-Concept Implementation

As a proof-of-concept, we laid out a programmable logic core using the Dual-Network architecture with $v=1.0$, and used it to implement a parallel network interface module from [84] as our target application, as described in [83, 85]. Unlike [85], in which only the combinational next-state logic was implemented, in our architecture, we implemented the entire state machine, including the flip-flops. The size of the combinational next-stage logic design contains about 22 4-LUT's and the entire state machine consists of roughly 35 4-LUT's.

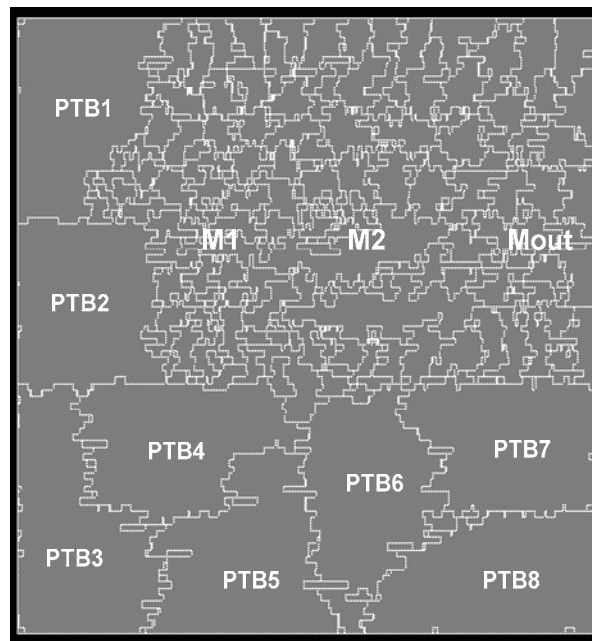
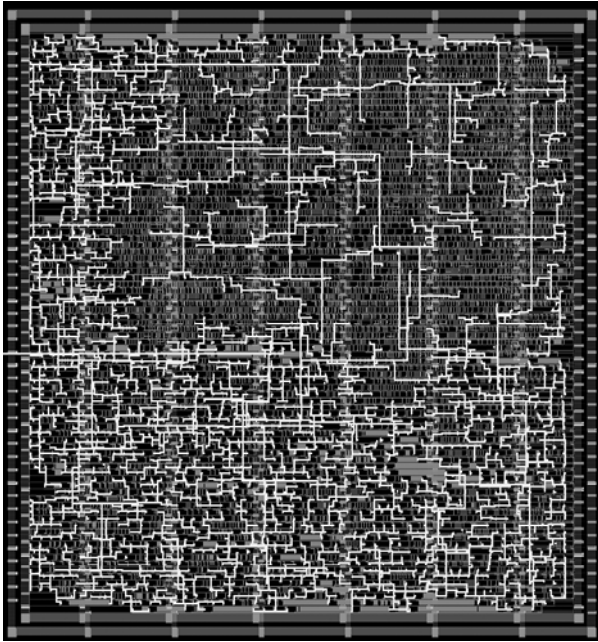
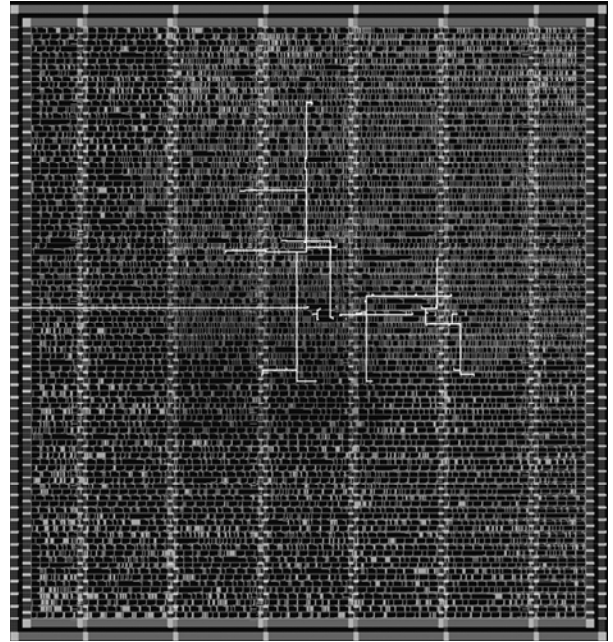


Figure 6.3: Floorplan of proof-of-concept chip.

Figure 6.3 shows the floorplan of our chip. The programmable core is a triangular PTB-based architecture with $\alpha=0.5$. The placement of the logic and interconnect blocks closely match the conceptual view in Figure 3.7. The PTBs labelled “PTB1 to PTB5” and “PTB6 to PTB8” are logic blocks belonging to the first and second level, respectively; and the multiplexers labelled “M1, M2, and Mout”, are interconnect blocks in the first, second, and last levels, respectively.



a) Configuration clock network



b) sequential support clock network

Figure 6.4: Clock network of proof-of-concept chip.

The majority of the area in the fabric is due to configuration bits. These bits store the state of the individual multiplexers and the logic contents of the PTBs. The configuration bits are built using flip-flops, each connected to a common configuration clock. Figure 6.4(a) shows the configuration clock tree for our synthesizable core. The clock for the registered signals, shown in Figure 6.4(b), is considerably less dense and localized in the center of the chip. One may expect these registers (that implement sequential logic) to be located close to the logic blocks. The physical placement tool, however, has placed them near the interconnect blocks. This is because the registered outputs are required to connect to many routing multiplexers. 12% of the total chip area was required to successfully route all the signals. This low percentage is likely due to improvements in physical design tools and the relatively small size of the fabric.

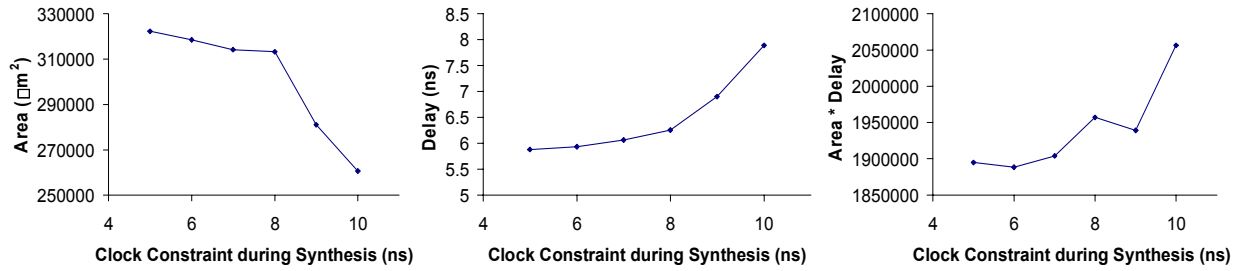


Figure 6.5: Synthesis results of proof-of-concept chip.

The speed and area required by our core depends on the timing constraints supplied to the synthesis tool. Figure 6.5 shows area, delay, and area*delay product graphs for several different timing constraints. Area decreases at a faster rate when the clock constraint is relaxed to 8ns or more because the synthesis tool does not need to add large buffers to meet the time constraint. The implementation with the lowest area*delay product had a delay of 6 ns, and an area of 348,000 μm^2 in a 0.18 μm TSMC process. Table 6.3 compares these measurements with those obtained from the lookup-table based fabric in [85]. The fabric in [85] does not include flip-flops, and thus, can only be used to implement the combinational parts of the state machine. As the table shows, our architecture is 12% smaller and 40% faster than the lookup-table based core. The difference is primarily due to the use of a product-term based architecture rather than a lookup-table based architecture.

	Product-term fabric Supporting whole circuit	Product-term fabric Supporting combinational portion	LUT-based fabric [85] Supporting combinational portion
Area (μm^2)	238,000	348,000	396,000
Delay (ns)	5.5	6.0	10

Table 6.3: Area and Delay Summary.

6.3 Summary

In this chapter, we have compared the PTB-based architecture against a LUT-based device in [14]. Overall, we found that the PTB-based architecture is 35% smaller and 72% faster; these improvements are primarily due to a dramatic reduction in the amount of circuitry needed to route signals.

A proof-of-concept implementation using the network interface module from [84, 85] was also presented. This device was designed on a 0.18 μm TSMC process and shown that it is 12% smaller and 40% faster than a lookup-table based core of similar capacity.

7.1 Summary

In this thesis, we have presented product-term based synthesizable programmable logic architectures. In order to help alleviate the increasing complexity of SoCs, mechanisms for post-fabrication flexibility such as embedded programmable logic cores are likely to become important. Currently, programmable logic cores are available in the form of custom hand-made layouts (“hard” cores) from third party vendors. Because of difficulties in integration, verification, and customization for hard embedded programmable cores, an alternative technique in implementing programmable fabrics has been described. In this approach, core vendors supply a synthesizable version (“soft” core) of the programmable fabric. A soft core is one in which the user obtains an HDL description of the behaviour of the core. The HDL description is then fed through standard synthesis tools to translate the RTL description of the programmable core to a gate-level description of the programmable core. Synthesizable programmable logic cores can be treated like regular logic during the design process, leading to simpler integration and more flexibility. When implementing small amounts of programmable logic, this may outweigh the inefficiencies in area, delay, and power that are inherent in soft cores.

This thesis began with a description of a family of synthesizable programmable logic cores that used product-term array blocks as the basic logic element for programmability. We presented two interconnect structures for the product-term blocks. The first structure imposed a rectangular arrangement of the logic blocks and the second structure imposed a triangular connection of the logic blocks. We then described a method to provide sequential logic support for synthesizable fabrics. Our solution was to provide both registered and unregistered outputs at the logic blocks. The two outputs fed two separate routing networks to avoid creating combinational loops in the unprogrammable fabric; combinational loops increase the difficulty seen by the synthesis tool. We presented two architectures employing our enhancement, which we refer to as the Dual-Network Architecture and the Decoupled Architecture.

Placement and routing tools were created to support our novel product-term based architecture. For placement, a greedy-based algorithm employing slack analysis was developed. Experimentally, we found the greedy-based algorithm to be very close or exactly the same as the optimal results obtained by hand. The development of the routing algorithm was simple because of the fully connected interconnect fabric. The main focus of the router was to assign nets to multiplexers without causing contention. The basic routing algorithm involves enumerating all the signals and assigning connections based on the enumerations.

We optimized a number of architectural parameters in order to obtain the most area and delay efficient architecture. The architectural parameters were separated into two categories, high-level and low-level parameters. High-level parameters are values specified by the VLSI designer who wishes to use our cores. In contrast, low-level parameters are not normally specified by the

VLSI designer. These parameters describe specific characteristics of the fabric and are often optimized through architectural experimentation. We proceeded to optimize all low-level parameters that were used to describe the product-term based architectures. We found the best size of a product-term block contains 10 inputs, 9 or 18 product-terms, and 3 outputs. We further found the triangular architecture to be 26% more efficient in area*delay product than the rectangular architecture. The triangular architecture was more efficient because its structure more closely matches that of user circuits. Finally, we found the Decoupled Architecture to be 8% more area*delay efficient than the Dual-Network Architecture.

We concluded with a description of a proof-of-concept chip implementation of a parallel network interface module. The chip was implemented in a 0.18 μ m TSMC process and shown that it is 12% smaller and 40% faster than a lookup-table based core of similar capacity. The standard-cell placement of the fabric closely matches the conceptual view. Also, 12% of the total chip area was required to successfully route all the signals in the fabric. Comparing the product-term based architecture to the lookup-table based device in [14], we found that our new architecture is 35% smaller and 72% faster on average MCNC benchmark circuits. This is primarily due to a dramatic reduction in the amount of circuitry needed to route signals.

7.2 Future Work

Although significant strides have been made to reduce area and delay of synthesizable architectures, better results could be obtained by “tweaking” the standard-cell library to include cells specifically optimized to implement our programmable logic fabric. We have not considered this in this work, since our goal was to create architectures that can be implemented using the standard synthesis tools, cell libraries, and design flows that integrated circuit

designers are already familiar with. Nonetheless, if this design technique was to become mainstream, specially-designed standard cells could be created. It is likely that only a small number of additional types of standard cells would be required. The product-term block architecture is primarily composed of multiplexers and flip-flops for configuration. If large multiplexers (perhaps 32:1 or 64:1 multiplexers) were present and custom-made logic blocks built from SRAM bits were employed, then considerable area and delay reductions can be achieved.

Further reductions in area and delay can also be achieved by limiting the amount of interconnect flexibility. The product-term based architecture currently employs a fully-connected interconnect fabric. This, however, requires more intelligent placement and routing tools. A hybrid architecture employing both product-terms and lookup-tables may also result a denser and faster synthesizable programmable logic core. Hybrid architectures have been studied in [17], but only for hard prefabricated programmable devices.

Another area we have not explored is expanding the application space for synthesizable architectures. Currently, the envisioned application for synthesizable architectures is the implementation of only small logic circuits, such as state machines. A heterogeneous or more coarse-grained architecture should be adopted to help expand the scope of synthesizable programmable logic cores. Similar to stand-alone FPGAs, integrating embedded blocks, such as CPUs, memories, and arithmetic blocks, will allow synthesizable architectures to play a larger role in the overall design of the SoC.

Lastly, power reduction techniques is another area worthy of exploration. Power can be optimized in a number of ways. Since the soft core approach is aimed at fine-grained programmability, we expect power optimization at the circuit and CAD level would result in the greatest reductions. For example, incorporation of low V_T and V_{DD} standard cells and power-aware CAD tools for synthesis, placement, and routing will help reduce power. A more simple approach to reduce power is through area reduction. Reducing area will reduce the number of transistors, thereby decreasing total power dissipation. A project is currently ongoing at the University of British Columbia to further investigate these issues.

7.3 Contributions of This Work

The contributions of this work are summarized as follows:

1. We presented novel product-term based synthesizable architectures. The architectures are based on a collection of product-term array blocks and arranged in two different fashions; namely a Rectangular and Triangular interconnect structure.
2. We enhanced existing synthesizable architectures to support sequential logic implementation. We introduced two novel architectures, the Dual-Network Architecture and the Decoupled Architecture.
3. We developed new CAD place and route algorithms to support the novel product-term based synthesizable architectures.
4. We identified and optimized architectural parameters in the product-term based synthesizable architecture to obtain the most area and delay efficient implementation.

5. We compared the product-term based synthesizable architecture with the lookup-table based architecture from [14] and found area and delay improvements of 35% and 72%, respectively.

6. We created a proof-of-concept chip implementation employing the product-term based architecture.

REFERENCES

- [1] "IC makers brace for \$3 million mask at 65 nm", EETimes article, September 2003. <http://www.eetimes.com/showArticle.jhtml?articleID=18309458>
- [2] "Chip industry growing, but experts wonder for how long", EETimes article, December 2003. <http://www.eetimes.com/showArticle.jhtml?articleID=18310473>
- [3] J. Greenbaum, "Reconfigurable Logic in SoC Systems", Proceedings of the 2002 Custom Integrated Circuits Conference, pp. 5-8, May 2002.
- [4] S.J.E. Wilton, R. Saleh, "Programmable Logic IP Cores in SoC Design: Opportunities and Challenges", Proceedings of the 2001 Custom Integrated Circuits Conference, pp. 63-66, May 2001.
- [5] Actel Corp, "VariCore Embedded Programmable Gate Array Core (EPGA) 0.18 μ m Family", Datasheet, December 2001.
- [6] Leopard Logic Inc, "HyperBlox FP Embedded FPGA Cores", Product Brief, 2002.
- [7] M2000, Inc, "M2000 FLEXEOStm Configurable IP Core", <http://www.m2000.fr>.
- [8] eASIC, "eASIC 0.13 μ m Core", <http://www.easic.com>
- [9] M. Borgatti, F. Lertora, B. Foret, L. Cali, "A Reconfigurable System featuring Dynamically Extensible Embedded Microprocessor, FPGA, and Customisable I/O", IEEE Journal of Solid-State Circuits, vol. 38, no. 3, pp. 521-529, March 2003.
- [10] T. Vaida, "PLC Advanced Technology Demonstrator TestChipB", Proceedings of the 2001 Custom Integrated Circuits Conference, pp. 67-70, May 2001.
- [11] P.S. Zuchowski, C.B. Reynolds, R.J. Grupp, S.G. Davis, B. Cremen, B. Troxel, "A Hybrid ASIC and FPGA Architecture", in the International Conference on Computer-Aided Design (ICCAD), pp. 187-194, November 2002.
- [12] T. Vaida, "Reprogrammable Processing Capabilities of Embedded FPGA Blocks", in the IEEE International ASIC/SOC Conference, pp. 180-184, September 2001.
- [13] F. Lien, J. Feng, E. Huang, C. Sun, T. Liu, N. Liao, D. Hightower, "A Hardware / Software Solution for Embeddable FPGA", in the Proceedings of the 2001 Custom Integrated Circuits Conference, pp. 71-74, May 2001.
- [14] N. Kafafi, K. Bozman, S.J.E. Wilton, "Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores", Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, pp. 1-9, February 2003.
- [15] S.J.E. Wilton, K. Bozman, N. Kafafi, J. Wu, "Method For Constructing an Integrated Circuit Device Having Fixed And Programmable Logic Portions And Programmable Logic Architecture For Use Therewith", U.S. Patent, submitted August 2003.

- [16] J.L. Kouloheris, A.E. Gamal, "FPGA Performance vs. Cell Granularity", Proceedings of the 1991 Custom Integrated Circuits Conference, pp. 6.2.1-6.2.4, May 1991.
- [17] A. Kaviani, S. Brown, "The Hybrid Field Programmable Architecture", IEEE Design and Test, Vol. 16, No. 2, pp. 74-83, April-June 1999.
- [18] A. Yan, S.J.E. Wilton, "Product Term Embedded Synthesizable Logic Cores", in the IEEE International Conference on Field-Programmable Technology, Tokyo, Japan, pp. 162-169, December 2003.
- [19] A. Yan, S.J.E. Wilton, "Sequential Synthesizable Embedded Programmable Logic Cores for System-on-Chip", in the Proceedings of the 2004 Custom Integrated Circuits Conference, pp. 435-438, October 2004.
- [20] M. Birnbaum, H. Sachs, "How VSIA Answers the SoC Dilemma", in the IEEE Computer Magazine, Vol. 32, No. 6, pp. 42-50, June 1999.
- [21] E.J. Marinissen, Y. Zorian, "Challenges in Testing Core-Based System ICs", in the IEEE Communications Magazine, Vol. 37, No. 6, pp. 104-109, June 1999.
- [22] V. Betz, "Architecture and CAD for the Speed and Area Optimization of FPGAs", Ph.D. Dissertation, University of Toronto, 1998.
- [23] V. Betz, J. Rose, and A. Marquardt, "Architecture and CAD for Deep-Submicron FPGAs", Kluwer Academic Publishers, 1999.
- [24] Xilinx, "Virtex FPGA Family", <http://www.xilinx.com>.
- [25] Altera, "Stratix FPGA Family", <http://www.altera.com>.
- [26] E. Ahmed and J. Rose, "The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density," in ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 3-12, February 2000.
- [27] J. Rose, R.J. Francis, D. Lewis, and P. Chow, "Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency" IEEE Journal of Solid-State Circuits, Vol. 25, pp. 1217-1225, October 1990.
- [28] G. Lemieux, D. Lewis, "Using Sparse Crossbars within LUT Clusters", in the ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, pp. 59-68, February 2001.
- [29] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault, D. Cashman, D. Galloway, M. Hutton, C. Lane, A. Lee, P. Leventis, S. Marquardt, C. McClintock, K. Padalia, B. Pedersen, G. Powell, B. Ratchev, S. Reddy, J. Schleicher, K. Stevens, R. Yuan, R. Cliff, J. Rose, "The Stratix-II Logic and Routing Architecture" to appear in the ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, February 2005.
- [30] M.I. Masud and S.J.E. Wilton, "A New Switch Block for Segmented FPGAs," International Workshop on Field Programmable Logic Applications, August 1999.

- [31] S.J.E. Wilton, "Architecture and Algorithms for Field- Programmable Gate Arrays with Embedded Memory", PhD Thesis, University of Toronto, 1997.
- [32] G. Lemieux and D.M. Lewis. "Analytical framework for switch block design", in the International Conference on Field-Programmable Logic, pp 122-131, 2002.
- [33] G. Lemieux, "Efficient Interconnection Network Components for Programmable Logic Devices", Ph.D. Dissertation, University of Toronto, 2003.
- [34] G. Lemieux, D. Lewis, "Design of Interconnection Networks for Programmable Logic", Kluwer Academic Publishers, 2003.
- [35] G. Lemieux, E. Lee, M. Tom, A. Yu, "Directional and Single-Driver Wires in FPGA Interconnect", in the International Conference on Field-Programmable Technology, Brisbane, Australia, December 2004.
- [36] P. Leventis, B. Vest, M. Hutton, D. Lewis, "MAX II: A Low-Cost, High-Performance LUT-Based CPLD", Proceedings of the 2004 Custom Integrated Circuits Conference, pp. 443-446, October 2004.
- [37] Xilinx Corp, "CoolRunner XPLA3 CPLD", Datasheet, <http://www.xilinx.com/>
- [38] Altera Corp, "MAX 7000", Datasheet, <http://www.altera.com/>
- [39] Z. Zilic, G. Lemieux, K. Loveless, S. Brown and Z. G. Vranesic, "Designing for High Speed-Performance in CPLDs and FPGAs", in the Proceeding of the Third Canadian Workshop on FPGAs, FPD 95, Montreal, Montreal, June 1995.
- [40] S. Brown and J. Rose, "FPGA and CPLD Architectures: A Tutorial," in IEEE Design and Test of Computers, Vol. 12, No. 2, pp. 42-57, June 1996.
- [41] G. Lemieux, D. Lewis, "Using Sparse Crossbars within LUT Clusters", in the ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, pp. 59-68, February 2001.
- [42] D. Hodges, H. Jackson, R. Saleh, "Analysis and Design of Digital Integrated Circuits", McGraw-Hill Series in Electrical and Computer Engineering, 2003.
- [43] J. Cong and Y. Ding, "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs", IEEE Trans. on Computer-Aided Design, vol. 13, no. 1, pp. 1-12, January 1994.
- [44] J. Cong, and Y. Hwang, "Simultaneous Depth and Area Minimization in LUT-Based FPGA Mapping", ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, pp. 68-74, February 1995
- [45] J. Cong, Y. Ding, "On Area/Depth Trade-off in LUT-Based FPGA Technology Mapping", IEEE Transactions on VLSI Systems, Vol. 2, No. 2, pp. 137-148, June 1994.

- [46] J. Cong, and Y. Hwang, "Structural Gate Decomposition for Depth-Optimal Technology in LUT-based FPGA Designs", *ACM Transactions on Design Automation of Electronic Systems*, Vol. 5, no. 3, July 2000.
- [47] J.H. Anderson, S. Brown, "Technology Mapping for Large Complex PLDs", in the *Proceedings of the 35th annual ACM/IEEE conference on Design automation Conference (DAC)*, pp. 698-703, June 1998.
- [48] K. Yan, "Practical Logic Synthesis for CPLDs and FPGAs with PLA-Style Logic Blocks", in the *Proceedings of the ASP-DAC 2001, Asia and South Pacific Design automation Conference*, pp. 231-234, February 2001.
- [49] S. Chen, T. Hwang, C.L. Liu, "A Technology Mapping Algorithm for CPLD Architectures", in the *IEEE International Conference on Field-Programmable Technology*, pp. 204-210, December 2002.
- [50] J. Kim, H. Kim, C. Lin, "A New Technology Mapping for CPLD under the time constraint", in the *Proceedings of the ASP-DAC 2001, Asia and South Pacific Design automation Conference*, pp. 235-238, February 2001.
- [51] D. Chen, J. Cong, M. Ercegovac, Z. Huang, "Performance-Driven Mapping for CPLD Architectures", in the *ACM International Symposium on Field-Programmable Gate Arrays*, pp. 39-47, February 2001.
- [52] B.K. Fawcett, "Map, Place and Route: The Key to High-Density PLD Implementation", *WESCON Conference, Microelectronics Communications Technology Producing Quality Products Mobile and Portable Power Emerging Technologies*, pp. 292-297, November 1995.
- [53] A. Dunlop and B. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits", *IEEE Transactions on Computer-Aided Design*, pp. 92-98, January 1985.
- [54] D. Huang and A. Kahng, "Partitioning-Based Standard-Cell Global Placement with an Exact Objective", *ACM Symposium on Physical Design*, pp. 18-25, 1997.
- [55] J. Rose, W. Snelgrove and Z. Vranesic, "ALTOR: An Automatic Standard Cell Layout Program", *Canadian Conference on VLSI*, pp. 169-173, 1985.
- [56] J. Kleinhans, G. Sigl, F. Johannes, and K. Antreich, "Gordian: VLSI Placement by Quadratic Programming and Slicing Optimization", *IEEE Transactions on Computer-Aided Design*, pp. 356-365, March 1991.
- [57] A. Srinivasan, K. Chaudhary, and E. Kuh, "Ritual: A Performance Driven Placement Algorithm for Small Cell ICs", *International Conference on Computer Aided Design*, pp. 48-51, November 1991.
- [58] B. Riess and G. Ettl, "Speed: Fast and Efficient Timing Driven Placement", *IEEE International Symposium on Circuits and Systems*, pp. 377-380, May 1995.

- [59] A. Marquardt, V. Betz, and J. Rose, "Timing-Driven Placement for FPGAs", ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, pp. 203-213, February 2000.
- [60] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by Simulated Annealing", Science, pp. 671-680, 1983.
- [61] C. Sechen and A. Sangiovanni-Vincentelli, "TimberWolf Placement and Routing Package", JSSC, pp. 510-522, 1985.
- [62] M. Huang, F. Romeo, and A. Sangiovanni-Vincentelli, "An Efficient General Cooling Schedule for Simulated Annealing", International Conference on Computer Aided Design, pp. 381-384, 1986.
- [63] J.S. Rose, "Parallel Global Routing for Standard Cells", IEEE Transactions on Computer Aided Design, Vol. 9, No. 10, pp.1085-1095, October 1990.
- [64] Y. Chang, S. Thakur, K. Zhu, and D. Wong, "A New Global Routing Algorithm for FPGAs", International Conference on Computer Aided Design, pp. 356-361, November 1994.
- [65] S. Brown, J. Rose, Z.G. Vranesic, "A Detailed Router for Field-Programmable Gate Arrays", IEEE Transactions on Computer Aided Design, pp. 620-628, May 1992.
- [66] G. Lemieux, S. Brown, "A Detailed Router for Allocating Wire Segments in FPGAs", ACM Physical Design Workshop, pp. 215-226, 1993.
- [67] G. Lemieux, S. Brown, D. Vranesic, "On Two-Step Routing for FPGAs", ACM Symposium on Physical Design, pp. 60-66, 1997.
- [68] M. Placzewski, "Plane Parallel A* Maze Router and Its Application to FPGAs", ACM Design Automation Conference, pp. 691-697, June 1990.
- [69] L. McMurchie, and C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, pp. 111-117, February 1995.
- [70] Y.-L. Wu, M. Marek-Sadowska, "An Efficient Router for 2-D Field-Programmable Gate Arrays", European Design Automation Conference, pp. 412-416, March 1994.
- [71] Y.-S. Lee, A. Wu, "A Performance and Routability Driven Router for FPGAs Considering Path Delays", ACM Design Automation Conference, pp. 557-561, June 1995.
- [72] E. Dijkstra, "A Note on Two Problems in Connexion with Graphs", Number Math., vol 1, pp. 269-271, 1959.
- [73] W. Dees and R. Smith, "Performance of Interconnection Rip-up and Reroute Strategies", in the ACM/IEEE conference on Design automation (DAC), pp. 382-390, June 1981.

- [74] F. Mo, R.K. Brayton, "River PLAs: A Regular Circuit Structure", in the ACM/IEEE conference on Design automation (DAC), pp. 201-206, June 2002.
- [75] F. Mo, R.K. Brayton, "Whirlpool PLAs: A Regular Logic Structure and Their Synthesis", in the International Conference on Computer-Aided Design (ICCAD), pp. 543-550, November 2002.
- [76] F. Mo, R.K. Brayton, "PLA-based regular structures and their synthesis", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 723-729, June 2003.
- [77] M. Holland, S. Hauck, "Automatic Creation of Reconfigurable PALs/PLAs for SoC", International Symposium on Field-Programmable Logic and Applications, pp. 536-545, 2004.
- [78] Y. Han, L. McMurchie, C. Sechen, "A High Performance CMOS Programmable Logic Core", in the Proceedings of the 2004 Custom Integrated Circuits Conference, pp. 439-442, October 2004.
- [79] S. Phillips, S. Hauck, "Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip", ACM/SIGDA Symposium on Field-Programmable Gate Arrays, February 2002.
- [80] N. Kafafi, "Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores", M.A.Sc. Thesis, University of British Columbia, 2003.
- [81] M. Hutton, J. Rose, J. Grossman, D. Corneil, "Characterization and Parameterized Generation of Synthetic Combinational Benchmark Circuits", in IEEE Transactions on CAD, Vol. 17, No. 10, pp. 985-996, October 1998.
- [82] A. Yan, R. Cheng, S.J.E. Wilton, "On the Sensitivity of FPGA Architectural Conclusions to the Experimental Assumptions, Tools, and Techniques", in the ACM International Symposium on Field-Programmable Gate Arrays, pp. 147-156, February 2002.
- [83] J.C.H. Wu, "Implementation Considerations for Soft Embedded Programmable Logic Cores", M.A.Sc. Thesis, University of British Columbia, 2004.
- [84] Mohsen Nahvi, Andre Ivanov, "A Packet Switching Communication-Based Test Access Mechanism for System Chips", in Proceedings of the IEEE European Test Workshop, pp. 81-86, 2001.
- [85] J.C.H. Wu, V. Aken'Ova, S.J.E. Wilton, R. Saleh, "SoC Implementation Issues for Synthesizable Embedded Programmable Logic Cores", in the Proceedings of the 2003 Custom Integrated Circuits Conference, pp 45-48, September 2003.