Bridging the Gap between Soft and Hard eFPGA Design

by

Victor Olubunmi Aken'Ova

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 Degree

in

The Faculty of Graduate Studies

Electrical and Computer Engineering

The University of British Columbia

March 2005

Abstract

Bridging the Gap between Hard and Soft eFPGA Design

by

Victor Olubunmi Aken'Ova

Potential cost savings that come from the ability to make *post* fabrication changes in System-on-Chip (SoC) designs make embeddable Field Programmable Gate Array (eFPGA) cores an attractive design option. However, they are only available as "hard" macros from vendors as a small number of fixed size cores, and may not be optimal in terms of area, power or delay for a given SoC. A "soft" eFPGA methodology [01][02] based on the ASIC design flow was used to create small amounts of programmable logic but incurs significant overhead. In this thesis, it is shown that this overhead can be reduced by deploying architecture-specific tactical standard cells in the ASIC flow, making eFPGA generation configurable, and imposing a regular structure on eFPGA architectures.

For the set of benchmarks considered, the use of tactical standard cells resulted in area and delay savings of 58% and 40% respectively, when compared to cores implemented with generic standard cells [02]. Also, a proposed IP-generator-based approach for eFPGA design is shown to achieve results that are competitive with commercial full-custom hard eFPGA cores. For example, for some large benchmark circuits (over 1000 4-LUTs) the generated eFPGA fabrics were up to 40% smaller than available hard eFPGA cores. Finally, it is shown that a regular structured architecture makes it possible to generate fabrics with logic capacities that greatly exceed what was previously possible [02] [15]. In addition, a structured layout approach yielded a 36% reduction (average) in wire lengths.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my supervisor Dr. Resve Saleh for the opportunity to work with him. His enthusiasm and support over the years are unmatched. He was easily my most vocal Cheerleader. I also would like to thank Dr. Guy Lemieux for teaching me everything I know about FPGAs.

I am also grateful to all the other professors of the SoC research Lab because I learned something from all of them. I especially would like to thank Dr. Steve Wilton who spearheaded the Soft eFPGA concept at UBC that ultimately led to my research topic. Thanks for all the comments and insight.

I would also like to thank the staff of the SoC lab. Roozbeh our CAD manager was a tremendous help and always tried his best to cater to my *many* requests. Roberto our Test Lab manager was always fantastic with the test lab equipment and great at finding solutions to all kinds of vexing problems. Sandy Scott our administrative assistant was always cheerful and happy to help. I am grateful to you all.

I would also like to thank past and present student members of the SoC lab for their support. I especially thank Pedram Sameni, James Wu, Ronald Fung, Louis Hong, Mohsen Nahvi, Marvin Tom, Zion Kwok, Neda Nouri, Noha Kafafi, Julien Lamoureux, Rod Foist, Brad Quinton, and Kara Poon.

Almost last, but certainly not least, I would like to thank my entire family for the tremendous love and support they have shown me my entire life. I am thankful to them for teaching me the value of hard work and dedication. A very gigantic hug also goes to my dear Laura for easily being my best friend.

Many thanks to NSERC, PMC-Sierra, the Canadian Microelectronics Corporation and the University of British Columbia for the financial and tool support that made all this work possible. Viva Canada.

In loving memory of my dearest Mama

Chapter 1

Thesis Introduction

1.1 Research Motivation

The increasing density of integrated circuit (IC) designs due to shrinking transistor sizes has led to the emergence of System-on-Chip (SoC) design to cope with the resulting increase in design size and complexity. However, this has resulted in significant increases in the cost of IC designs due to corresponding increases in engineering and mask costs in the order of tens of millions of dollars. Therefore, designers are pursuing software and hardware methods to build programmable SoCs and avoid the extra costs of chip re-spins. A programmable device can be used to compensate for errors, adapt to changes in standards or design specifications and amortize costs over design derivatives.

Embedded Programmable Logic Cores (ePLCs) have emerged as a natural hardware solution to meet this growing challenge because they allow logic functionality to be changed after fabrication. Such cores are generally suited to small and medium on-chip logic functions such as accelerator functions for on-chip processors to speed up embedded software, data encryption circuits in wireless devices that need to be changed from time to time, packet routing switches for the newly emerging Network-on-Chip design paradigm [71], and I/O standards for data communication.

In spite of the potential cost benefits and useful applications of ePLCs in SoC design, commercial success has been limited by a number of difficult issues that arise. These range from design and implementation issues [14] to issues related to the nature of commercial ePLCs devices in general. For example, common design problems concern the selection of blocks to make programmable, the

integration of fixed and programmable blocks and the size of the programmable block. However, by far the biggest issue with ePLCs is the high area, power and delay overhead that they generally incur.

These overhead issues are not unique to ePLCs alone; it has also become an important issue for stand-alone programmable logic devices like Field Programmable Gate Array (FPGA) chips. However, this issue is more critical for ePLCs because they are needed in high performance IC designs that have stringent area, speed, and power requirements. Further, this problem is made even worse because ePLCs, like FPGA chips, are available in limited sizes and ranges, and so designers must select the smallest core or chip that will accommodate their circuit(s). The selected ePLC could be much bigger than needed, slower, and consume more power, because vendors include excess resources for potential circuit implementations. As a result, prospective users are sometimes forced to abandon ePLCs altogether, and this has resulted in a general lack of interest in these devices.

Current commercial programmable logic design techniques do not afford vendors much opportunity to address some of the reasons for the overhead in ePLCs because they use expensive full-custom design techniques to build ePLCs, and so have to find a cost effective tradeoff when designing their products. Hence, for cost reasons, these vendors design a minimum set of cores that can serve the broadest range of user applications possible while keeping the device performance overhead within tolerable bounds. This model works well for standalone FPGA chips because they are not used in high performance situations and therefore users can tolerate the additional performance overhead.

The above model is unsatisfactory for ePLCs because higher performance is often required. Furthermore, because their application space is limited and target user circuits often share common characteristics, surplus resources on the same scale as FPGA chips is not required. For example, if a

designer uses an ePLC for the purpose of allowing future "bug fixes" or design revisions, then it is reasonable to assume that potential future circuit implementations will be of about the same size. The existing model for ePLC design cannot take advantage of domain information, since tailoring ePLCs for application specific scenarios is expensive for vendors. However, if ePLCs are to make inroads into mainstream digital design, domain knowledge should be exploited [01] in some fashion.

The potential benefits and the difficulties associated with embeddable programmable logic design motivate the research work presented in this thesis report. For most of the benefits to be realized, current ePLC design methodologies must be revisited. For example, rather than a "one-size-fits-all" model for designing programmable logic cores, an alternative approach that tailors cores to a particular application domain might be more appropriate. Therefore, in this research, we explore some new techniques that could make a "one-size-fits-few" design model feasible for ePLC design.

1.2 Research Objectives

Given the inherent difficulties associated with using ePLCs in high performance chip design, this work proposes some new ways to reverse the current trends and make the use of ePLCs more attractive to users. To achieve this goal, some important observations about current methodologies for embedded programmable logic design will be made and solutions investigated and evaluated.

First, the advantages and disadvantages of the "soft" [02] and "hard" [44] ePLC design methodologies are evaluated. Following from this, an embeddable programmable logic architecture is implemented in a way that combines the best characteristics of the existing design methodologies.

Second, inefficiencies in area and speed that result from implementing a ePLC within an automated, generic-cell-based design flow [02] are investigated, and the compared to previous research results. The goal here is to pinpoint areas of the design flow that could benefit the most from optimization.

Third, architecture-specific tactical cells are designed to replace generic cells and eliminate inefficiencies resulting from the use of a generic-cell-based IC design flow [02] [14] [15]. ollowing custom cell design, area and speed improvements that can be achieved as a result of implementing an embeddable programmable logic architecture with custom cells in an automated design flow are reported. The results obtained are then compared with other previous approaches [02][15][68].

Fourth, certain inefficiencies that exist in current commercial design approaches are investigated by comparing our design results for area and speed to an existing commercial hard ePLC library [44].

Lastly, a new paradigm is introduced for embedded programmable logic design that combines domain driven architectural exploration with a flexible and efficient semi-custom circuit design flow.

1.3 Thesis Organization

Chapter 2 of this thesis presents some general background on the research subject, summarizes work done in this area to date, describes benefits and difficulties associated with existing ePLC technologies and design methods, illustrates potential uses of programmable technology on a wireless IC (Bluetooth) and then introduces an approach for ePLC design explored in this thesis.

Chapter 3 describes variants of a well-known architecture to be used in the proposed approach for embedded PLC design. This chapter includes a detailed description of some design issues that were resolved during architecture specification in order to ensure proper ePLC operation.

Chapter 4 presents results obtained after implementing our ePLC in a "soft" design flow as described in [14]. In Chapter 5, an alternative approach similar to work presented in [01][68][69] which requires the design and implementation of architecture-specific tactical cells is explored. The details of design and implementation issues related to this work are presented. Next, our results are compared with earlier results obtained using architectures and approaches described in [14][15][68].

Chapter 6 compares the results obtained in Chapter 5 with estimates for commercial hard eFPGAs [41], to illustrate key problems related to current design approaches. In addition, two blocks in a Bluetooth baseband SoC are used to investigate the impact of eFPGA cores on area. Finally, a novel design paradigm, for automatic embedded programmable logic generation is presented.

Chapter 7 summarizes the work presented, suggests topics for future work and lists contributions.

Chapter 2

Background and Related Previous Research

This chapter begins with a general overview of logic design methodologies and then focuses on programmable logic design. In particular, two commercial technologies for embedded programmable logic design are described. The description focuses on important features of both approaches and highlights their advantages and disadvantages. Next, a wireless SoC platform is used to illustrate potential applications of programmable logic in IC design. Also, issues related to different methodologies for *re*programmable logic implementation are discussed. Finally, a description of the research problem and an outline of the focus of this thesis report are presented.

2.1 Overview of Integrated Circuit (IC) Design Techniques

Over the last three decades, several logic design methodologies have evolved to cope with technological advancements in semiconductor circuit design. As shown in Figure 2.1, these methodologies tend to fall into two main groups: full-custom and semi-custom IC design methods.

Full-custom design relies to a large extent on manual effort for most design decisions. For example, design decisions such as transistor sizing, transistor layout, device placement and routing are all carried out manually with the aid of rudimentary Computer-Aided Design (CAD) tools. This technique offers the greatest flexibility from a designer perspective, because circuits can be tailored to specifications with superior performance in terms of area, delay or power. However, there is a high engineering cost overhead involved. Furthermore, given shrinking time-to-market windows and

shelf-life of IC products, it becomes more difficult to depend on full-custom techniques for IC design. For example, a large IC design house like Intel® requires large teams of designers working for the equivalent of hundreds of man-years to deliver high-performance full-custom products such as the Pentium® chip on schedule. This results in a huge expense that has an impact on the pricing of such chips and the products that use them (e.g., personal computers). Similarly, well-known programmable logic device vendors like Xilinx® and Altera®, use full-custom techniques to design their Field Programmable Gate Array (FPGA) devices, and this is part of the reason some of these devices can cost anywhere from a few hundred dollars to a few thousand dollars per chip.

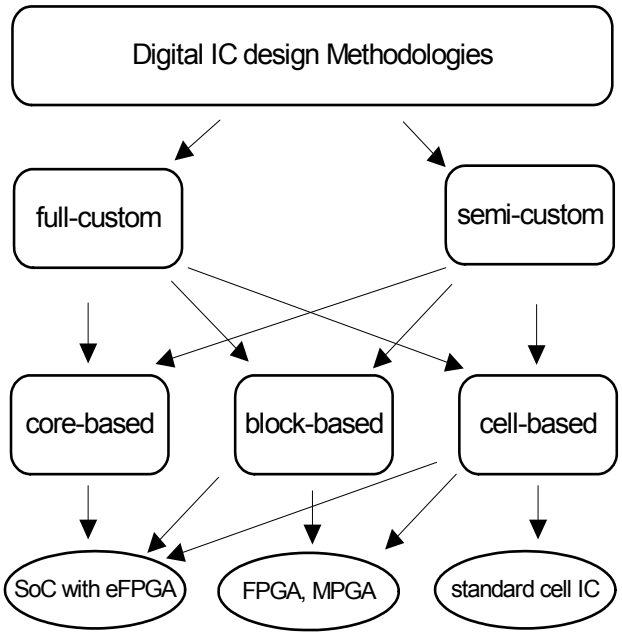
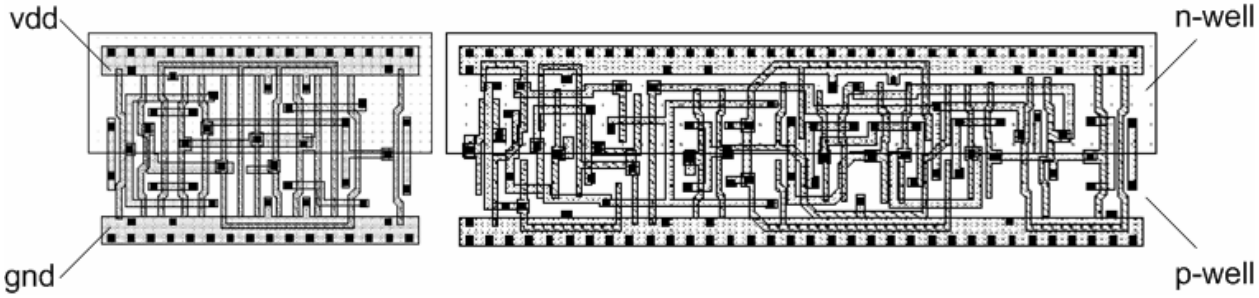


Figure 2.1: A general overview of existing integrated circuit design techniques

In order to reduce engineering effort of full-custom IC designs, some vendors resort to *semi-custom* design techniques to deliver new products much faster and at lower cost. Semi-custom design relies more on automated flows and Electronic Design Automation (EDA) tools to implement circuits. These EDA tools use libraries of pre-designed logic cells, blocks, and or cores to implement circuits in much shorter time but with some overhead costs, and certain restrictions on the IC designer.

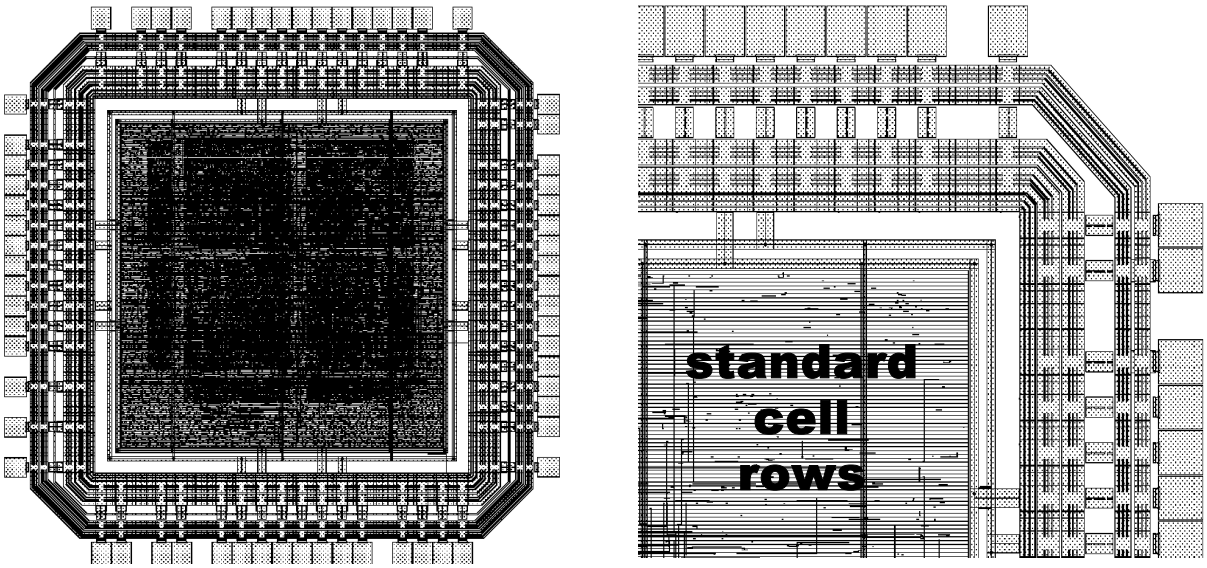
Some examples of semi-custom design include: cell-based, block-based and or core-based ICs. However, it is important to notice from Figure 2.1 that semi-custom designs do have elements of full-custom design and vice-versa. In other words, some components like cells, blocks, or cores in a semi-custom design, are implemented using full-custom techniques to some degree. The extent to which this "crossover" occurs depends upon performance requirements and time-to-market constraints. Furthermore, full-custom designs can also make use of cells, blocks and cores as needed.

In standard-*cell-based* semi-custom designs, vendors develop cell libraries that implement generic logic gates such as NORs and NANDs with different drive strengths and in accordance with a constrained physical layout format. These libraries of logic gates are supplied to designers who then map Register Transfer Level (RTL) descriptions of desired hardware behavior written in VHDL or Verilog, into gates using logic synthesis tools such as Design Compiler® and Cadence-PKS®. A more aggressive form of cell-based design allows designers to build custom cells and tools according to their own specifications and tailored for a particular circuit or group of circuits [46] [48] [53] [55]. Figure 2.2(a) and 2.3(b) below shows typical standard cell physical layouts. Usually PMOS devices are in the top part of the cell near Vdd and NMOS devices are in the lower part of the cell near Gnd. Figure 2.3(a) and 2.3(b) show a chip implementation of [19] that uses standard cells.



(a) simple function **(b) complex function**

Figure 2.2: Sample generic standard cell layout for two CMOS logic functions



(a) Full chip layout **(b) Top corner of chip**

Figure 2.3: Typical standard cell chip layout and a closer view of cell rows in chip

In order to implement the standard cell ASIC shown in Figure 2.3(a), special EDA tools are used for placement and routing. Placement involves reading a net-list of gates generated through logic synthesis and then arranging the corresponding standard cells in rows as shown in Figure 2.3(b). Placement is automated and can be constrained for area and performance. Standard cells are arranged in rows so that power (Vdd) and ground (Gnd) rails of adjacent cells are connected by abutment. Routing tools connect nodes (with metal wires) to implement the desired logic function.

In semi-custom *block-based* designs, vendors implement larger functions like arithmetic logic units (ALUs), multipliers, and adders as blocks that designers can also include in a standard cell design. One reason for doing this is that certain elements in a cell-based design are not efficiently implemented with generic standard cell-based logic gates like NANDs and NORs. Therefore, circuit blocks like ALUs, adders and multipliers would be included in a cell design to improve performance.

In *core-based* designs, vendors implement even larger and more complex circuits called cores, for inclusion with cells and blocks in ICs. The most common cores are microprocessor cores such as the ARM7®, MIPS, and PowerPC® cores. For example, Figure 2.4 shows a Virtex-II Pro® programmable platform IC [72] with an embedded PowerPC® microprocessor core (highlighted).

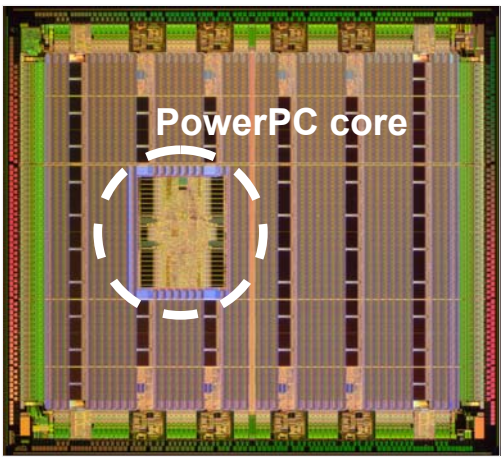


Figure 2.4: A programmable platform IC with an embedded PowerPC® processor

In order to design a *semi-custom* IC, designers often use a flow similar to the one shown in Figure 2.5. This flow is known as the ASIC flow [73]. As shown in Figure 2.5, design data that contains dimensions, timing, drive-strength, and power requirements for cells, blocks, and cores are used in EDA tools at all stages of the design flow. For example, during timing verification, timing and drive strength data for cells, blocks, and cores used in a design are combined with resistance and capacitance values extracted from metal traces after routing, and used to estimate the timing characteristics of all valid signal paths. If a placed and routed design satisfies all timing requirements, it is converted into a special layout file format called GDSII. A GDSII file contains all the design data required to manufacture a chip. If a design does not meet timing, a new and improved net-list must be created, placed, routed and again verified for timing until design goals are met (Figure 2.5).

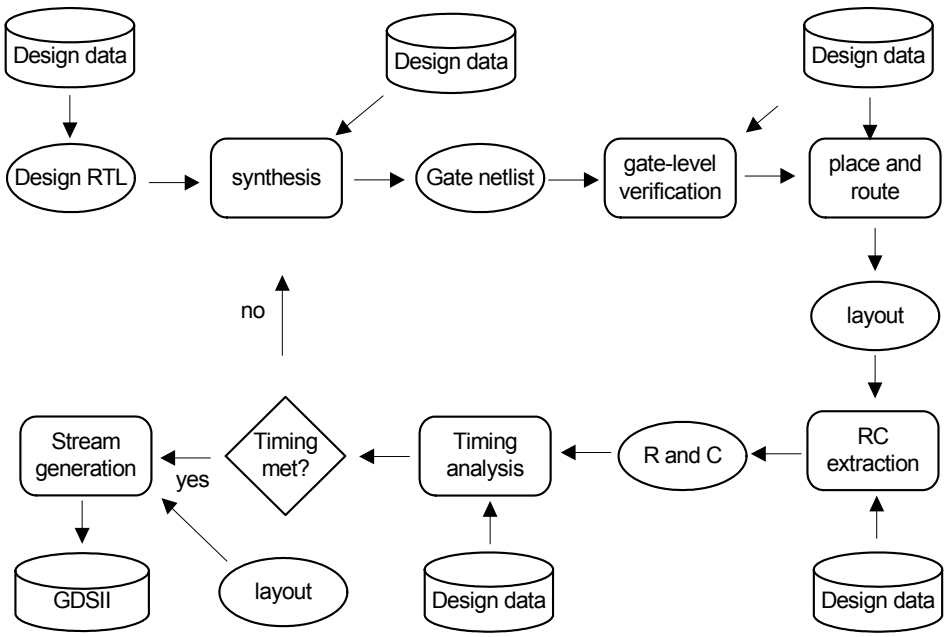


Figure 2.5: Typical flow for *semi-custom* cell, block, and core based ASIC design

Full-custom cell, block, and core based design flows includes most of the steps shown in Figure 2.5. Designs are created and tested at the transistor level via circuit schematics and analog simulations. In addition, full-custom designs are often hierarchical and structured so that complex circuit designs can be better managed and optimized. Designers often begin full-custom designs by first building cell-like entities that are then combined to form larger structures like blocks and cores. Placement of cells, blocks and cores relative to one another is done manually along with routing of interconnects. Manual place and route often gives the best results but it is very time consuming. Microprocessor chips and programmable logic devices are often designed in this manner. For example, programmable logic devices like FPGAs, are built from a single, highly-optimized full-custom programmable logic block or tile, that is replicated to form a two-dimensional logic array.

2.2 Embedded Programmable Logic IC Design Techniques

Programmable logic ICs in the form of Field Programmable Gate Arrays (FPGA) have been available for over two decades. In the last few years, there has been a push to develop embedded FPGA cores that reside in a chip alongside hardwired or fixed logic. The idea is that the exact logic function of a programmable core can be defined after fabrication, much like any stand-alone FPGA.

The advantages of this approach include the ability to cater to multiple customers with a single programmable chip, accommodate changes in standards or design specifications, or allow designers to fix design errors that are caught after chip tape-out (if they are suitable for correction by the embedded FPGA). An embedded FPGA or *programmable fabric* is usually arranged as a structured array of logic blocks, switches and routing tracks. A logic function is implemented on the array by using SRAM cells set to 0 or 1 to define logic functionality as well as define routing connections.

While programmable fabrics afford designers a tremendous amount of flexibility, there is significant overhead associated with this approach when compared to fixed logic (hardwired) ASICs. Specifically, the area of programmable fabrics can be 50 to 100 times higher, speeds can be 2 to 10 times worse, and power dissipation is substantially higher. Over the past few years, the creation of efficient programmable logic cores has been an active area of research and development in an effort to reduce overhead, improve ease of use, and make this option more attractive to chip designers.

So far, leading approaches for embedded programmable logic implementation include embedded Field Programmable Gate Arrays (eFPGAs) [08] [41] [44] and Mask Programmable Gate Arrays (MPGAs) [45] [60]. This section presents a general architectural overview of programmable logic

devices within an eFPGA context, followed by a general description of MPGAs. The important features of each technology are emphasized, as well as their associated advantages and disadvantages.

2.2.1 Embedded Field Programmable Gate Array (eFPGA)

In this section, an overview of the popular island-style FPGA architecture [11] is presented. This architecture (Figure 2.6) is the basis for many commercial programmable logic devices used today.

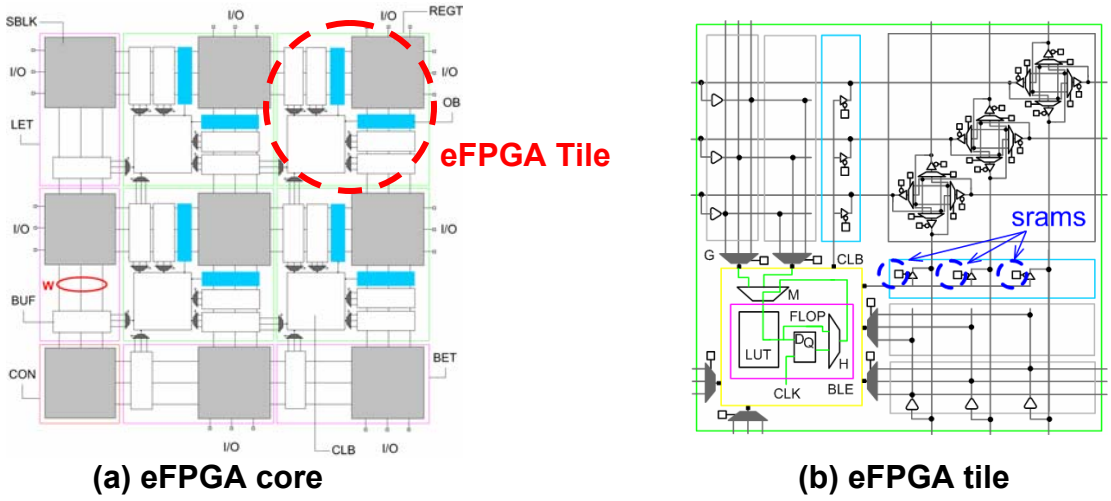
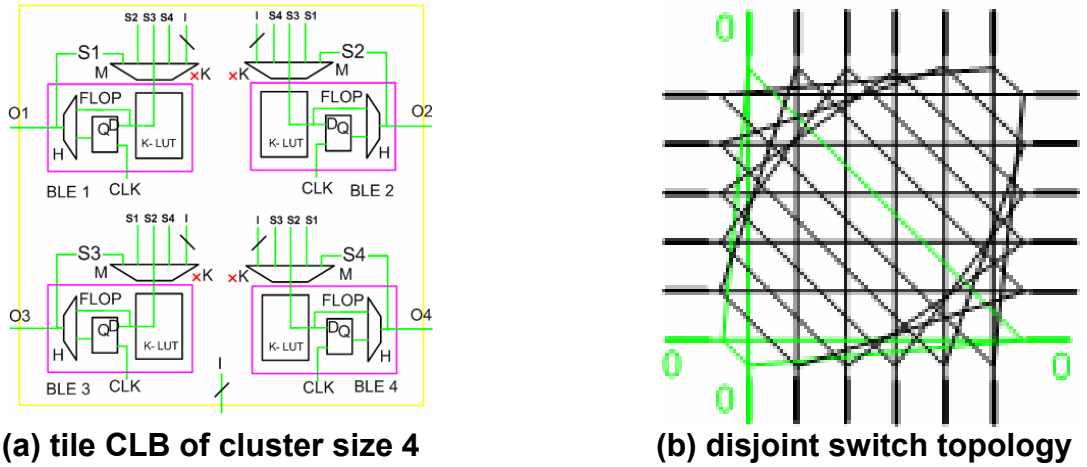


Figure 2.6: 2x2 island style eFPGA *core* architecture and regular *tile* architecture

In Figure 2.6(a), CLB is a configurable logic block, SBLK is a routing switch block, LET is a left edge tile, BET is a bottom edge tile, CON is a corner tile, REGT is a regular tile, BUF contains track buffers for input connection blocks, and OB is an output connection block. Also, Figure 2.6(a) shows an eFPGA core as a structured array of logic clusters, switches and routing tracks of width W. The enlargement in Figure 2.6(b) shows that each regular tile contains a CLB, a switch block, connection blocks and SRAM cells for configuration. The left and bottom edge tiles only contain switch-blocks and connection blocks for routing. The corner tile connects the LETs and BETs.

Because logic blocks implement logic functions, they constitute the *logic architecture* of a programmable fabric. Similarly, switch blocks and connection blocks constitute the *routing architecture* because they route signals between tiles, and between the eFPGA and external logic via I/O ports. SRAMs and decoders constitute the *configuration architecture* since they configure the fabric. In essence, programmable logic devices are a combination of logic, routing, and configuration architectures.

Within the logic architecture, CLBs comprise one or more basic logic elements (BLEs). The number of BLEs in a CLB is its cluster size, N. For example, in Figure 2.7(a) the cluster size is 4. Each BLE in a CLB is comprised of a K-input look-up-table (K-LUT), an edge-triggered flop (LUT-flop) to generate a registered copy of the LUT output, and an output selection multiplexer ("H" in Figure 2.6(b)) to select between registered and unregistered LUT outputs. LUT input multiplexers ("M" in Figure 2.7(a)) are used to drive K inputs of all N LUTs in a CLB of size N (4 in this example).



(a) tile CLB of cluster size 4 **(b) disjoint switch topology**

Figure 2.7: programmable logic CLB of size 4 and a disjoint switch block topology

The switch block design of the routing architecture is typically based on a disjoint topology [11] as shown in Figure 2.7(b). This means a given net cannot hop between track "lanes". For example, a net that enters a switch on routing track 0 as shown in Figure 2.7(b) will exit and continue on track 0

until it terminates. Connection blocks connect CLB inputs and outputs to adjacent routing channels. For example, in Figure 2.8(a) an input connection block multiplexer, G, selects a net from track "2" in the routing channel to drive a cluster input I (the input track buffers are used to minimize loading). Output connection blocks drive cluster outputs (O in Figure 2.8) onto the routing channel. An output connection block has a tri-state driver for each track that it can drive. The value in the SRAM on the enable input of the tristate-driver determines whether or not a routing track is driven.

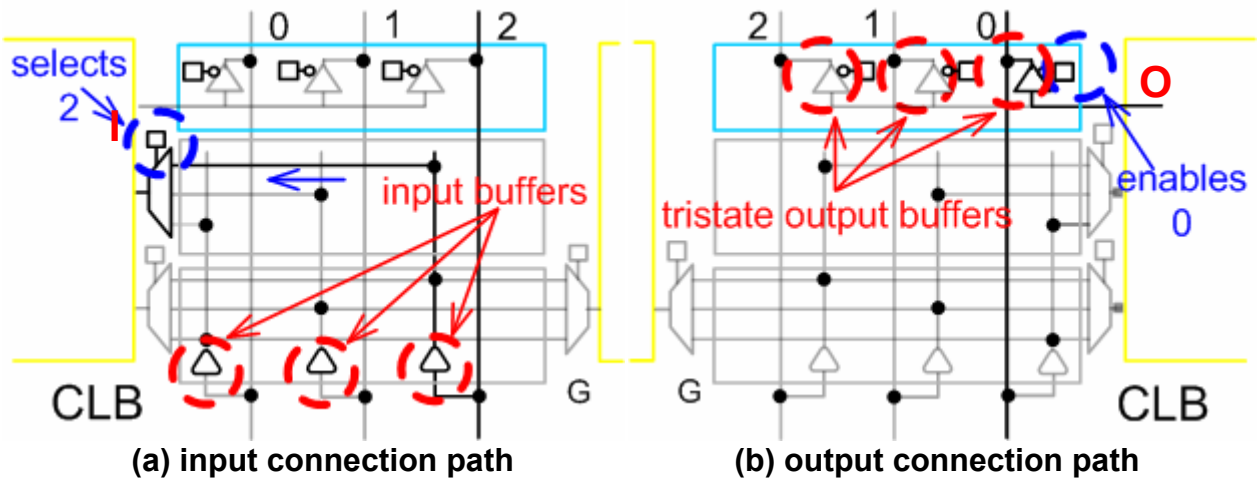


Figure 2.8: Input and output connection blocks in the eFPGA routing architecture

The configuration architecture includes configuration SRAMs, decoders, and a configuration finite state machine. A row decoder selects a row in Figure 2.6a for programming, and combined with the column decoder, can select a single tile for programming. A state machine controls configuration.

The bits (sequence of zeros and ones) needed to configure a programmable logic device are generated using specialized CAD tools such as VPR [11]. The details of this tool are outside the scope of this work. However, at a high level, these CAD tools contain detailed models of programmable logic architectures like the one shown in Figure 2.6. These models are used to decide suitable placements for circuits that have been mapped into LUTs [11]. Once a suitable placement

for a circuit is found, it is routed [11]. FPGA routing involves finding suitable paths between all the connected nodes (sources and sinks)in a circuit implementation given the available routing resources in the target programmable logic device and any user-supplied path timing constraints.

Figure 2.9(a) shows a VPR screen capture of a 6x6 programmable logic device after placement. The darker blocks represent occupied logic clusters. Figure 2.9(b) shows the same architecture after routing. Figure 2.9(b) shows just routing tracks that are used. Placement and routing files for a given circuit implementation are generated within VPR and used in software programs that generate the appropriate sequence of zeros and ones needed to program a circuit on a programmable device.

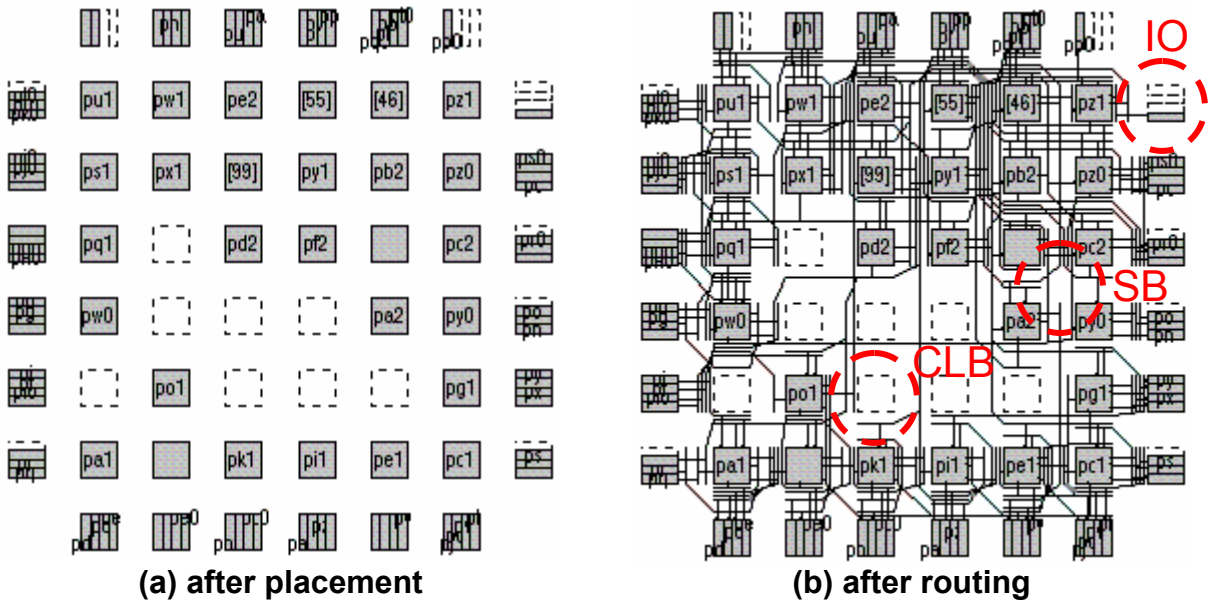


Figure 2.9:VPR placement and routing of a sample user circuit in a 6x6 logic array

An eFPGA is essentially an unpackaged standalone FPGA (Figure 2.9) that has been stripped of its I/O pad ring and adapted to function as an embeddable IP core. Like a standalone FPGA, the exact logic function of the eFPGA can be decided after fabrication. Typically, an eFPGA would be used to implement small to medium logic functions like microprocessor accelerator functions or hardware data encryption algorithms that need to be reprogrammed periodically to adapt to changes.

2.2.2 Embedded Mask Programmable Gate Array (MPGA)

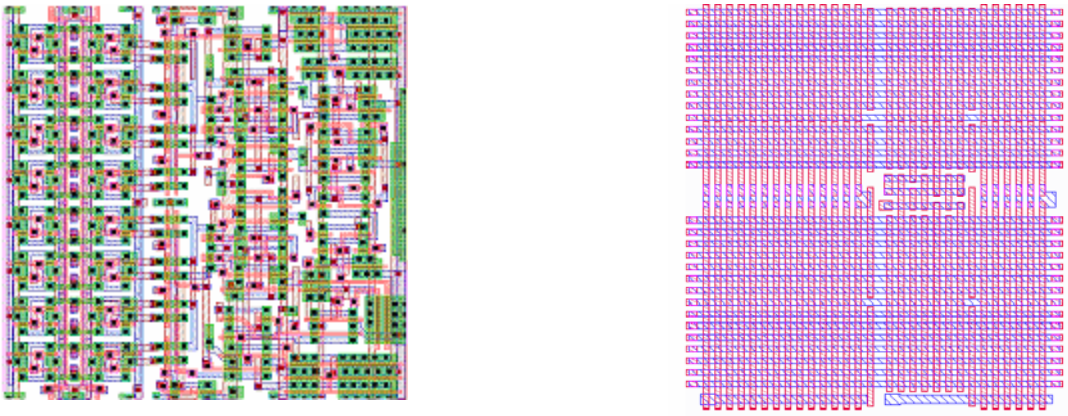
More recently, MPGA devices like the eASIC® core [45] were introduced to eliminate some of the shortcomings of eFPGA cores. An eASIC core is modeled on SRAM programmable look-up-tables but the routing infrastructure between look-up-tables is quite different from eFPGAs. In particular, the routing interconnect is either metal/via or just via programmable (one-time programmable). The advantage here is that the extra logic needed for switch elements in the routing of a typical eFPGA are no longer required. Therefore MPGAs tend to be are smaller, faster and more power efficient.

Furthermore, the wafers associated with a particular design can be processed in advance of final metal/via programming. Therefore, the turnaround time for a programmed design is only about a week or two compared to a few months for so-called hardwired ASICs that are not programmable.

Although current MPGA cores are denser, faster, and more power efficient than an eFPGA, there is a loss of flexibility because the routing interconnect is not ***re***programmable. For example, once via connections between metal layers for a particular circuit implementation are made, they cannot be changed. Any further changes to the routing interconnect will require a new silicon die with an unprogrammed embedded MPGA. Although changes in LUT functionality is possible (reprogrammable SRAMs), the scope of such changes is limited once the routing has been finalized.

Figure 2.10(a) below shows a transistor-level full-custom layout of an MPGA tile [45] modeled on the popular island-style architecture, and similar in many respects to the eFPGA tile shown in Figure 2.6(b). For example, the LUTs in an MPGA CLB are also programmable using SRAMs. The main difference however, is the absence of reprogrammable routing switches in the MPGA tile. Instead,

an upper metal layer grid of potential global routing connections is used to form an interconnect mesh or *connectivity fabric* [45] as Figure 2.10(b) shows. After fabrication, this grid has no connection to lower layers of metal. Instead connections are only made after it has been determined what circuit will be implemented in the MPGA, and which connections to the lower metal layers (hardwired) will be needed to implement a logic function. Such connections are made with a via mask layer that connects the uncommitted connectivity fabric to the hardwired "base array" [45] in Figure 2.10(a).



(a) MPGA base array **(b) Connectivity Fabric**

Figure 2.10: A MPGA logic tile layout, and a connectivity fabric of top metal layers

Table 2.1: Comparison of eFPGA, MPGA and standard cell logic design methods

0.18 micron CMOS	eFPGA	MPGA(eASIC®)	Standard Cell IC
Density (gate/mm^2)	1.5K	30K	60K
Performance(MHz)	100	400	600
Power(nW/Gate/MHz)	1000	40	20-30
NRE ($)	0	30K	500K
Prototype TAT(days)	0	5-10	20-40

Table 2.1 [45], presents useful data on key features of eFPGA and MPGA technology that further illustrates important differences between these embedded programmable logic solutions. For example, the absence of *re*programmable routing in MPGAs makes them 20 times smaller than an

eFPGA of the same logic size. The Non-Recurring Engineering (NRE) costs and development turn-around-times (TAT) for an eFPGA is more or less zero because a designer simply needs to load the appropriate bitstream to get a working chip. On the other hand, designs with MPGA cores require one or two metal and or via mask sets to program a core. In this case, IC manufacturing services are needed and accounts for the NRE and TAT overhead. The NRE costs and TAT for standard cell IC (non-programmable) designs are even higher because all IC layers must be fabricated to realize a working chip. However, standard cell IC designs have superior area, speed, and power efficiency.

An MPGA offers "static" or one-time programmability that is suited to designs that do *not* require "on the fly" *re*programmability for bug fixes, or hardware adaptability to changing standards. For designers, who require reprogrammability, the eFPGA is a better design choice, but as already mentioned there are large area, performance, and power issues associated with this design approach.

2.2.3 Example Application: Bluetooth Base-band System-on-Chip

To illustrate potential applications of eFPGA and MPGA cores in a real chip design, we use a Bluetooth SoC platform to show on-chip components that could be made hardware programmable.

Bluetooth is another name for the IEEE standard, 802.15, a 2.4GHz wireless radio frequency (RF) communication protocol with an operational range of about 10–100 meters. A Bluetooth SoC platform implements hardware and software components of the Bluetooth protocol as specified by the Bluetooth Special Interest Group (SIG) [74]. As in most wireless SoCs, the Bluetooth SoC has a microprocessor core, interface peripherals such as General Purpose I/O (GPIO), system buses, radio

codecs, an RF interface and a dedicated ASIC core that implements the Bluetooth baseband protocol. When fully implemented this SoC is a multi-million-transistor IC design with over 200 I/O pins.

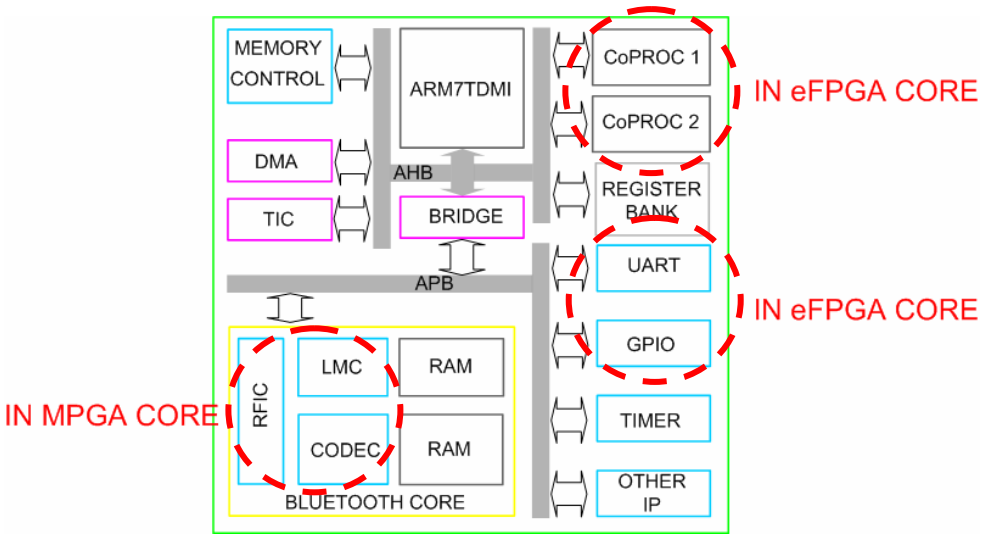


Figure 2.11: A System-on-a-Chip platform that implements the Bluetooth Protocol

Figure 2.11 above shows a typical system level illustration of a Bluetooth SoC design. In this figure, the larger blocks in the Baseband, such as the Link Management Controller (LMC), Radio Frequency Interface Controller (RFIC), or radio encoder-decoder (CODEC) could be replaced with one MPGA fabric. The LMC is a good candidate for programmability, because it changes as the Bluetooth standard is revised. Furthermore, the RFIC and radio CODEC implementation depends on the RF front-end chip and radio that a customer supports and so should also be adaptable. MPGAs facilitate standards revisions and or product differentiation for customers because stockpiles of unprogrammed chip die can be produced and then later programmed and repackaged as needed. Using an eFPGA in this case would result in an unacceptably high area, power and speed penalty. Instead, eFPGAs are best suited to small and medium functions like processor accelerators, also known as coprocessors (CoPROC1 and CoPROC2 in Figure 2.11) and perhaps low speed I/O protocols such as General Purpose I/O (GPIO) and Universal Asynchronous Receiver Transmitter

(UART). Accelerator eFPGAs can be used to speedup execution of parts of the Bluetooth software Protocol Stack and I/O eFPGAs can be used to implement updated communication protocols.

2.3 Embedded Programmable Logic as an Intellectual Property (IP)

SoCs (like the Bluetooth chip) are not designed entirely by a single design team; instead, parts of the chip are comprised of Intellectual Property (IP) obtained from third-party vendors. In other words, parts of, and in some cases, the entire SoC, could be designed with in-house or third-party IP. The aim of this paradigm shift, sometimes called the SoC revolution [07], is to boost productivity through IP reuse. Productivity gains for derivative designs come from the ability to incorporate pre-designed and pre-verified IP like MPGA and eFPGA cores, into new designs relatively quickly.

In the IC design industry, there currently exists one way of embedding hardware programmable logic in a SoC design such as the Bluetooth IC of Figure 2.11, namely, as hard Intellectual Property or hard IP [41][44]. More recently in [02][14], a somewhat new approach for embedded programmable logic design that was first used in [01], was introduced as soft Intellectual Property or soft IP. Since the focus of the rest of this thesis will be on embedded FPGA design, the following sections, describe the hard and soft IP approaches within an embedded FPGA design context.

2.3.1 Hard eFPGA IP

The current approach for eFPGA usage is to purchase a core as a hard IP block from a vendor and integrate the block into the design flow with the rest of the design circuitry. Hard IP implies that the user cannot change the eFPGA design in any way because physical dimensions, speed, power

efficiency, and other characteristics are already fixed. The major advantage of this approach is that the user does not need to design and build the programmable core or fabric, because a vendor has already pre-designed a number of fabrics of differing sizes using structured, full-custom layout techniques. Full-custom design techniques ensure that every hard eFPGA core is highly optimized.

However, in an effort to reduce design and support costs, hard eFPGA vendors offer as few variations of the eFPGA core as possible. As a result, there are numerous sources of inefficiency or underutilization. First, vendors typically offer a single eFPGA architecture that may not be ideal for a particular application. Second, some eFPGA architectural parameters, such as the number of inputs to a LUT, are determined by implementing a large number of proprietary benchmark circuits and "choosing the one which gives the best performance on average". While this approach makes sense for potentially broad ranges of applications, it is not necessarily the best approach for a specific application domain. Third, for a given architecture, the vendor can only offer a limited selection of logic capacity or core sizes. For example, Actel's Varicore [44] is offered only in sizes of 512, 1024, 2048, and 4096 four-input LUT. As a result, if the logic gate requirements just slightly exceed a given core size, the user must buy the next-largest core which is 2x larger. Fourth, the smaller eFPGA cores are sometimes based on the same "layout tile" as the larger cores, leading to unnecessary interconnect (routing) capacity for core sizes with fewer LUTs and logic capacity.

To illustrate these last two problems, the area overhead arising from having too many LUTs and too many wires is shown in Figure 2.12. The x-axis is the desired number of LUTs for a given application. The y-axis is the overhead or how much larger than necessary the smallest available hard eFPGA core is with logic and routing considered. As the number of LUTs needed increases along the x-axis (including any spare LUTs the user may wish to reserve), the overhead decreases. Once

the logic capacity of a given core size is reached, e.g., 512 LUTs, the overhead drops to 1.5X (now due to routing overhead only). However, if more LUTs are needed, the next larger size of eFPGA is needed and the overhead jumps to almost 2.8X. The saw-tooth pattern in Figure 2.12 is repeated as each eFPGA core size becomes fully utilized and the next larger eFPGA core size is selected.

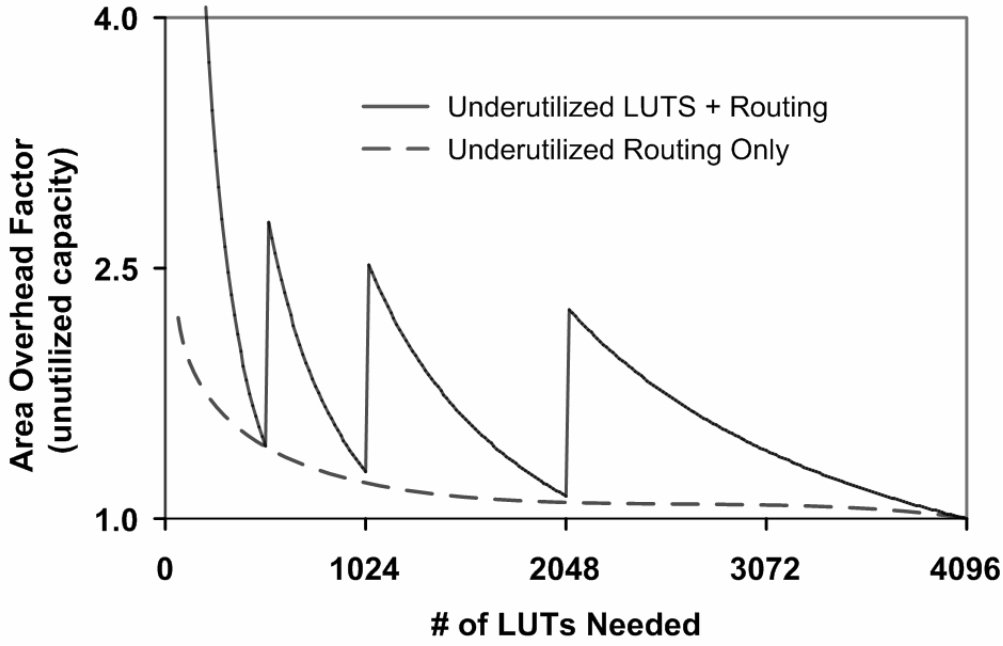


Figure 2.12: An example of architectural inefficiencies in existing hard eFPGA IP

2.3.2 Soft eFPGA IP

Given the disadvantages of the hard IP approach, an efficient and less restrictive approach would be preferable. A relatively new approach has been used to automatically generate an eFPGA fabric within the ASIC design flow [01][02][15]. This is referred to as the soft PLC [02] [14] approach. The main idea here is that an eFPGA architecture is described in behavioral RTL using Verilog or VHDL, and implemented alongside the rest of the user logic. Users implement the eFPGA using logic synthesis tools that map the behavioral RTL to a standard cell library within the ASIC design

flow. ASIC place and route is performed as usual to create the physical IC layout. The main advantages of this approach are flexibility and ease of use. Furthermore, this approach does not restrict a designer to certain foundries, as may sometimes be the case with hard eFPGA cores.

Although the soft IP approach affords a designer much flexibility, significant inefficiencies exist in this approach. For example, because this flow relies exclusively on logic synthesis for technology mapping (mapping of RTL constructs to logic gates in a standard cell library) some elements of the architecture are not efficiently built. For example, SRAMs that are used to hold program bits in an FPGA do not exist in standard cell libraries and so flip-flops (much larger cells) are used instead. Similarly, multiplexers for implementing LUTs and other large multiplexers in an eFPGA architecture are built from discrete CMOS logic gates like NANDs and NORs. As a result of these logic implementation inefficiencies in soft IP, significant overheads in area, delay, and power occur.

Furthermore, there is no structure or regularity imposed during logic synthesis or layout of soft eFPGA cores [02]. Consequently, it is possible that even identical repeated structures in the layout have different path delays [14]. This is not necessarily a flaw, but it makes timing characterization of eFPGA CAD tools more difficult. The tools and flow are made more complex because each timing arc in the architecture must now be considered separately. In addition, wires may be longer than necessary and thus increase delay. This is in contrast with the hard eFPGA IP approach that uses architectures with well-defined repeated structures to simplify physical layout and CAD tool design.

Finally, although the flexibility and ease-of-use afforded by the "soft" eFPGA [02] [14] [15] design methodology is highly desirable, design improvements that minimize the area, delay and power overhead [01] [02] [14] associated with standard cell based embedded FPGAs are clearly needed.

2.4 Research Problem Definition and Thesis Research Focus

Following from preceding descriptions, hard and soft eFPGA IP design methodologies can be viewed as two separate extremes of a design spectrum. On one extreme lies the hard eFPGA IP methodology, based on full-custom design and with ***limited flexibility*** but ***relatively*** high density, speed, and energy efficiency. On the other extreme is soft IP, which affords ample flexibility (due to a semi-custom ASIC flow) but has much lower density, speed and energy efficiency relative to hard eFPGA IP. Given these problems with both approaches, an optimum solution should aim to combine their best features by: retaining the design flexibility afforded through design automation (e.g. ASIC flow), and, at the same time, incurring significantly less area, speed and power overheads.

Following from the above problem description, the focus of this research is to investigate possible improvements in the quality (area, speed and power efficiency) of eFPGA circuits implemented within an automated design framework. An automated framework such as the ASIC flow ensures that the flexibility afforded by the soft IP approach is preserved. In addition, it has previously been shown that the quality of standard-cell-based designs implemented with the ASIC flow can be improved significantly through customization. For example, a study in [46] showed that the replacement of generic standard cells with so-called "crafted" (architecture specific) standard cells in a *datapath* circuit design reduced area and delay overhead factors to within 1.64 and 1.11 respectively relative to an identical full-custom implementation. Furthermore, in [27] [68] [69] [70], an area overhead factor of 1.36 relative to a full-custom implementation was reported using a flow similar to the ASIC flow but instead using *custom designed "FPGA-centric" EDA tools and non-standard custom cells.*

In this work, the ASIC flow is also used for eFPGA design and implementation. For this research, an island-style programmable logic architecture has been selected as the reference platform because such a choice has numerous benefits. For example, this architecture is the basis for most of the programmable logic devices in use today, hence comparisons to commercial (full-custom) devices are more relevant. Similarly, the CAD tools for this type of architecture are more widespread and mature (e.g., VPR tool) hence, they can be leveraged directly without any need for new CAD tools.

Next, the impact of tactical architecture-specific cells on the design quality of standard-cell-based eFPGAs is investigated. In particular, after the island-style architecture has been implemented using generic standard cells and the ASIC digital design flow, key sources of area, speed, and power inefficiency are identified and improved through circuit design and customization. This involves creating new standard cells called tactical cells that can be incorporated in the ASIC design flow.

Furthermore, the implications of such an approach (as described above) for eFPGA IP design are investigated. Specifically, design results obtained for a set of benchmarks are compared with results that would be obtained if the same benchmarks were implemented using the hard IP approach. In addition, two modules in an actual Bluetooth Baseband SoC, namely, the base-band ***frequency hopping*** module and ***data encryption*** module are implemented in a standard-cell-based eFPGA. This study gives an idea of the impact these eFPGAs can have on the area of a "real-world" design.

Finally, the ASIC flow was chosen for this research because it was decided that it would be useful to fully investigate the appropriateness of existing ASIC tools for eFPGA design before embarking on the expensive process of designing a new set of EDA tools for automatic eFPGA circuit generation.

Chapter 3

An Embedded Programmable Logic Architecture Family

3.1 Island-Style eFPGA Architectures

Any eFPGA design approach that is flexible (like the soft approach) must include a basic architecture that is configurable. The popular island-style architecture provides such an opportunity. Therefore, in order to explore the impact of configurable architectures on eFPGA IP design, this chapter describes some variations of the island-style architecture so that area and speed tradeoffs can be investigated over a set of benchmarks. The goal here is not necessarily to design the "best" architecture because this is to some extent a function of the application domain [01][03]. However, there are cases where there exists a clear advantage of one architecture over another. For example, data in [20] suggests that the area of LUT-based island FPGA architectures could be improved by making the routing interconnect directional rather than bidirectional. Implementing this, and other potential eFPGA architectures, is beyond the scope of current work since we focus on block-level (lower level) architecture optimizations. Architecture optimizations of the kind presented in [20] are left to future research work. Consequently, the eFPGA architectures presented in the following subsections are really derivative architectures of the well-known island-style FPGA architecture.

The research work presented in this section focuses on switch element circuit design (switch block design) as well as resolving some problems normally associated with programmable logic design and implementation, such as I/O design, design for testability issues, and power-up issues. The purpose of revisiting some of these issues is to determine if any improvements can be made with

better circuit design. In addition, the total cost and overhead due to these architecture components can be properly assessed when more typical design considerations are taken into full account.

3.1.1 Bidirectional Routing Architectures for eFPGA design

In this section, four derivative architectures with bidirectional routing tracks are presented. The architectures differ only in the switch used in the switch-block. These architectures are suitable for implementing sequential and combinational circuits. The first architecture uses tri-stated routing switches [11]; the second uses a multiplexed switch from [49], the third and fourth are based on novel multiplexed switches that were designed to improve the speed of the original design from [49].

Figure 3.1(a) shows a tile architecture based on tri-state routing switches [11][49]. As the enlarged Figure 3.1(b) shows, each potential net route is controlled by a tri-state buffer pair. If a net is routed from left to right across a switch, only one of two tri-state buffers (highlighted by the circle in Figure 3.1(b)) in that path can be turned on or enabled. The other buffer must remain turned off or else a local combinational feedback loop will be created. Therefore, for every active route through the switch, only one tri-state buffer is turned on. If no net is routed, both buffers in a pair are disabled.

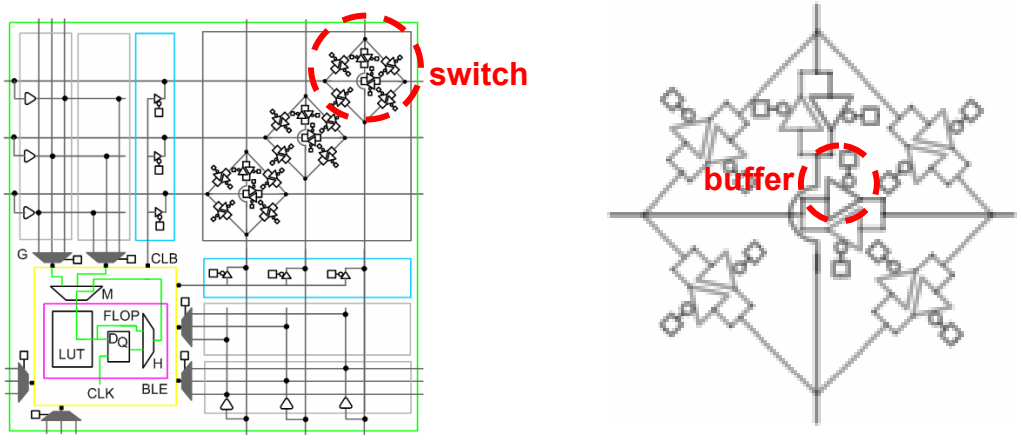


Figure 3.1: An island style eFPGA tile architecture with tri-stated routing switches

The *logic* architecture of the second derivative architecture shown in Figure 3.2(a) is identical to the tri-buffered island architecture in Figure 3.1(a). However, the routing architecture is based on the multiplexed-switch presented in [49]. Each of the four multiplexers in this switch element routes a net from *one* of *three* possible inputs to a *fourth* and different direction. For example, in Figure 3.2(b), a multiplexer and buffer pair (mux-buf) on the right side of the switch element (highlighted by the circle in Figure 3.2(b)) can route nets from either the bottom, left, or topside of the switch to the right output. The unused (floating) multiplexer inputs in Figure 3.2(b) can be tied to ground [20].

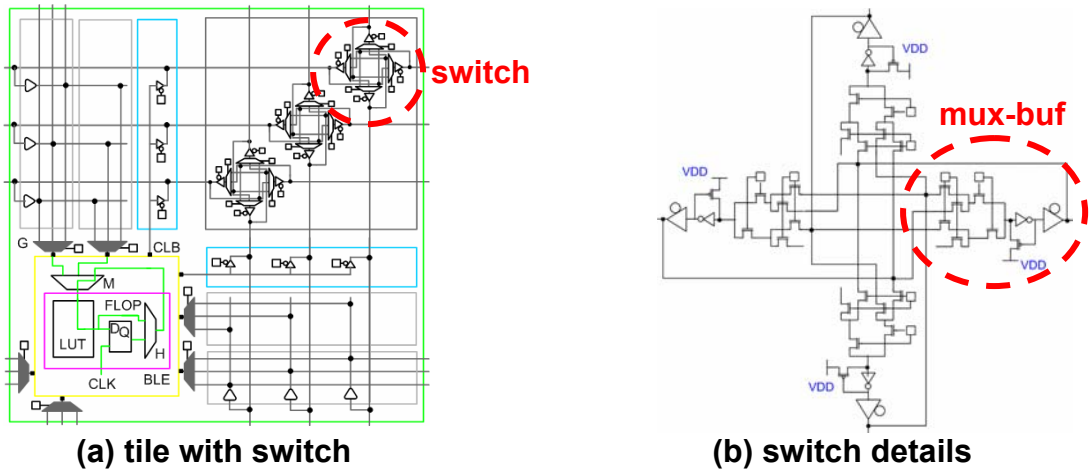


Figure 3.2: Island style eFPGA tile architecture with multiplexed routing switches

Figure 3.2(b) also shows that a net routed from left to right (or from any of the other two valid sources) must traverse two levels of a pass transistor tree network and buffers before exiting. When compared to the tri-buffered switch presented earlier, this architecture would have a higher delay overhead due to the NMOS pass-transistor multiplexing logic. However, previous work [49] showed this design to be more area efficient. These tradeoffs are examined further in the next chapter.

The architecture in Figure 3.3(a) is based on a novel switch element called "Improved multiplexer switch 1". It is similar to the original multiplexed switch presented earlier but with some important modifications. As in the previous design, each of the four multiplexers in this switch routes a signal from *one* of *three* possible directions to a *fourth* direction. The difference here is that nets routed from left to right (or vice-versa) and from top to bottom (or vice-versa) across a switch are faster because multiplexing logic is bypassed. Therefore, any critical nets that are routed vertically or horizontally across several tiles in an eFPGA would have reduced delay. However, there is now an extra tristate buffer for each of the four exit routes in a switch, and this contributes to the switch area overhead.

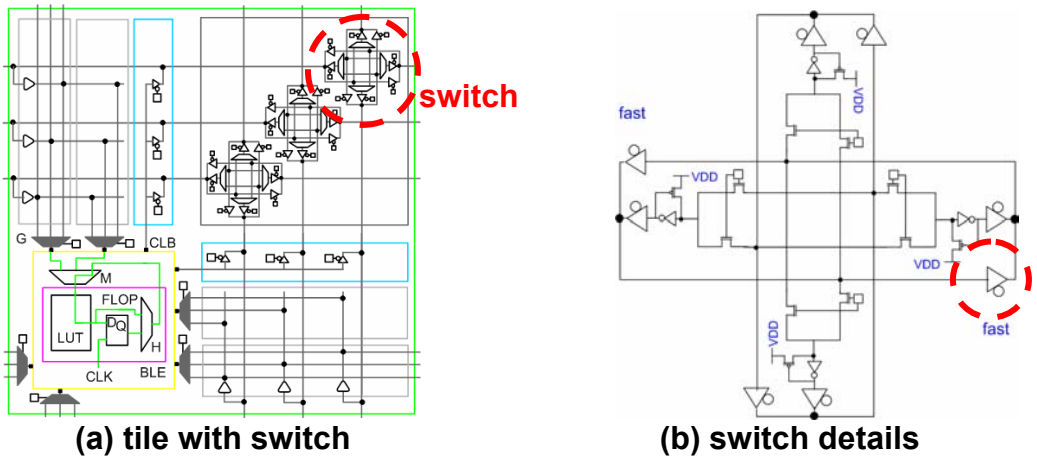
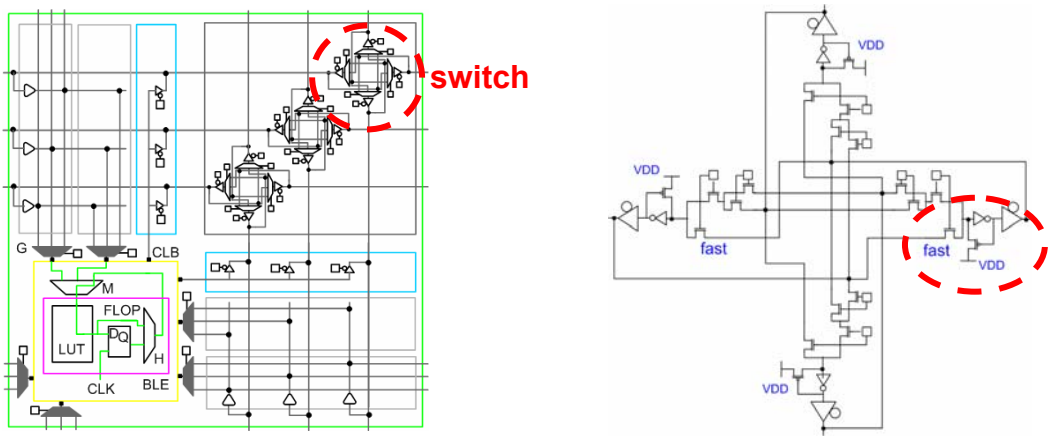


Figure 3.3: Island style eFPGA tile with improved multiplexed routing switch 1

Figure 3.3(b) shows that a smaller delay overhead would be experienced by a net routed from left to right in the new multiplexed switch compared to the original switch design in Figure 3.2(b). For example, a net routed from left to right across the switch experiences a single tri-state buffer delay (circled in Figure 3.3(b)) by completely bypassing the extra multiplexer delay overhead. The same is true for the reverse route (right to left) and also for vertical routes (top to bottom and reverse). Only routes changing direction (e.g., turning from top to left) incur a multiplexer delay overhead. Furthermore, only a single level multiplexer is used and this results in a smaller delay overhead.

A second novel switch *configuration* called "Improved multiplexer switch 2" was devised to reduce the area overhead of the previous design and improve the speed of the original multiplexer switch architecture [49]. In the novel tile architecture shown in Figure 3.4(a) below, an attempt is made to combine the area efficiency of the original multiplexer switch design with the speed of "Improved multiplexer switch 2". In particular, it was observed that the speed of the original multiplexed switch could be improved without any area penalty, simply by rearranging internal routes. To achieve this, an "unbalanced" NMOS pass transistor network can be constructed as shown in Figure 3.4(b). Similar to the previous novel switch design, horizontal and vertical routes are connected to the "fast" route through the pass-transistor tree network, thus speeding up these switch routes.



(a) tile with switch **(b) switch details**

Figure 3.4: Island-style eFPGA tile with improved multiplexed routing switch 2

3.2 Architectural Issues

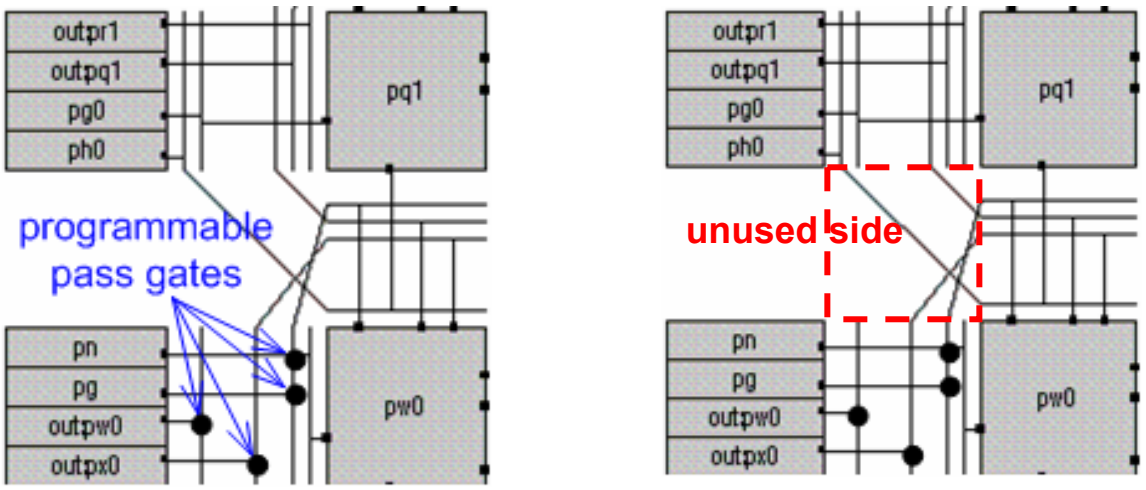
During architecture specification, a number of practical design issues were considered so that this design could be used in an actual SoC implementation at some point in the future. These issues include I/O design, design-for-testability (DFT) issues and configuration SRAM power-up issues. In the following subsections, each problem is described and possible design solutions are presented.

3.2.1 Input/Output Design

In stand-alone FPGA chips, an I/O pad ring surrounds the programmable core as shown in Figure 2.9. The large size of these pads means that the number of I/O on an FPGA device must be kept somewhat low in order to save area. In embedded FPGAs, these large I/O pads are not needed and therefore more I/O can be included without any significant increase in core area or delay. In fact, eFPGA I/O are simply tiny metal segments around the core edges that serve as "contacts" for external nets. It is important for eFPGA fabrics to be rich in I/O because some applications may be very I/O intensive compared to their logic size. It would be undesirable to select a much larger eFPGA fabric simply to gain access to more I/O. This is a problem that standalone FPGA users sometimes face. Moreover, any unused I/O in the eFPGA adds very little to the area overhead.

In a typical implementation of a generic island style FPGA architecture [11] [13] [49], I/O pads are connected to the channel routing tracks via programmable pass-gates (shown as darkened circles in Figure 3.5(a)). Figure 3.5 is a section of the left edge of the programmable device shown earlier in Figure 2.9 where "pq1" and "pw0" are CLBs. The remainder of the shaded rectangles in the figure such as "pn", and "out:pq1" are input and output pads respectively. The pass-gates used to connect

I/O pads to the routing channel contribute to the area overhead, and their number increases as the number of connections, routing channel width (W), and number of I/O increase. To avoid this, the embedded FPGA I/O need to be redesigned in a more area-efficient manner. Rather than have extra logic like pass gates to connect I/O to routing tracks, the switch-blocks around the edges of the embedded FPGA could be used to implement the I/O. This is possible because some of the logic in switch-blocks around the edges of an FPGA chip are unused (after place and route). For example, Figure 3.5(b) is a section of the *left edge* of the placed and routed device in Figure 2.9(b). In Figure 3.5(b) no nets are routed to the *left* of the switch-block (the switch-block is the region enclosed by dashed lines). Similarly, along the *top edge* of the FPGA (refer back to Figure 2.9(b)), no nets are routed to the *topside* of the switch blocks, and likewise for the bottom and right edges.

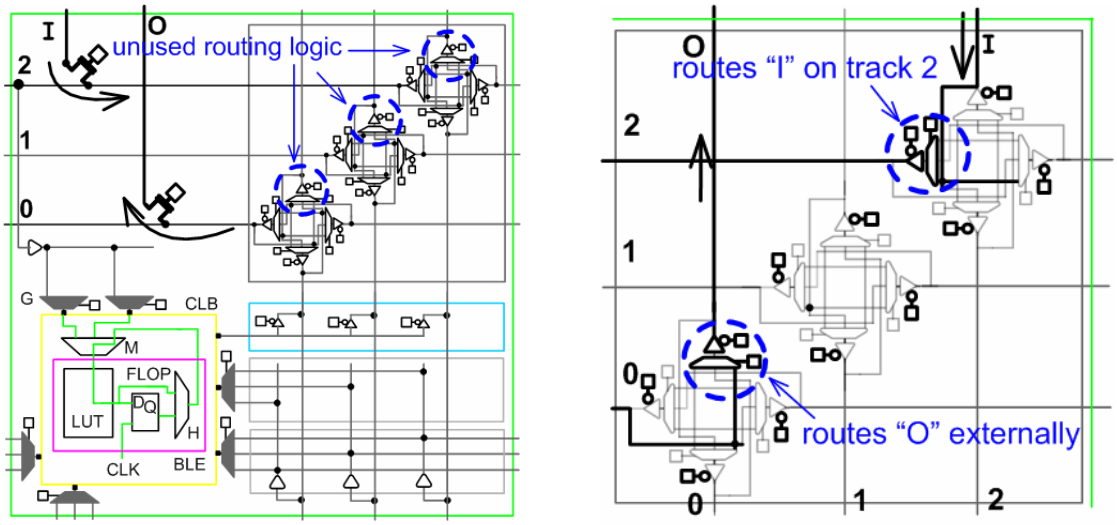


(a) I/O pass gate connectors **(b) switch with unused left side**

Figure 3.5: Programmable I/O pad connections and unused switch connections

As a result, unused "free" routing logic in the switch blocks around the edges of the FPGA could be used for routing I/O. Typically, an external input such as "I" in Figure 3.6(a) would be routed into the tile via the programmable pass-transistor that connects to routing track "2". Similarly, an external output such as "O" would typically be routed through the pass transistor that connects to routing track "0". Assuming that the tile in Figure 3.6(a) is situated on the top edge of an eFPGA

fabric, there would be no reason to route any nets, except external nets, through the top edge of the switch-block. Figure 3.6(b) illustrates how the connectivity for input "I" and output "O" could be implemented using the "free" routing logic along the top edge of a switch. As shown, the external input "I" can be routed from the top port of the switch element that controls routing track "2" because the top port is unused. Further, this signal can exit from the left port of the same switch (see Figure 3.6(b)), since this segment of track is already "reserved" for input "I" in Figure 3.6(a).



(a) typical design **(b) *novel* design**

Figure 3.6: Equivalent external input routing in standalone and embedded FPGA

Similarly, as shown in Figure 3.6(b) the external output "O", is driven by a net that is routed from the left port of the switch element on routing track 0 to its top port (the external output). This ***novel*** design approach eliminates the need for the programmable pass-transistors used in Figure 3.6(a) which contribute to the area overhead in standalone FPGAs. In essence I/O logic in Figure 3.6(a) has been "squeezed" into a single switch block in Figure 3.6(b) due to reuse of "free" routing.

However, the I/O implementation described above places a restriction on the eFPGA architecture. In particular, there can be no overlapping IO connections. For example, as shown in Figure 3.7(a),

in a typical FPGA I/O implementation, external inputs and outputs ***can*** have overlapping programmable connections to the routing track since both will never be enabled at the same time. Therefore, if input "I" is driven on routing track 1 (via pass gate), then output "O" cannot be driven on track 1 (pass gate is disabled to prevent signal contention). In the new design in Figure 3.7(b), such overlaps would ***always*** lead to signal contention because there are no programmable pass gates to use to isolate external nets from each other. To see how this occurs, consider that in Figure 3.7(a), "I" can potentially be driven onto tracks 1 and 2, and so in the equivalent novel implementation in Figure 3.7(b), the output of the tri-state on the topside of the switch on track 1, would be "wired" to the output of the tri-state on the topside of the switch on track 2. Regarding output "O" which can connect to track 0 or 1, the outputs of the respective topside tri-states would be "wired" together.

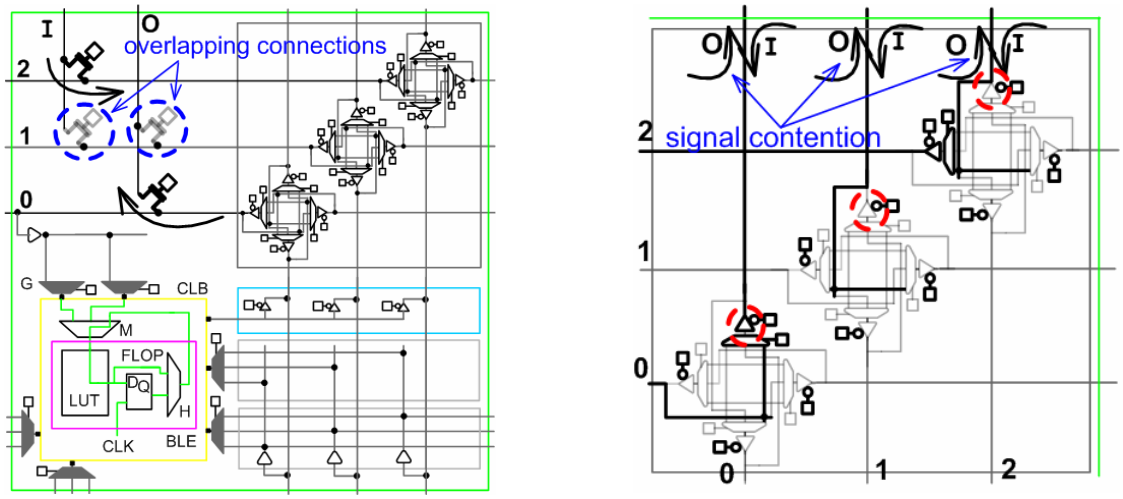


Figure 3.7: A limitation of the *novel* embedded FPGA external I/O architecture

Notice from the above description of the novel I/O implementation in Figure 3.7(b), that "I" and "O" are actually "shorted" due to their ***shared connection*** to the output of the tri-state on the topside of the switch on routing track 1. Basically, the outputs of all tri-states circled in Figure 3.7(b) are shorted together so that signals "I" and "O" are driven on all three tracks. This clearly leads to signal contention. Therefore, only *non-overlapping* connections such as in Figure 3.6 can exist.

3.2.2 Design for Testability

Once a chip with an embedded FPGA is fabricated, it must be tested along with other blocks on the chip. In particular, it should be possible to test the embedded FPGA fabric in isolation, as well as within the context of the chip in which it is embedded. For testing to be successful, the embedded fabric itself must be designed to be testable. Otherwise, it will be difficult to determine whether it functions as expected. Testing an eFPGA core involves testing the logic, routing, and configuration architectures. However, a configuration architecture that is testable is of crucial importance, because if correct eFPGA configuration cannot be assured, then logic and routing architecture tests become meaningless. Therefore, in this work a configuration architecture that is testable has been designed.

A testable design must be both controllable and observable. Control is needed to put the design under test into a desired state, and observable nodes or outputs are needed for comparison with expected results. Therefore, techniques and structures that facilitate controllability and observability during configuration are necessary. Figure 3.8(a) is a high level illustration of the main parts of a design solution. The serial input (SI) is used to load program and address bits. The address shift register (SR) is used to shift in row and column addresses. Row and column address decoders are used to assert word-lines (WL0, WL1, WL2) and column lines (CL0, CL1, CL2) based on target row and column addresses. Clock gating logic in each tile (not shown) is used to turn on or off clocks to configuration flip-flops based on the target row and column addresses. All configuration flip-flops are also linked in a shift chain that can be routed through a multiplexer to an external output (SO).

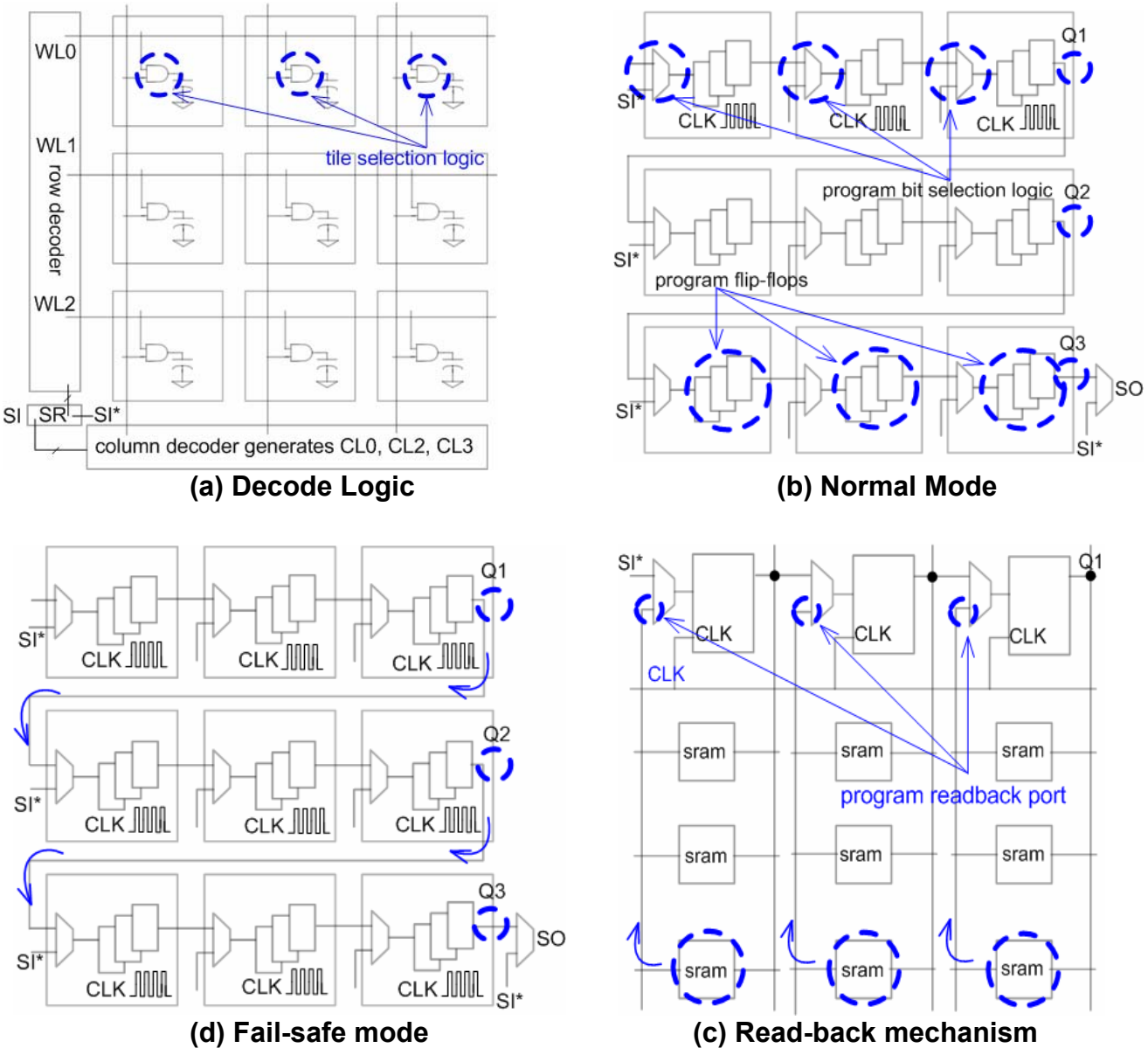


Figure 3.8: An embedded FPGA IP configuration architecture designed for test

The contents of the address shift register can also be routed to SO. Q1, Q2, and Q3 (highlighted in Figures 3.8(b) and 3.8(d)) are serial outputs of the programming shift chain of their respective rows. Finally, a finite state machine controller (not shown for simplicity) is used to arbitrate configuration.

This architecture facilitates control and observation during testing of the configuration architecture.

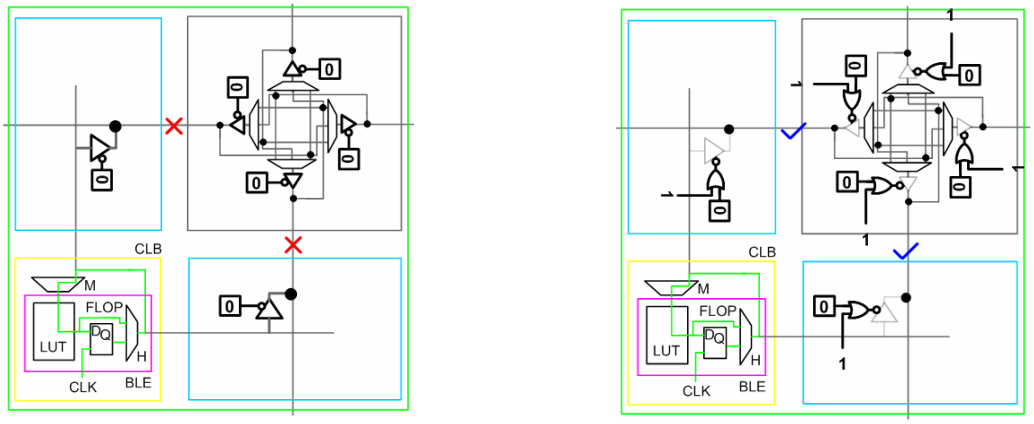
For example, a row decoder activates the word-line [56] of a target row of tiles based on a row address, and tile selection logic in each tile (simplified in Figure 3.8(a)) decides which tiles can be targeted for programming. Figure 3.8(b) shows a single row being targeted for programming (top row); hence, only the clock in that row is activated during programming. Individual tiles can also be targeted for programming. Serial outputs from each row (Q1, Q2, Q3) facilitate observation of the configuration bits for correctness because the program bits for each row can be shifted in, and then shifted out and analyzed. Also, SRAM storage cells could be programmed and then read (see Figure 3.8(c)) to ensure proper functionality. A similar approach called read-back is used in commercial SRAM-programmable logic devices [56][57]. Also, the scan output in the proposed architecture, SO, allows the output from the programming shift chain, or address shift register to be analyzed.

To conclude, there are two modes of operation in the proposed configuration architecture: normal mode and fail-safe mode. In normal mode, the finite state machine is functional and program bits can be loaded into tiles individually or in rows. In fail-safe mode, the finite state machine is not functional (also deactivated), and so a single shift chain is used for programming and all shift registers are activated during configuration (Figure 3.8(d)). The output of this shift chain can be observed at SO in fail-safe mode. In essence, this design is robust enough that limited testing of the configuration architecture can still continue even if the configuration state machine is not working.

3.2.3 SRAM Power up State

The embedded FPGA architectures used in this research have bidirectional routing, which in our case means there can be multiple potential drivers per routing track. In order to prevent contention, these drivers are tri-stated. Furthermore, after configuration, there can be just one driver per active

(driven) routing track. In other words, the SRAMs or flip-flops that are used to enable or disable these tri-state drivers must hold the appropriate values. However, upon power up, and prior to configuration, it is impossible to know what value the configuration cells (flip-flops or SRAMs) will hold initially [56]. It is important to consider this because a situation could arise where all potential drivers of a tri-stated routing track are enabled and driving opposing logic values on a routing track (denoted by "X" in Figure 3.9(b)). This can create high-current short circuit paths from power to ground through the tri-state devices. Such high current paths through transistors can cause irreversible damage and result in a bad chip (unusable routing tracks). Therefore, as a safeguard, it is necessary to disable all tri-state drivers upon power-up. One possible solution is shown in Figure 3.9(b), where a NOR gate is used to disable all the tri-state drivers upon power-up. One input is set to a high value (depends on the polarity of tri-state enable input) so that when the chip is powered up, this signal is asserted and the tri-state enable logic disables all tri-state buffers. The other input from the SRAM/Flip-flop ("0" input in Figure 3.9(b)) has no control over the tri-state buffer.



(a) driver contention **(b) no driver contention**

Figure 3.9: Logic to prevent driver contention in bidirectional routing architecture

Once a circuit is programmed onto an eFPGA fabric, the SRAM value becomes the controlling input to the NOR gate. The other input is set to a logic "0" value and becomes non-controlling.

Chapter 4

Island-Style eFPGA Design with Generic Standard Cells

4.1 The Existing Design flow

Given the eFPGA architecture design specifications described in the preceding chapter, the next step involved implementing the architectures using generic standard cells [01][02][16] and the ASIC flow. Although other programmable logic architectures have previously been implemented in this way [01][02][16], the architectures described in Chapter 3 were selected for a number of reasons.

First, it has been observed in previous work [02] [14] that there are limitations on the size of circuits that can be implemented using existing architectures. Therefore, it would be useful to compare the programmable logic architectures described in the previous chapter with the previous approaches to determine if it has similar limitations or is suitable for implementing circuits of much larger designs.

Second, there is the issue of combinational logic loops in standard-cell-based eFPGAs [02] [17] and its impact on ASIC-flow-based eFPGA design. Combinational logic loops occur when the output of a combinational logic block or gate is also one of its inputs. Combinational loops (in most cases) are an indication of a design error. In an un-programmed eFPGA combinational loops may exist (depending on the architecture) but after configuration combinational loops should no longer exist. Traditional ASIC CAD tools are not equipped to handle such loops (whether due to a designer error or otherwise), and so these loops are "broken" by the ASIC tools, in ways that could affect the accuracy of timing estimation during logic synthesis and optimization. Furthermore, this problem also implies that architectures of the type described in Chapter 3 will be affected because the

potential for combinational loops exists. Architectures described in previous work have either designed such loops out of their architectures [02] with some loss in flexibility, or implemented dual routing networks [17], all of which have resulted in some area penalty. Therefore, the architectures used in this research provide an opportunity to investigate alternative solutions to this problem.

A third reason for using the architectures described in Chapter 3 is their regular and modular structure. This makes them well-suited to optimization experiments that will be carried out as part of this work. For example, it is possible to change the routing architecture of an embedded programmable logic fabric that uses one of these architectures by simply swapping switch blocks.

Fourth, since these architectures are modeled on the island architecture, it is possible to leverage existing FPGA CAD tools in this work. For example, CAD algorithms in the VPR CAD tool suite are well-tuned for island-style architectures since the original version of the tool targeted this kind of architecture. As a result, little or no FPGA CAD tool *re*design is necessary. Furthermore, several recent enhancements to this tool have also targeted island architectures. For example, the power model in [26], and the power-aware algorithms in [28], all target island-style architectures. Although power is not the focus of this research, it would be straightforward to investigate power issues with these eFPGA architectures and the new power-aware FPGA CAD tools. Likewise, a version of VPR using 0.18-micron CMOS process data for area and speed characterization was developed in [49]. Since all of the research data presented here will be based on 0.18-micron CMOS process technology data, it is possible to leverage this new version of the VPR CAD tool in this research.

Fifth, a stated future research goal is to have a collection of different "parent" architectures available so that the most suitable architecture or its derivative, can be selected for a given domain application. The derivative architectures presented in this research work contribute to this effort.

The ASIC design flow used in this work consists of a Front-End flow and a Back-End flow (shaded parts of Figure 4.1). An ASIC flow using generic standard cells for eFPGA design is described next.

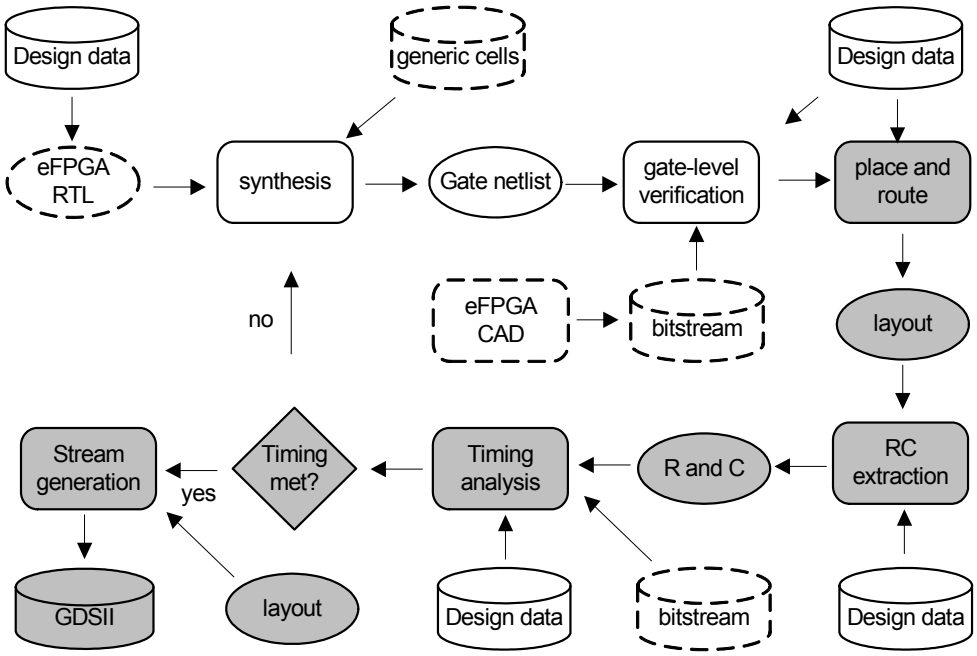


Figure 4.1: Typical flow for *semi-custom* cell, block, and core based ASIC design

4.1.1 Front-End Flow

Referring to the eFPGA ASIC design flow in Figure 4.1 [14], the "Front-End" flow consists of logic synthesis of an RTL description of an eFPGA architecture, and its gate-level functional verification.

The RTL hardware description of the architectures described in the previous chapter preserves many of the user-definable parameters that exist in a generic, clustered island-style FPGA architecture [11][49]. In particular, the cluster size (N), look-up-table input size (K), track width (W),

core dimension (number of regular tiles per column and row), and BLE input and output track connections can be specified as generic statements in the eFPGA RTL description prior to logic synthesis. Consequently, fabrics with valid combinations of these parameters are easily implemented.

The RTL description of a given eFPGA architecture is converted to a netlist of logic gates using logic synthesis tools. These tools map RTL descriptions of eFPGA architectures written in Verilog or VHDL to a gate netlist based on a library of generic standard cells [74]. The end result (gate netlist) depends on the specific synthesis constraints. As a result, in this research, synthesis scripts were parameterized so that constraints are automatically updated based on architecture parameters.

Gate level functional verification occurs after logic synthesis to ensure that the synthesized gate netlist functions as expected. Gate-level functional verification of an eFPGA fabric requires more effort compared to other kinds of logic since programming bits are also needed in order to test the design. Prior to programming, an eFPGA gate netlist does not perform any specific logic function. FPGA CAD tools are used to generate programming bits from place and route data of target or example user circuits. As shown in Figure 4.1, generating eFPGA programming bits requires an FPGA CAD [11] [28] flow. In the flow shown in Figure 4.1, VPR [11][50] is used for circuit placement and routing on a given eFPGA architecture. Final placement and routing files for each circuit are also generated by VPR. After the placement and routing netlists for target user circuits have been generated, specially designed scripts are used to parse these files and generate the programming bits needed for the eFPGA gate netlist configuration. Once the eFPGA gate-level netlist has been programmed and verified, the back-end design phase of the ASIC flow can begin.

4.1.2 Back-End Flow

Back-end ASIC design begins with the placement and routing of a verified gate netlist generated during Front-End design. Back-end design or physical design as it is sometimes called, uses physical geometries of logic cells in an eFPGA gate netlist as well as timing constraints, to achieve the best possible placement of cells within a specified core area. Additional cells not included in the original gate-level netlist may also be included for clock tree optimization as needed. Routing tools connect gate/cell pins with metal lines after placement, based on gate/cell connections specified in a netlist.

After routing, the geometries (width and length) of all wires in the soft eFPGA design are known and so their impact on speed can be estimated using technology data, extracted RC values, and static timing verification tools. As previously reported in [14], static timing verification of programmable logic architectures presents a unique set of challenges depending on the kind of architecture. As shown in Figure 4.1 programming bits generated from FPGA CAD tools are preprocessed into timing exceptions that aid static timing verification. Although not shown in Figure 4.1, gate-level functional verification is also performed using extracted RC values after routing. The Back-end flow proceeds as usual after static timing checks and post-routing functional verification.

4.2 Design Flow Issues and Solutions

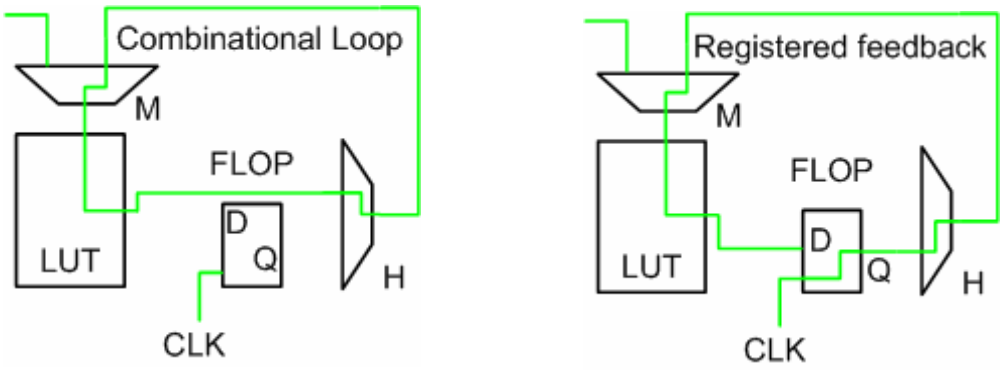
The implementation of eFPGAs using the ASIC flow in Figure 4.1 led to a number of CAD and design problems. In the following sections we describe these issues and attempts to resolve them.

4.2.1 Combinational Loop-back

Combinational loops in digital logic cause difficulties for ASIC tools and should be avoided.

Likewise, in embedded FPGA architectures, these loops should not exist. However, prior to programming, an eFPGA fabric is a network of several potential connections. Therefore, it is possible for combinational loops to exist. If after programming, combinational loops still exist in an eFPGA fabric, there is either a fault in the eFPGA design, or an invalid configuration bit-stream has been loaded, or a bad design that contains such loops has been implemented on the eFPGA fabric.

During logic synthesis of the island-style architecture numerous combinational loops were detected. This was not unexpected since the eFPGA architectures described in the previous chapter contain potential combinational loop-back paths. As shown in Figure 4.2(a), the potential for combinational loops exists because each BLE output (output of H) is also a *potential* input to itself (via the local routing multiplexer, M). In Figure 4.2(a), the output of the LUT (combinational logic) is selected by the output selection multiplexer H and then fed back to the LUT input selection multiplexer M.



(a) combinational loop **(b) sequential loop**

Figure 4.2: Combinational and sequential loops in an island-style eFPGA tile BLE

This multiplexer could *potentially* drive this input to the LUT output (the original "starting" point). This is a combinational loop path because a series of combinational logic paths form a loop with no distinct starting and endpoint. In contrast, the path in Figure 4.2(b) is not a combinational loop path. It is a sequential loop path with distinct starting and end points because no timing arc or path exists

from the input D to output Q of the flip-flop that separates them. Unlike the combinational loop of Figure 4.2 (a) this loop has a distinct starting point, Q, and endpoint, D as shown in Figure 4.2(b).

Furthermore, in switch elements, the existence of signal paths from the output of a driver to its input as shown in Figures 4.3 is also another *potential* source of combinational loops, because similar to the example described above, starting and end points for these timing paths are indistinguishable. Therefore, such paths are also reported as combinational loops by the ASIC design tools during logic synthesis, because the eFPGA fabric has not been programmed at this stage of the flow.

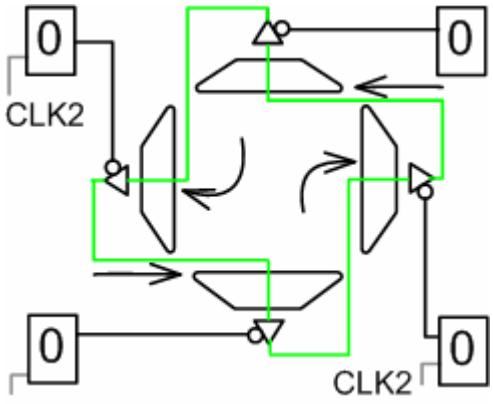
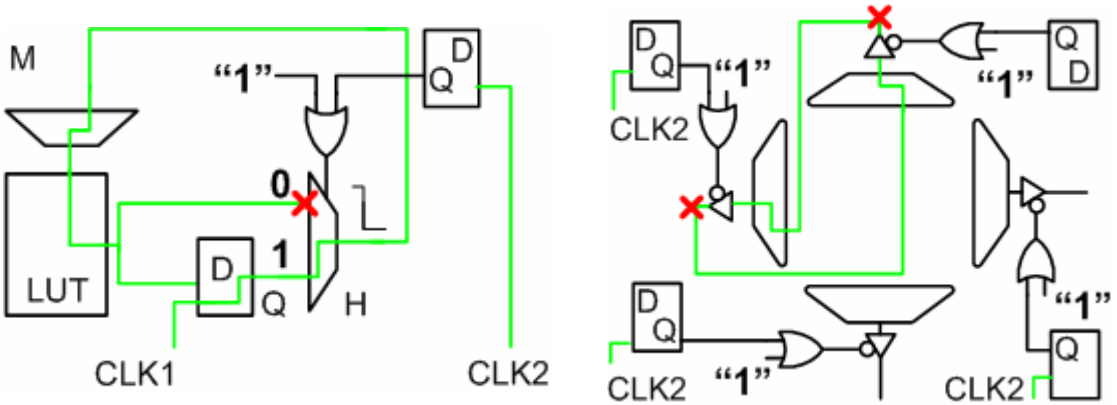


Figure 4.3: Example of combinational loops path in a multiplexed routing switch

The existence of combinational loops during eFPGA logic synthesis is a problem because synthesis tools use automatic loop breaking techniques during logic optimization. This happens because certain tools require combinational-loop-free logic to operate properly. Unfortunately, somewhat arbitrary rules are used to break the loops. As a result, timing paths are created with new and sometimes optimistic timing constraints [75]. Therefore, logic optimization steps, such as gate sizing could be affected (e.g., a gate with a weaker drive strength could be selected, which results in slower logic). Overly pessimistic constraints could also lead to larger gate sizes and increases in area.

In order to prevent automatic loop breaking during logic synthesis and eliminate its possible negative side-effects on timing optimization, path exceptions can be set to exclude potential combinational loop paths from consideration during optimization. For example, setting the select input of the two-input multiplexer, H, in Figure 4.4(a) to a high logic state during synthesis, ensures that only the registered LUT output is assumed to be the valid multiplexer input during logic optimization. This is achieved by forcing one of the inputs of the OR-gate (see Figure 4.4(a)) to a high logic state. In essence, a sequential loop like that shown in Figure 4.2(b) is created. A similar approach is used to eliminate automatic loop breaking in switch elements as shown in Figure 4.4(b).



(a) eFPGA tile BLE **(b) eFPGA tile switch**

Figure 4.4: Technique to avoid loop breaking during synthesis and verification

Of greater significance is the effect that combinational loops have on soft eFPGA gate-level verification. Before gate-level functional simulation can begin, the embedded FPGA must first be programmed with bits that implement the desired logic function. However, during programming, the logic value of the configuration flip-flops changes several times as the configuration bits are shifted into the fabric (configuration memory in standard-cell-based eFPGAs is implemented as a shift chain of flip-flops [14]). Thus, it is almost certain that the fabric will be forced into a number of invalid states before programming is complete. It was observed during programming that some of these invalid states created combinational loops in the eFPGA similar to those described earlier. ASIC logic simulators cannot resolve such loops and will terminate if combinational loops exists.

When a gate netlist functional simulation terminates due to combinational loops, even before the eFPGA fabric under test has been programmed, the fabric cannot be verified. This implies that eFPGA architectures of the kind described in the previous chapter cannot be *reliably* implemented in the ASIC design flow. This is not desirable, because all designs must be verifiable prior to committing them to silicon. It is also not advantageous to exclude this class of eFPGA since most of the high performance FPGA architectures in use today incorporate some of the same features that cause problems for the ASIC verification tools. Furthermore, this problem could restrict designers to unidirectional routing architectures of the kind described in previous work [02] [15] [16] [17].

A technique similar to the one used during logic synthesis (see Figure 4.4) is used to prevent the occurrence of combinational loops during gate-level functional verification. However, in this case, an actual port or pin must be added to the eFPGA fabric so that its value can be set appropriately. This global input controls the NOR-gate input not driven by a flip-flop in Figure 4.4. As a result, it is possible to mask flip-flop transitions that cause combinational loop paths during programming.

To summarize, the implications of this rather simple solution to the combinational loop problem are quite significant. It guarantees that ***all*** programmable logic architectures with potential combinational loop-back paths can now be implemented using the standard ASIC design tools, and with no impact whatsoever on the choice of routing architectures. For example, there should no longer be any need for a dual interconnect tree (costly in area) described in [15] to cope with the existence of potential feedback paths. Furthermore, it should now be possible to implement a more optimal central interconnect switch matrix for the product-term architecture presented in [15] [16].

4.2.2 Architecture Discrepancies

As shown in Figures 3.6 and 3.7, the I/O design for an embedded FPGA is different from a standalone FPGA. However, the VPR CAD software that was used in this research assumes a standalone FPGA chip. Therefore, there is an I/O mismatch between the eFPGA architectures used in this work and the architecture that VPR supports. A normal solution to this problem would be to change the CAD software for VPR and adapt it to the different architectures we have implemented. However, a different solution was developed that involved "mapping" between the architecture assumed in VPR, and the architectures that were implemented. For example, because the original VPR software routes external nets to I/O pads via programmable connections as shown in Figure 3.6, the problem was to find the equivalent routing solution in our eFPGA architectures because edge switch blocks are used to implement our I/O. All such mappings were incorporated in configuration-bit-stream-generation programs that were implemented as part of this research work.

4.2.3 Static Timing Exceptions

Static timing verification is an essential part of the ASIC design flow. However, as pointed out earlier, programmable logic architectures are unlike other kinds of ASIC designs. For example, before static timing verification of an eFPGA can commence, the eFPGA netlist must first be "programmed". This cannot be done in the same way as during functional verification because the static timing verifier is not a simulation tool and does not create a hardware model of the logic design like a functional simulator can. However, it is possible to "program" an eFPGA gate netlist using built-in commands for enabling and disabling timing paths during static timing verification. This is essentially what happens when an eFPGA core or fabric is programmed with a bit-stream.

In collaboration with [14] a technique was devised to translate programming bits into explicit static timing exceptions. However, the architectures used in this work have more complex routing architectures, and it is nontrivial to incorporate these timing exceptions in the timing flow. Another solution was to update the timing of the netlist after less explicit timing exceptions have been applied, so that the timing database for the netlist is current. This technique was adequate for smaller benchmarks but not for large benchmarks. Instead, for these larger circuits (>1000 LUTs), the *strategic* use of the more explicit approach developed with [14] and described therein, completely eliminated false timing from timing reports. More specifically, applying stringent exceptions to the input and output connection blocks alone was sufficient. To confirm this, path delay reports from the ASIC tools were compared with those from VPR to ensure the paths were true paths and not false paths. For example, in Figure 4.5 the VPR-calculated critical path starting at "1" and ending at "6", ***matched exactly*** the critical timing path reported in the Primetime® static timing analysis tool.

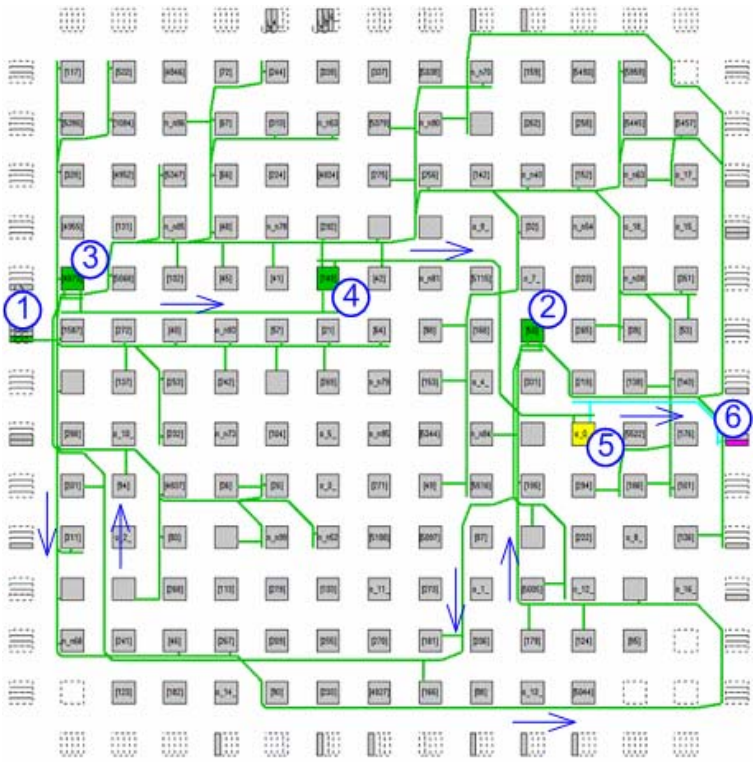
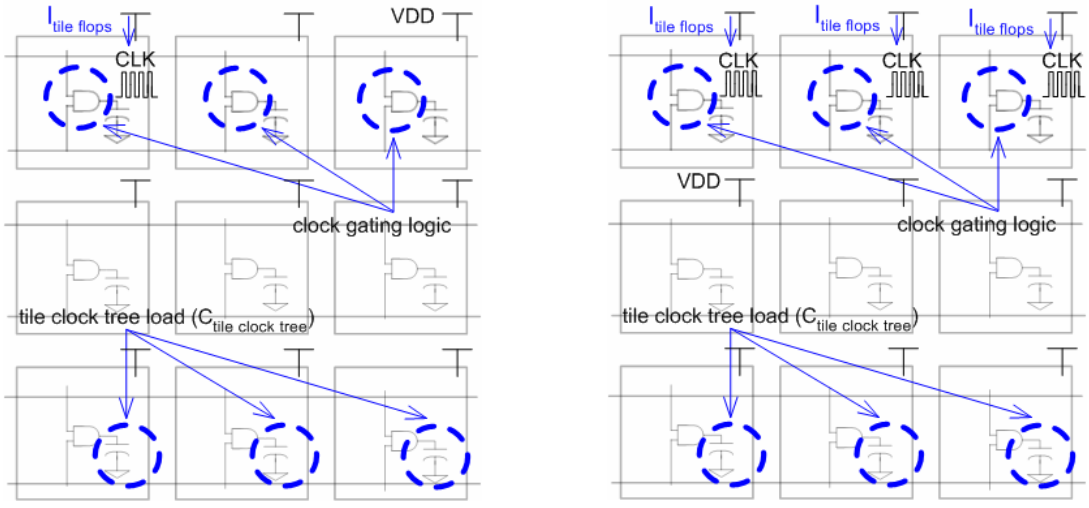


Figure 4.5: VPR critical timing path trace for "golden-20" benchmark circuit apex4

4.2.4 Configuration Power

The configuration architecture of an eFPGA is used to load and hold the configuration bits that implement a particular user function. As mentioned in Section 2.3.2, the use of flip-flops rather than SRAMs for program storage is very costly, because flip-flops are larger and several are needed.

Furthermore, the large number of flip-flops in eFPGAs also requires a large network of configuration-related signals that are spread throughout the eFPGA. For example, it is expected that the clock network for the configuration flip-flops will be quite extensive, and the results in [14] suggest that this is the case. This has implications for power dissipation in the clock network during programming. In addition, the programming scheme used in an architecture implementation also affects power dissipation. For example, in [14], a single shift chain of flips-flops was used to implement the eFPGA programming architecture. This approach forces all flip-flops in the chain to be clocked for the entire duration of the configuration phase. This means that all the flip-flops, and their entire clock tree network, could potentially draw large currents from off-chip power supplies.



(a) single tile programmed **(b) entire row programmed**

Figure 4.6: Proposed configuration scheme targeting a single tile and row of tiles

In order to make soft eFPGA configuration more power efficient, two schemes were proposed that are somewhat similar to the approach used in commercial SRAM-programmable FPGAs [56] [57] (see also Figure 3.8(d)). In the first scheme shown in Figure 4.6(a), a single tile in the soft eFPGA can be targeted for programming so that only the clock network and configuration flip-flops for that tile are activated during configuration (see Figure 4.6(a)). In the second scheme shown in Figure 4.6(b), a single row of the embedded FPGA can be programmed. Again, this means that just the clock network and configuration flip-flops for the target row are activated during programming. Individual rows or tiles are targeted for programming until the entire eFPGA has been programmed. Clearly, the first scheme is more power efficient since only the capacitive loading on a tile clock tree network and the total current (data dependent) drawn by all the flip-flop in the active tile contribute to power. However, as the number of flip-flops targeted scales upwards from a single row to one "gobal" program shift chain, power and energy increase. The programming time is roughly constant.

Another disadvantage of using flip-flops for eFPGA configuration storage is the likelihood of glitch power dissipation. Glitches are rapid transitions that occur at the outputs of combinational logic in response to changing inputs. Because flip-flops in this configuration scheme are part of a shift chain, there is a good chance that the state of each flip-flop will toggle several times before the end of configuration of a single eFPGA tile or row. Some of these flip-flops are inputs to combinational logic and as such will cause transitions to occur at combinational logic outputs during configuration.

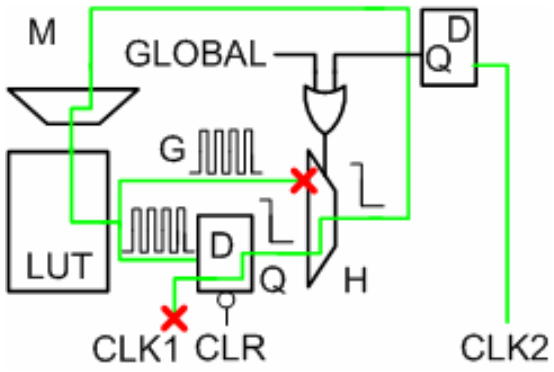


Figure 4.7: Proposed scheme for glitch isolation during soft eFPGA configuration

Figure 4.7 shows a proposed scheme to minimize the propagation of glitches to global routing wires (primary source of dynamic power dissipation) and other parts of the logic architecture during configuration. A global masking input labeled "G" in Figure 4.7 is used to control the OR-gate output, so that the registered (non-toggling) output of a LUT is selected by the output multiplexer "H" during the configuration phase. The transition that occurs on the registered output (shown as a high-to-low transition in Figure 4.7) prior to configuration is due to a global reset (CLR) on the LUT-flop. The registered LUT output remains unchanged until programming is complete. The embedded FPGA system clock (CLK1) is also inactive during the configuration phase to prevent glitch propagation through the LUT-flop to tristate buffer inputs and other parts of the logic architecture. CLK2 is the programming clock. The "Xs" in Figure 4.7 indicate disabled paths.

Without the above safeguard, glitches could be propagated through output drivers and onto routing tracks and contribute to dynamic power dissipation. Ultimately, the magnitude of glitch power dissipation is a function of the capacitance of routing tracks (function of the track wire length), the width, W, of the routing channels in a device, the number of connections from a given output to a routing channel, the total number of routing tracks in a device (related to core size), and how often

it is reprogrammed during operation. It is also important to notice that the glitch isolation scheme shown in Figure 4.7 was also used to prevent combinational loops during functional verification.

Finally, a case study of an example design from [14][15][19] was used to successfully verify the island-style fabric implementation from synthesis through to static timing signoff as shown in the ASIC flow of Figure 4.1. Further details of this implementation are provided in Appendix A.1.

4.3 Design Results

Successful verification of the eFPGA RTL implementation meant that valid area and speed results for architecture derivatives presented in Chapter 3 could be obtained. It is useful to note that the current design supports *only* architectures that have routing tracks that span one tile. In other words, each switch block is *directly* connected to *only* switch blocks in the nearest tiles. These tracks are referred to as *single segment length* tracks [11]. Likewise, tracks spanning L tiles are segment L tracks.

Figure 4.8 shows a plot of critical path delays for 18 MCNC benchmarks. All delays are normalized relative to the delay of an equivalent tri-buffered switch architecture. Results for 3 of 4 architectures described previously in Chapter 3 are presented. Based on the results, the eFPGA cores with tri-buffered switches are on average 24% faster than those implemented with the original multiplexed switch described in [20] [49]. In terms of area, the cores implemented with tristate-buffer-based switches (tri-buff) are on average about 3.3% larger than multiplexer based switches (see Figure 4.9).

Based on these results, the tri-buffered switches have the smallest critical path delay overhead and a relatively small area overhead when compared to the multiplexer based switch presented in [49]. In an effort to reduce the delay overhead of the original multiplexer-based switch, a new switch

element referred to in Section 3.1.1. as "Improved multiplexed switch 1" was designed. This design was contrived based on an important observation that is evident in Figure 4.5. In Figure 4.5, nets on the critical path span several CLBs *vertically* and *horizontally* without changing direction. For example, the path from "node 1" to "node 2" has long vertical and horizontal sections. Therefore, it was anticipated that delay overhead could be reduced by creating fast horizontal and vertical routes through the original multiplexer-based switch. Fast connections with *single buffer delay* overheads were incorporated in the original multiplexer switch design as shown in Figure 3.3(b). This approach is somewhat analogous to including long routing wires [11] [20] in programmable logic architectures.

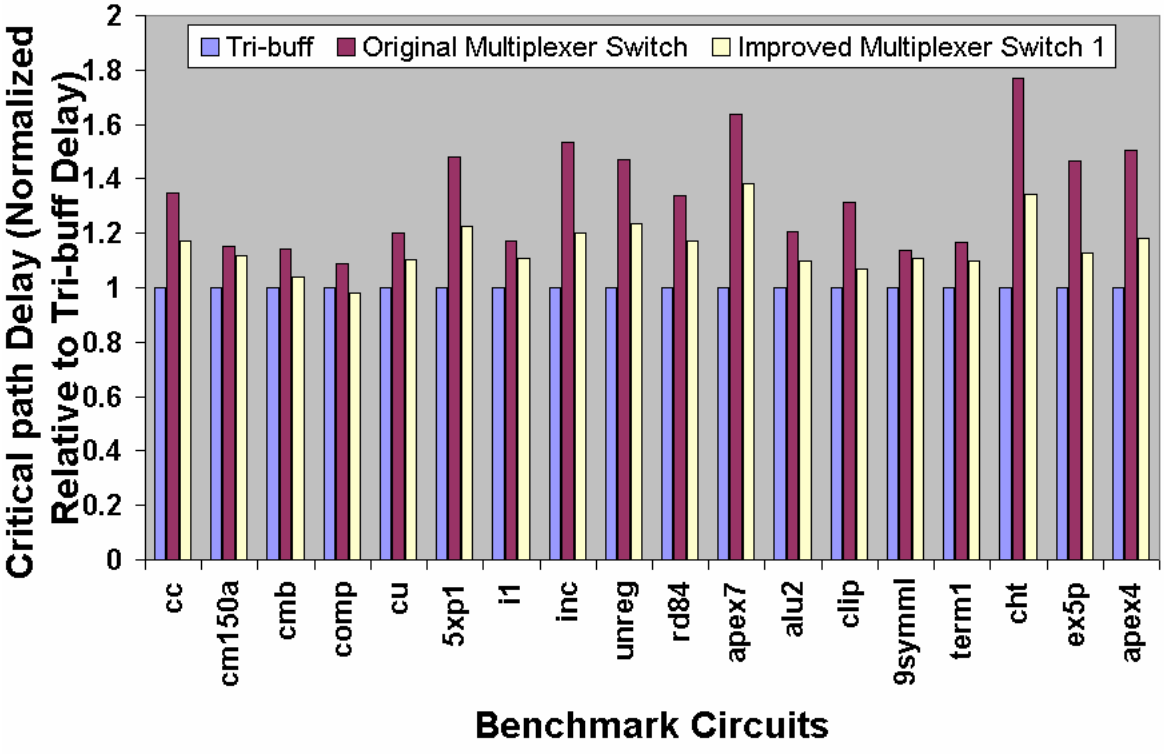


Figure 4.8: Critical path delay comparison of three *soft* island-style architectures

Measured critical path delays showed that "improved multiplexer switch 1" is on average ***11% faster than the original multiplexed switch and only 2.6% larger***. The tri-state-based switch is still faster by about 13% and only 0.7% larger. From the plots in Figure 4.9 and data in Appendix A, fabrics with larger dimensions provided the biggest speed gains. For example, two large circuits from

the MCNC benchmark suite, ***apex4 and ex5p, were sped up by 23% and 22% respectively***. The larger circuits show the biggest gains because critical nets follow longer vertical and horizontal paths.

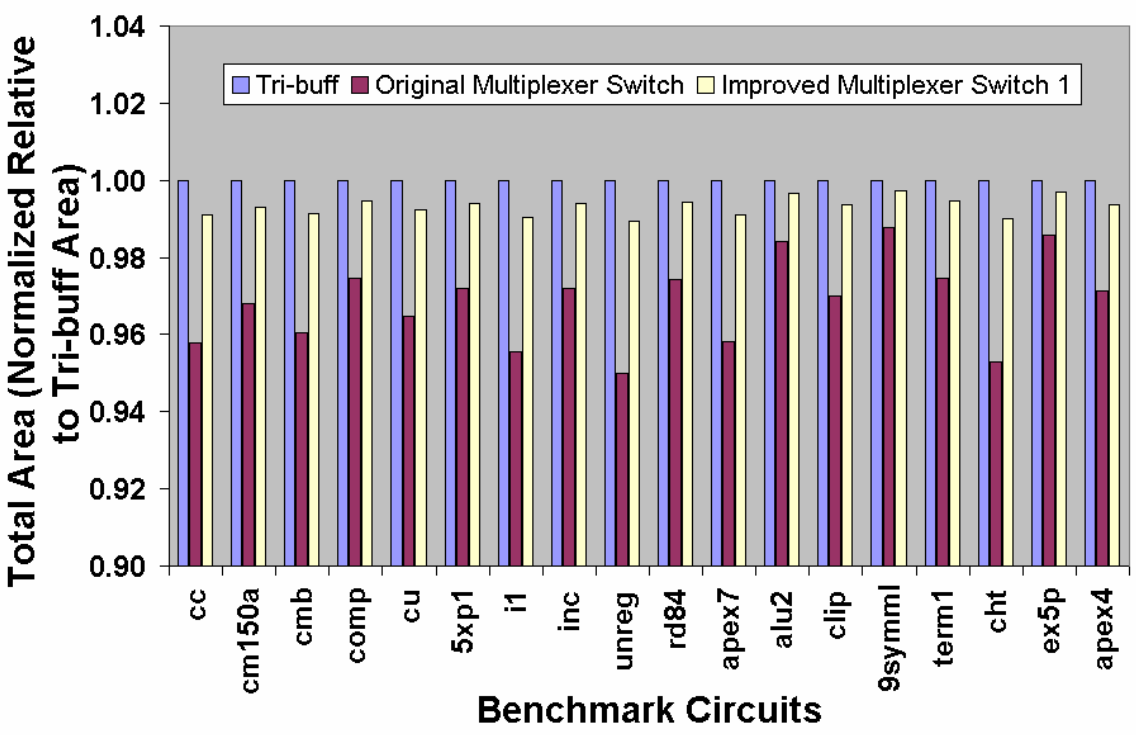


Figure 4.9: Overall core area comparison of three soft island-style architectures

It is *very* important to note that the area and delay results presented above are *sensitive* to the composition of the standard cell library used in the ASIC design flow. For example, architectures using multiplexer-based switches and implemented with CMOS logic gates in generic standard cell libraries are at a disadvantage because a significant area and delay overhead is incurred relative to architectures that have multiplexers implemented with pass transistor logic [20]. On the other hand, architectures with tri-buffered switches (see Figure 3.1) are at an advantage when compared to multiplexer-based switches, because tri-state buffers are available as well-optimized cells in most commercial standard cell libraries while pass-transistor-based multiplexers are not available.

For the above reasons, derivative architectures using the novel "improved multiplexer switch 2" and "original multiplexer switch" [20] [49] (see Figures 3.2, 3.4) are best implemented using pass transistor logic in order to *fairly* compare area and delay overheads relative to the tri-buffered switch architecture. For example, in the original multiplexer switch design, it has been observed that the multiplexer delay overhead dominates the tri-state buffer delay overhead by a factor of about 2.5. This imbalance is due in large part to the fact that the multiplexers are constructed from logic gates. Furthermore, it is expected that the area superiority of multiplexer-based switches can be improved.

Given these observations, architectures based on the original multiplexer switch and "improved multiplexer switch 2" offer the best chance to further close the speed gap *and* improve area relative to the tri-buffered switch. This is because "improved multiplexer switch 1" has eight tristate buffers per switch, compared to four for "original multiplexer switch" and "improved multiplexer switch 2".

Another example of the sensitivity of the ASIC-flow-based eFPGA design approach is evident in results in [02] and [14]. For example, in [14], although the same architecture was implemented for the same benchmarks circuits used in [02], the final area results differ quite significantly, and in some cases by over a factor of 2. The reason for this may be due to different emphasis during synthesis. For example, a *delay constrained* logic synthesis usually results in larger area. Therefore, in this work, similar constraints were applied to all circuits during logic synthesis to avoid any inconsistencies.

In addition to the architecture comparisons described above, the area overhead of the standard cell approach relative to full-custom design was investigated. Since the architectures implemented here are identical to the architecture assumed in the VPR tool, it was possible to leverage the area model in VPR for this purpose. The VPR tool reports area based on the number of minimum sized

transistors in a programmable logic device, and this can be easily converted to area in sq. microns. For this comparison, the derivative architecture based on the original multiplexing switch was used because the version of VPR that has been characterized for 180nm technology is based on this architecture. This eliminates any bias that may result from comparing slightly different architectures.

From the results in Table 4.1, the standard cell approach has an average area overhead of about 6.8X relative to a full-custom design. Another important observation from Table 4.1 is the difference between the estimated area and the actual area obtained for the real design. This estimate is based on the area model in Appendix B that was developed as part of this research. Based on results in Table 4.1 the estimated area is on average within +5.8/-3.8% of the actual synthesis area. This implies that the area model is quite accurate. This model was useful for predicting the core area before ASIC implementation, so that only cores with acceptable area (and delay) overheads were implemented.

Table 4.1: *Area* overhead of "soft" eFPGA design relative to full-custom approach

Circuit	Estimated area (um^2)	Actual area (um^2)	% Area difference	VPR area (um^2)	Soft vs. VPR area Ratio
Cc	419158	451298	7.7	62741.02	7.2
cm150a	221086	240130	8.6	33038.89	7.3
Cmb	248244	257889	3.9	36705.79	7.0
Comp	443320	475532	7.3	69590.83	6.8
Cu	306440	290039	-5.4	39844.19	7.3
5xp1	370696	366387	-1.2	51354.28	7.1
I1	277208	267236	-3.6	40112.47	6.7
Inc	370696	366387	-1.2	51354.28	7.1
unreg	569176	604075	6.1	88647.52	6.8
rd84	2506527	2401023	-4.2	362839.93	6.6
apex7	1483108	1455608	-1.9	239553.32	6.1
alu2	2744876	2936981	7.0	437718.72	6.7
clip	1955266	2049542	4.8	321019.09	6.4
9symml	929304	848430	-8.7	120710.88	7.0
term1	1216817	1233960	1.4	181702.80	6.8
cht	1211974	1289779	6.4	196652.12	6.6
ex5p	23128464	22168164	-4.2	2832898.74	7.8
apex4	36441909	38133916	4.6	4567776.75	8.4

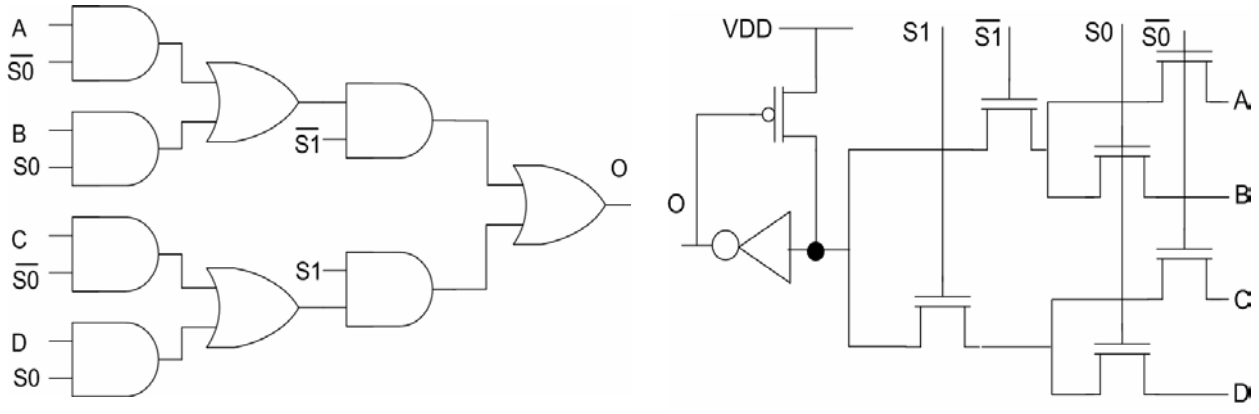
Table 4.2 shows results obtained for critical path delay. The second column shows delay results after placement, routing and parasitic extraction. The third column shows delay results prior to placement (without metal wire delay). Based on these results, it was estimated that on average, wire delay is roughly 10.3% of the overall critical path delay. This result is not surprising since single segment length wires are used in all architectures considered. However, it is expected, that in architectures that have long wires, the delay overhead due to wires would be much greater. It is also interesting to draw parallels between this result and the results obtained in [14]. In [14], a flow was developed to minimize the wire delay overhead on critical paths by rerouting nets along paths with less wire delay. The experiment in [14] yielded modest improvements, and the results obtained here show that a low wire delay overhead is the likely reason. Finally, Table 4.2 also shows that the critical path delay overhead of a standard cell eFPGA design relative to a full-custom equivalent is about 1.8X.

Table 4.2: *Delay* overhead of "soft" eFPGA design relative to a full-custom design

Circuit	Post layout delay (ns)	Pre layout delay (ns)	% Wire overhead	VPR path delay (ns)	Soft vs. VPR Delay Ratio
cc	10.3	9.6	-7.8	5.81	1.8
cm150a	10.3	9.6	-6.7	5.94	1.7
cmb	9.7	8.8	-9.6	5.45	1.8
comp	17.7	16.0	-9.8	9.79	1.8
cu	9.9	8.8	-11.5	4.68	2.1
5xp1	7.6	7.0	-8.0	4.85	1.6
I1	11.3	9.9	-12.6	6.09	1.9
inc	8.3	7.5	-9.2	4.84	1.7
unreg	10.1	9.3	-7.9	6.04	1.7
rd84	28.0	24.4	-12.7	15.2	1.8
apex7	18.4	17.1	-6.8	10.8	1.7
alu2	28.7	23.6	-17.7	15.6	1.8
clip	23.7	20.6	-13.2	13.7	1.7
9symml	14.5	12.2	-16.0	7.73	1.9
term1	18.6	15.6	-16.0	8.71	2.1
cht	9.6	8.9	-7.3	6.65	1.4
ex5p	40.2	37.3	-8.0	20.7	1.9
apex4	56.7	51.1	-11.0	36.9	1.5

The 18 benchmarks used in the above experiments were selected based on availability and size. Further, MCNC benchmarks have historically been used for FPGA research. In addition, it was necessary to use benchmarks with sizes that span a wide range (32 to > 1000 4-LUTS in this case).

Given the area and delay overhead results reported in Tables 4.1 and 4.2, it is necessary to find ways to reduce the overhead. The use of custom cell libraries to improve ASIC designs is well documented in [48], therefore, it is expected that the same techniques might be applicable in this case. However, before embarking on designing custom cells, it was necessary to first evaluate the sources of inefficiency in the existing approach. As previously reported in [14] it is expected that the biggest sources of area inefficiency comes from the use of standard flip-flops for configuration memory and discrete CMOS logic gates like NAND-gates and NOR-gates for multiplexer implementation. For example, as Figure 4.10 shows, a pass transistor network implementation of a 4:1 multiplexer contains a total of 13 transistors including inverting gates for complimentary select signals ($\overline{S0}$ and $\overline{S1}$). On the other hand a CMOS logic implementation with discrete gates has over 40 transistors in total. This translates to an area overhead of about 3X for a 4:1 multiplexer. Further, it is also expected that the multiplexer area will grow *exponentially* with the number of select inputs.



(a) 4:1 multiplexer in CMOS logic **(b) 4:1 multiplexer in pass gate logic**

Figure 4.10: Comparison of CMOS logic and pass transistor based multiplexers

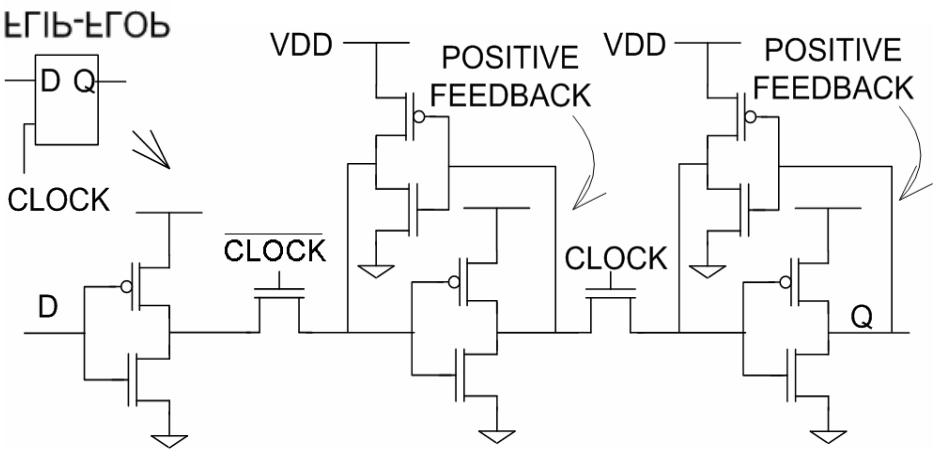
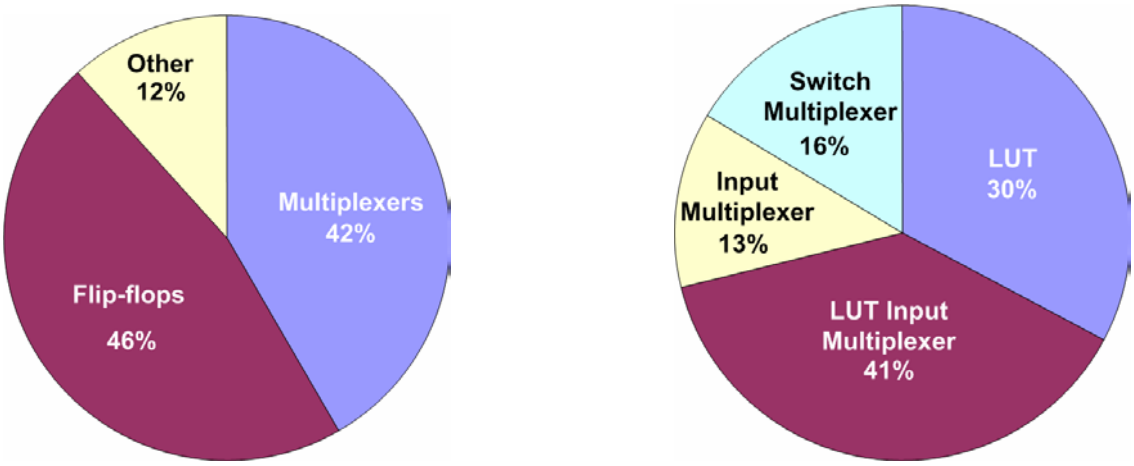


Figure 4.11: transistor level illustration of a simple single-edge-triggered flip-flop

Similarly, the area overhead of flip-flops is due to the fact that flip flops contain more transistors and therefore occupy more area compared to 6T (six transistor) SRAM cells which are typically used for program memory. The flip-flop design in Figure 4.11 comprises a total of 12 transistors. Therefore an SRAM with six transistors (see Figure 5.2) is approximately *half* the size of a flip-flop

The pie charts in Figure 4.12 show the area overhead distribution in the "soft" implementation of the island-style architecture described earlier. From the chart in Figure 4.12(a), it is clear that configuration memory (flip-flops) and multiplexers are the biggest contributors to area overhead. "Other" in Figure 4.12(a) includes the BLE output tristate buffers, and shared buffers used to drive the inputs of the connection block multiplexers. Figure 4.12(b) shows that the biggest contributors to multiplexing logic area are the LUTs and the LUT input selection multiplexers ("M" in Figure 2.6(b)). Since generic standard cell multiplexers and flip-flops make up 88% of the total eFPGA island-style tile area, and are also not efficiently built using the current approach, they are prime candidates for custom standard cell substitution. Moreover, it is expected that replacing generic standard cell multiplexers will have a significant impact on delay since multiplexers make up the logic and routing architecture. Furthermore, the results in Table 4.2 have shown that gate delay

comprises about 90% of the overall delay. Although the current research work is not focused on power optimization, it is expected that power saving will occur as a side-effect of logic optimization.



(a) tile area distribution **(b) multiplexer area distribution**

Figure 4.12: pie charts showing *area* distribution in a *soft island-style eFPGA* tile

Finally, in the next thesis chapter, circuit-level design techniques that could improve area and delay for standard-cell-based eFPGA fabrics are explored based on the observations in Figure 4.12.

Chapter 5

Island-Style eFPGA Design with Custom Standard Cells

5.1 An Improved Design flow

Hard and soft eFPGA fabric design approaches have several advantages, but also have significant disadvantages. For example, a hard eFPGA has superior area, speed, and power characteristics compared to an equivalent soft eFPGA. However, hard eFPGAs are inflexible and can be very inefficient depending on logic requirements. Soft eFPGAs on the other hand are flexible and offer the best chance to match an eFPGA to an application, but have large *area*, *delay*, and *power* overheads. In particular, flip-flops used for configuration storage, and the multiplexers used in routing and LUTs, account for a significant proportion of the area overhead. The reason for this is simple: these two circuits are the most pervasive in programmable architectures, but generic standard cell libraries do not include SRAM cells needed for configuration storage, or large fan-in multiplexers needed for parts of the routing and logic architectures. On the other hand, in hard eFPGAs, full-custom techniques are used to design much smaller configuration SRAMs and fast, wide fan-in multiplexers.

Given the inefficiencies that exist in the hard and soft design approaches, a compromise approach that combines the best features of both approaches is needed. Such an approach should retain the soft IP advantage of configurable architectures, that is, it must remain flexible but at the same time, incorporate some full-custom design techniques to minimize area, delay and power overheads. However, any improvements should fit within the ASIC digital design flow in order to maintain the ease of use characteristics of the soft approach. Given the fact that configurable RTL descriptions of eFPGA architectures have been designed and implemented in Chapter 4, this requirement has been

achieved. Therefore, the next stage of this research was to minimize area, delay, and power overheads using full-custom design techniques, in order to derive the benefits of the hard approach.

Previous work in [46] [48] [53] has demonstrated that generic standard-cell-based ASIC designs can be improved using customized standard cells. These are called *tactical cells* in [53]. Most of this previous work was aimed at nonprogrammable ASICs, but in [01] [68] some of the same techniques have been used to design standard-cell-based programmable architectures. However, this work did not take into account the significant area overhead due to wide-fan-in multiplexing logic (see Figure 4.12) and the potential savings that come from optimizing them (see Figure 4.12). Also, work in [01] showed area improvements of 16-19% for a data-path architecture but it is anticipated that larger gains might be possible in an island-style architecture [11]. This could be because the architecture in [01] is not multiplexer-intensive and has fewer circuits that benefit from the use of custom cells.

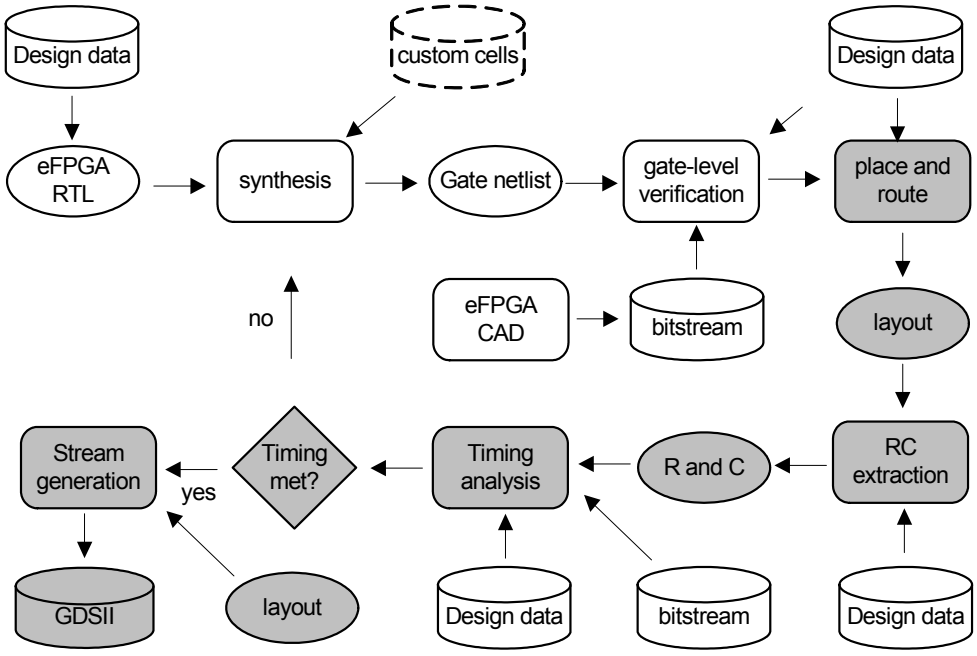


Figure 5.1: an enhanced ASIC design flow for eFPGA design and implementation

A modified ASIC digital design flow using custom standard cell libraries is shown in Figure 5.1. The main difference is that customized standard cell data would be used for embedded programmable logic generation, rather than generic standard cell library data. The details of the design and implementation of these customized standard cells is described in the next section.

5.2 Design of Custom Cells

Given the area distribution results obtained in the previous chapter, this section describes the detailed design and implementation of SRAM and multiplexing logic cells for use in eFPGA cores.

5.2.1 SRAM Cell Circuit Design

The SRAM circuit design used in this work is based on the 6-T (six transistor) cell [25] [56] [66]. However, the SRAM cell design for FPGA circuits is different from conventional SRAMs in one important regard, namely, FPGA SRAM cells have dedicated read and write "ports" (one for writing and another for reading). In other words, the cell is unidirectional. As a result, one of the inverters in the cell acts as a "charge keeper" (via positive feedback) and keeps the value written to the SRAM, "bit", from being lost, while the other inverter acts as a "sensor" to quickly detect a new value being written to the SRAM. Consider the 6T cell of Figure 5.2: because the "sense" inverter is driven by an NMOS pass transistor that passes a weak logic "1" value, the sense inverter was skewed (larger NMOS) to respond faster to a rising input (i.e., when writing a logic "1" value to memory or setting node "bit" to logic "1"). A slow response to a weak logic "1" input is not desirable because the programming time would increase. Moreover, the sense inverter is sized larger since it also is used as an input driver in the function table of LUTs. In this case, the LUT input is connected to "bitb". Figure 5.2 shows transistor sizes obtained from SPICE simulations. All channel lengths are 0.18um.

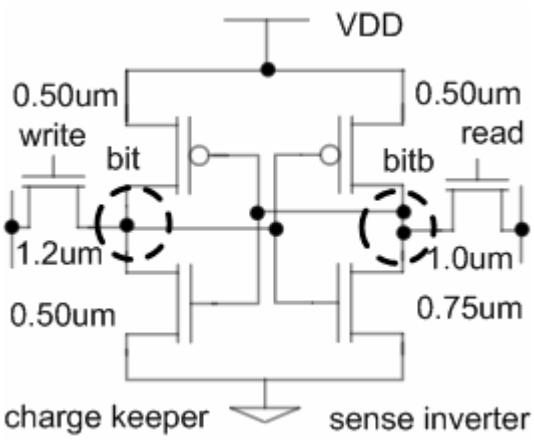
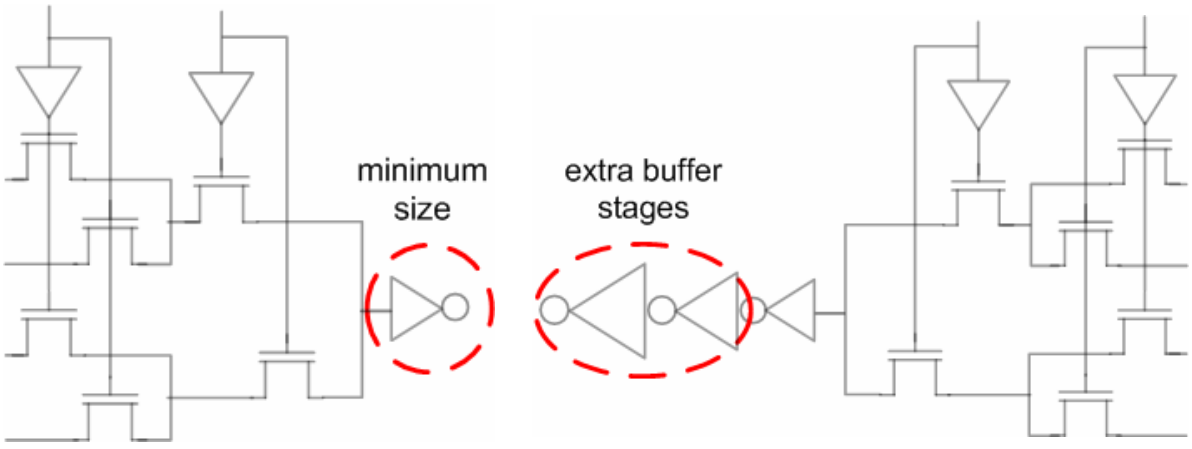


Figure 5.2: SRAM transistor sizing for embedded FPGA configuration memory

5.2.2 Multiplexer Circuit Design

To minimize area overhead, pass transistor logic was used for multiplexer circuit design. All pass transistors were minimum size because this gives the best area and speed tradeoffs [66]. As a result, circuit design work was focused on issues like buffer sizing for pass tree select lines, output driver sizing, and buffer insertion (repeaters) within the pass tree to improve speed in larger tree networks.



(a) single output driver **(b) multiple drivers**

Figure 5.3: multiplexer with minimum size output buffer and extra buffer stages

The output driver inside each multiplexer circuit is a minimum sized inverter (Figure 5.3(a)) since this presents the smallest load for the pass transistor network, and higher output drive strength can be achieved with extra buffer stages as needed (see Figure 5.3(b)). In addition, because NMOS pass transistor gates transmit a weak high logic value ($V_{DD} - V_T$), a level restorer [11] [49] [66] was placed at the output. This ensures that the input to the output driver is equal to the supply voltage, and prevents static power dissipation in the output driver. Since, eFPGAs typically contain a significantly large proportion of multiplexers, it is important to eliminate static power dissipation in these drivers.

Furthermore, the level-restorer must be of the appropriate size because in cases where its transistor $\frac{W}{L}$ ratio is too large, it can "overpower" the pass tree driver, and the output buffer never switches.

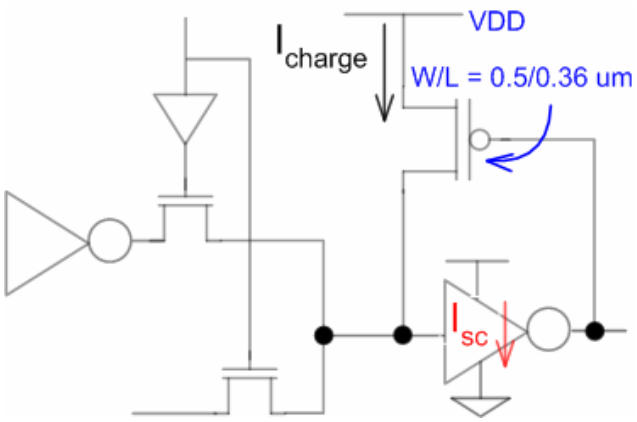


Figure 5.4: a level restoring circuit for pass-transistor-based multiplexing logic

Buffer sizing of the select line drivers was done using Logical Effort techniques as described in [25] [48]. For each size of multiplexing logic, the number of buffer (inverter) stages that gave the best speed and area tradeoff was computed. As a result, the select lines for some multiplexer sizes were inverting, while others were non-inverting. Notwithstanding this fact, an important aspect of the design was to recognize the strong coupling between LUT input multiplexers and the LUTs

themselves. In particular, optimization of each LUT select-signal-path must include the output buffer of the LUT input multiplexer (as shown in Figure 5.5) in order to achieve best results.

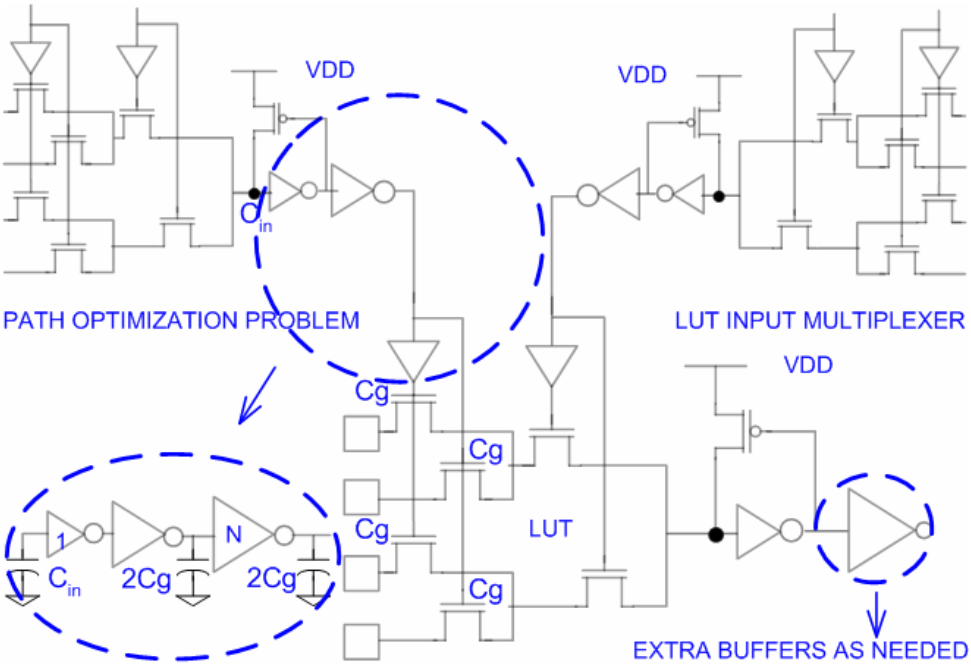
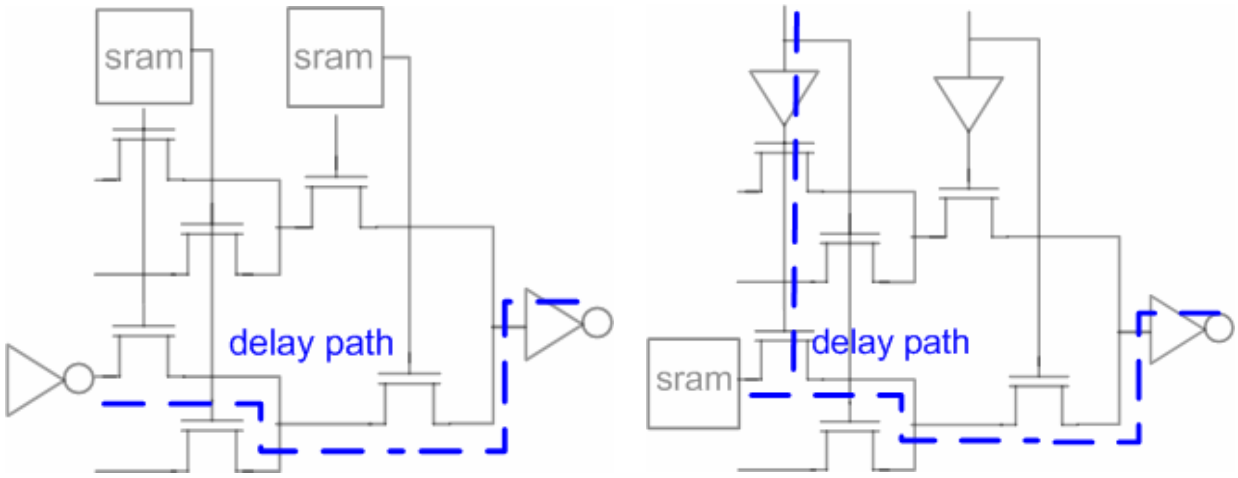


Figure 5.5: Delay optimization problem specification for LUT input-selection paths

The above optimization is not required for other multiplexing circuits in the island-style eFPGA architecture, because the critical path in all the other multiplexing logic circuits does not include the select signal path. For example, the select-inputs of the LUT input multiplexer are driven by *static* values that are stored in SRAMs after configuration (see Figure 5.6(a)). Therefore, the critical delay path for this multiplexer is simply the path from the multiplexer input to its output as shown in Figure 5.6(a). However, the critical delay path for a LUT is shown in Figure 5.6(b), and includes not only the input-to-output path delay of Figure 5.6(a) but also the delay of the select-input-path.



(a) "regular" multiplexer **(b) LUT multiplexer**

Figure 5.6: Critical delay paths for the different multiplexing circuits in an eFPGA

Also related to area reduction was the decision *not* to include buffers at each input of wide fan-in multiplexers but instead leave them as source/drain inputs. For multiplexers used in implementing LUTs, this was an obvious choice since these inputs are SRAM-driven, and the SRAMs had been sized for this purpose. However, for multiplexers used elsewhere in the eFPGA (e.g., the LUT-MUXs), this decision required further consideration. In particular, difficulties arise during driver sizing for source/drain inputs, because conventional ASIC tools are designed for CMOS logic with gate inputs and not for pass transistor logic circuits that have source/drain inputs. Furthermore, source/drain capacitances when considered alone, underestimate the load of pass transistor logic paths, while the loading CMOS gates of a given size allow for accurate input load estimation.

Figures 5.7(a) and 5.7(b) illustrate the problem that diffusion loads pose for standard ASIC tools. In Figure 5.7(a) it is evident that specifying the input diffusion capacitance (C_{diff}) of a pass transistor as the input load is inaccurate, because, the real load is the RC network highlighted from the multiplexer input to the output buffer input. Since standard ASIC tools are not designed to handle

RC loads of this type, the configuration shown in Figure 5.7(a) is recommended since the buffer gate capacitance (C_g) becomes the input load – ASIC tools are only able to interpret loads of this kind.

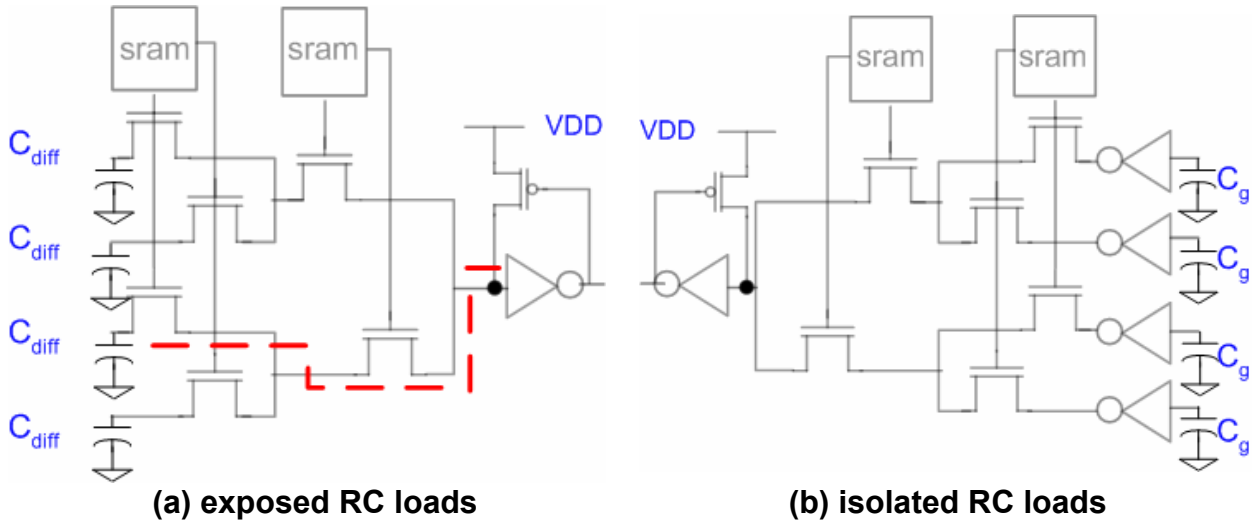


Figure 5.7: Issues around input loading for pass tree networks in the ASIC flow

However, in our case, the multiplexer circuits can be designed with diffusion inputs because the eFPGA architecture allows multiplexers to benefit from buffer sharing. Therefore, rather than buffering each multiplexer's inputs and increasing the total cell area (see Figure 5.8), a single buffer chain is used to drive multiple shared inputs (shown in Figure 5.8). Furthermore, rather than using ASIC tools to get an inaccurate estimate of buffer sizing for these multiplexer inputs, a special case of the Elmore equation [42] for distributed RC networks [43] achieves accurate buffer sizing and improved circuit speed. Equations (C.0) through (C.2) in Appendix C.1 are based on equations in [66] and are used to estimate buffer sizing for a given pass transistor RC network (similar to Figures 5.8 and C.1). Logical effort techniques are then used to find the number of stages and size of buffers to drive the buffer obtained from these equations. The appropriate buffers are then instantiated as gates in the eFPGA description. This process is made easier because it is known *exactly* where these buffers will be used in the eFPGA and so worst-case loading constraints are easily estimated. An example calculation that uses these equations for buffer/driver sizing is presented in Appendix C.2.

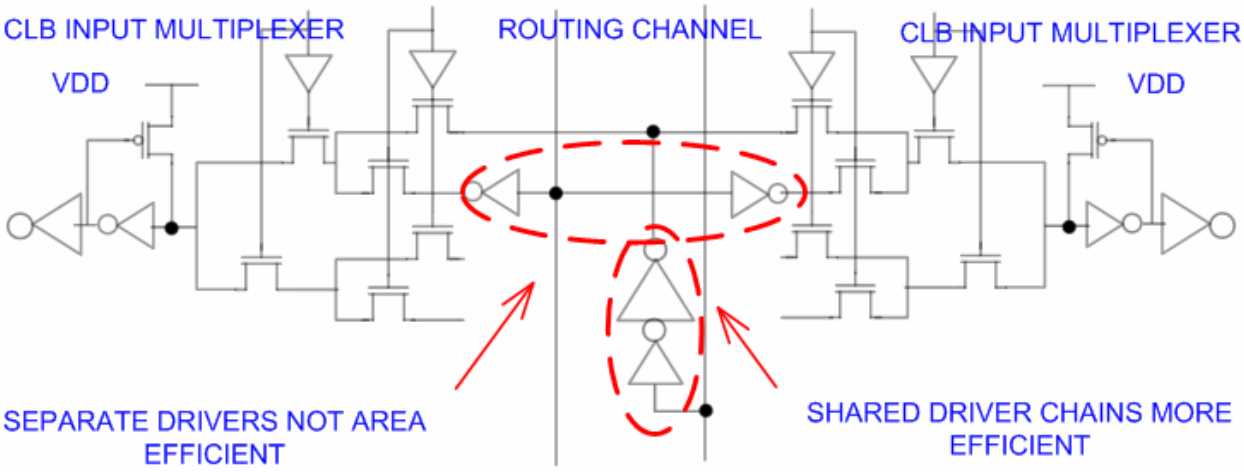


Figure 5.8: two possible multiplexing-logic buffering schemes for eFPGA design

In multiplexers with more than four levels of pass gates from input to output, repeaters are needed in the pass tree RC network to speedup the critical path. A significant degradation in speed occurs as the depth of the pass transistor tree increases with increasing input size. This result is not very surprising because it is known that the delay of a pass transistor network is *quadratic* with the number of pass transistors in the tree from input to output (depth). SPICE experiments showed that repeaters after every four levels of pass transistors sped up the circuits as needed. Also, since the pass tree transmits a weak high logic value, skewed gates are used in the repeaters. In particular, a $\frac{W_{nmos}}{W_{pmos}}$ ratio of 1.5 was used because this gave the best area, speed, and power tradeoffs.

Finally, analytical results for the repeater problem in NMOS-only pass transistor tree networks in Appendix C.2 corroborate the experimental results that were obtained through SPICE simulations.

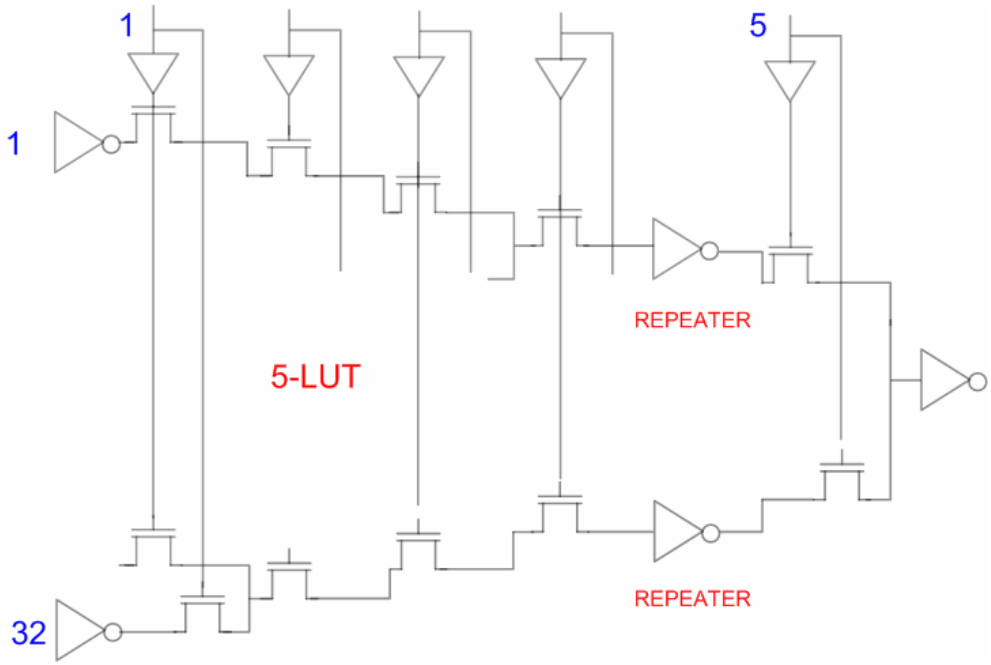


Figure 5.9: RC network representation of repeater insertion in an eFPGA 5-LUT

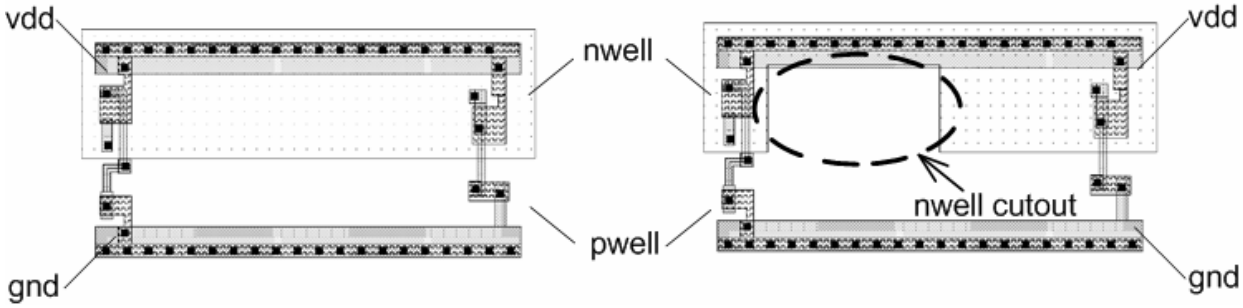
5.3 Layout Design

Custom layout design was the next step after detailed transistor-level circuit design. Since this research is concerned with standard-cell-based design, and given the fact that not all the cells in the eFPGA implementation may be substituted with custom tactical cells, it was important that our tactical cell layouts match the format of the original standard cell library [24] used in this work. For example, rules for cell height, cell width, pin location, n-well overlap had to be enforced throughout.

Perhaps the most important of the layout challenges had to do with how best to implement an NMOS pass transistor tree network within a standard cell format. As shown in Figure 2.2, a typical standard cell reserves the upper part of the cell (closer to Vdd) for the PMOS network of CMOS logic while the lower part (closer to Gnd) is reserved for NMOS transistors of the CMOS pull-down network. However, since the most area-efficient wide fan-in multiplexers are best implemented

using NMOS pass transistors, there was a potential problem of wasting the space in the upper section of the standard cell. Therefore, it was necessary to find new ways to improve area efficiency.

In order to make the most efficient use of standard cell area for NMOS pass transistor network design, a technique was devised to make good use of the otherwise wasted space reserved for PMOS transistors. In particular, a significant portion of the n-well was cut out from the middle section of each multiplexer cell so that more NMOS transistors could be included without increasing cell size. Figures 5.10(a) shows a normal cell before making the n-well cut and Figure 5.10(b) is the result of making the n-well cut. Notice that in Figure 5.10(b) the n-well around the fringes of the cell have been reserved so that level-restoring logic that includes PMOS transistors can still be implemented.



(a) before n-well cutout **(b) after n-well cutout**

Figure 5.10: standard cell layout structure before and after n-well cutout is made

Further, part of the n-well regions were used to implement CMOS logic drivers (inverter chains) for pass transistor gates. Also of importance, is the fact that no design-rule violations occur when these cells are abutted against "normal" cells. In essence, fringe n-well regions preserve the continuous n-well region that ***must*** extend across an entire standard cell row after cell placement and routing.

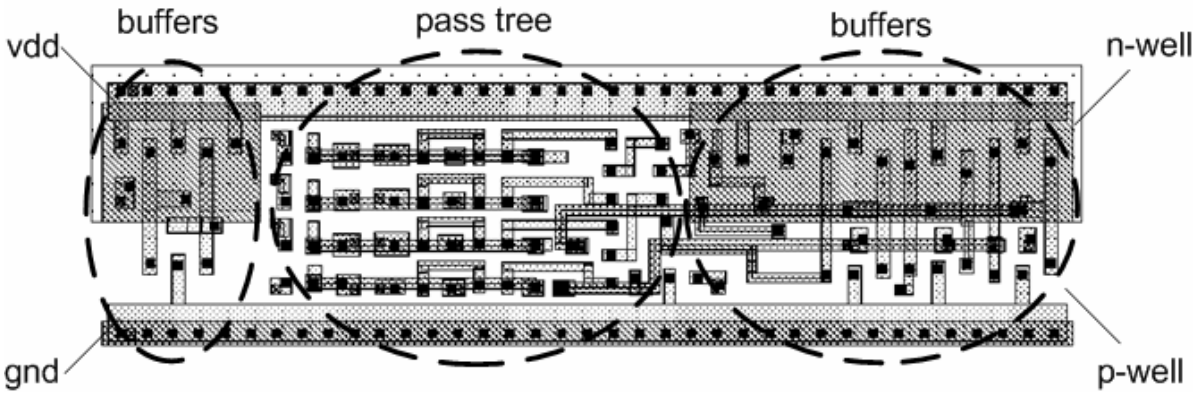


Figure 5.11: illustration of resource allocation in a multiplexer standard cell layout

In addition to the above design strategies, multi-height standard cells [78] were also used to create more area-efficient designs. None of these new cells use more than two metal levels for routing. Figure 5.12 shows a double-height standard cell that uses two metal levels in a 32:1 multiplexer.

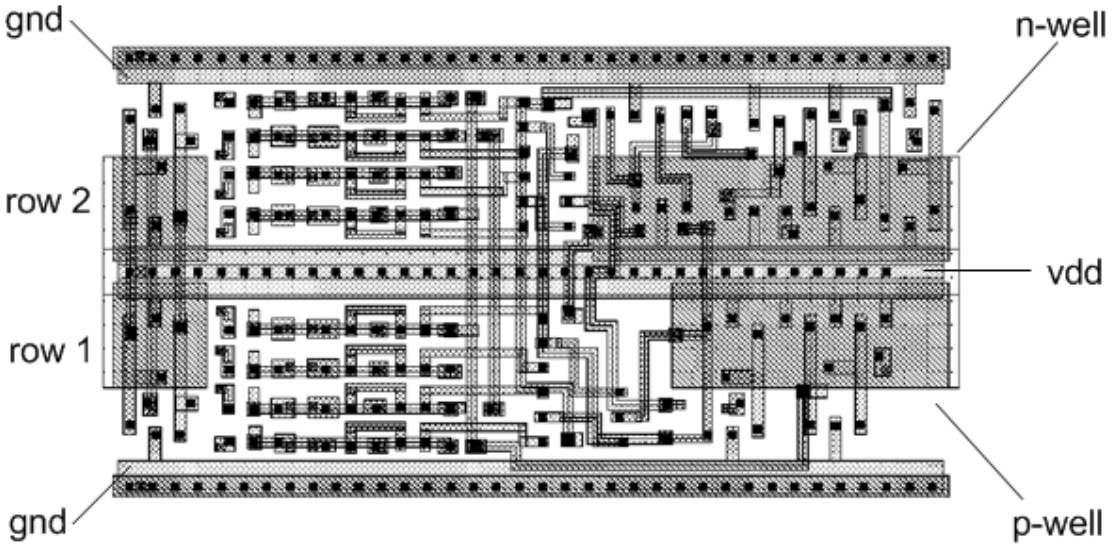


Figure 5.12: double height standard cell layout of 32:1 multiplexer (2 metal layers)

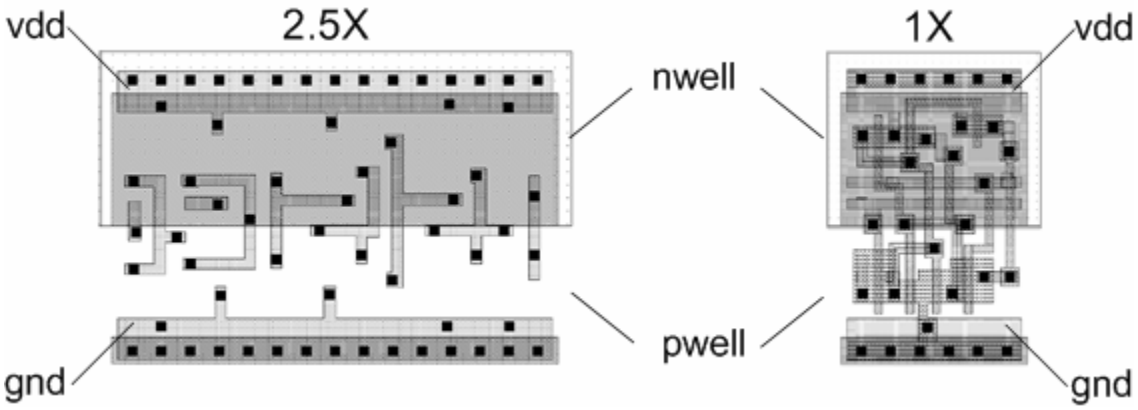
5.4 Layout Improvements

Results from tactical cell implementation are provided in Table 5.1. They show that it is possible to achieve area improvement factors of 2.5x for a single SRAM cell when compared to a flip-flop. In addition, area reductions of between 3.5x and 7.6x for multiplexer and LUT circuits were achieved.

Table 5.1: layout area improvements with tactical cells vs. generic standard cells

Cell	Generic Standard Cell Area (um^2)	Custom Standard Cell Area (um^2)	Improvement Factor
1-SRAM	61	24	2.5
8:1 Mux	267	77	3.5
16:1 Mux	899	146	6.1
32:1 Mux	2,228	293	7.6
4-LUT	1,875	530	3.5
5-LUT	4,180	1,061	3.9

The SRAM cell is smaller than a flip-flop because it has fewer transistors (compare Figures 4.11 and 5.2). Also, the custom multiplexers are much smaller than generic standard cell implementations because minimum size NMOS pass transistors occupy less area compared to CMOS logic gates (see Figure 4.10). Furthermore area saving techniques described in Section 5.3 impact cell area efficiency.



(a) Flip-flop (routing only) **(b) SRAM (Detailed)**

Figure 5.13: l flip-flop used for configuration memory in the previous approaches

Figure 5.13 shows a tactical standard cell SRAM alongside a standard cell flip-flop to further illustrate the size differences that can exist between customized cells libraries and generic libraries.

5.5 eFPGA Design Results

Given the significant cell area reductions achieved with custom tactical standard cells relative to generic standard cells, the next step was to evaluate their impact on the overall area and performance of standard-cell-based, island-style eFPGA architectures described in prior chapters.

5.5.1 Area Improvements

In order to evaluate the impact of tactical cells on the area of eFPGAs implemented with generic standard cells, a detailed area breakdown of island architectures implemented using generic standard cells was performed. These results were retrieved from reports generated from the ASIC design tools. From these reports it was possible to measure the area contribution of each cell or cell group used in the soft eFPGA implementation. With this information it was then straightforward to replace cell-groupings in the eFPGA with their equivalent tactical standard cell implementations. Therefore, in order to estimate the impact on area of tactical cells, the total eFPGA core area was re-calculated as if tactical cells had been used instead of generic standard cells. This was done for all cell groups for which custom tactical standard cell equivalents were designed and implemented.

Experiments show that on average, an ***area reduction of 58%*** is achievable for an island-style architecture using customized tactical standard cells rather than generic standard cells. In Figure 5.14, the results are provided for a set of 9 MCNC benchmark circuits and all area results have been

normalized relative to the area per tile of a full-custom equivalent. Other circuits were considered but only the results for some of the largest benchmarks (between 56 and 1300 4-LUTs) are shown. Also, in Figure 5.14 the results obtained with custom tactical cells are compared with a full-custom eFPGA of the same architecture. The area results for the full-custom implementations are based on estimates of area obtained from a version of the VPR tool that is characterized for 0.18 um process [49]. The results show an equivalent full-custom eFPGA is about 2-3X smaller than the implementation that uses custom tactical standard cells. The average overhead (geometric) is 2.86X.

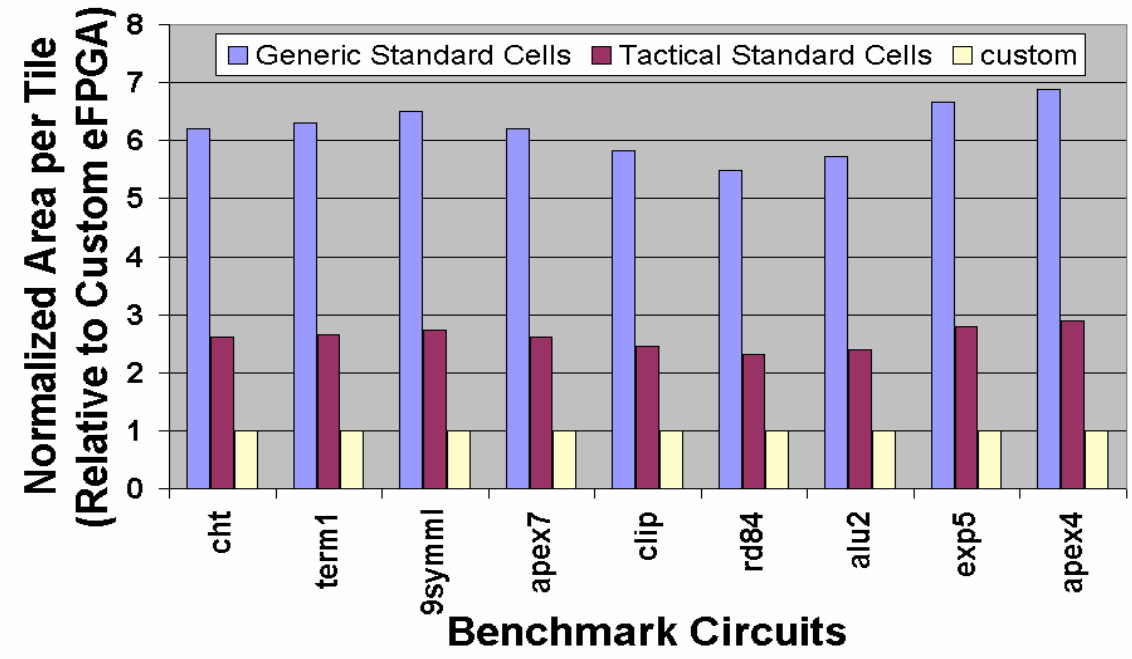


Figure 5.14: area comparisons of customized tactical standard-cell-based eFPGA implementations with generic standard cell, and full-custom implementations

5.5.2 Delay Improvements

In order to evaluate the impact of tactical standard cells on eFPGA speed, a scheme was devised that leveraged existing static timing verification tools in the ASIC flow. In particular, the eFPGA gate netlist was "programmed" as described in the previous chapter and then a static timing report was generated for the critical delay paths in the original eFPGA ASIC implementation. This report

details the timing contribution of standard cells and wires in the critical path. From this report, it is possible to identify the generic standard cells or standard cell groups that would be replaced with tactical cells if the eFPGA were implemented using these cells. The main task was then to characterize the tactical cells for speed in a fashion that accounted for the loading and drive characteristics of the cells that remained unchanged (that is, cells not replaced with tactical standard cells).

SPICE simulations were used to characterize the tactical cells for speed. However, rather than simulate each cell in isolation, it was more efficient to simulate groups of tactical cells that were connected as if in an actual FPGA. For example, the schematic in Figure 5.15 models the path starting from an input connection block multiplexer input ("A" in Figure 5.15) and ending at the output of a LUT in a BLE ("D" in Figure 5.15). Notice also that in this figure the loading due to other parts of the eFPGA architecture have been included (e.g., LUT-MUXs). Also modeled is the path starting at the output of a LUT and ending at the output of another LUT in the same CLB (e.g. the path from "D" to "G" in Figure 5.15). As shown in Figure 5.16, the path from the input of a BLE output tristate buffer through one level of switch element and ending at one of four possible switch outputs was also simulated (the choice of switch output is unimportant since each one is loaded about the same). This is the path from "B" to "C" in Figure 5.16. The goal of this SPICE experiment was to determine the timing overhead from the input to the output of a switch element.

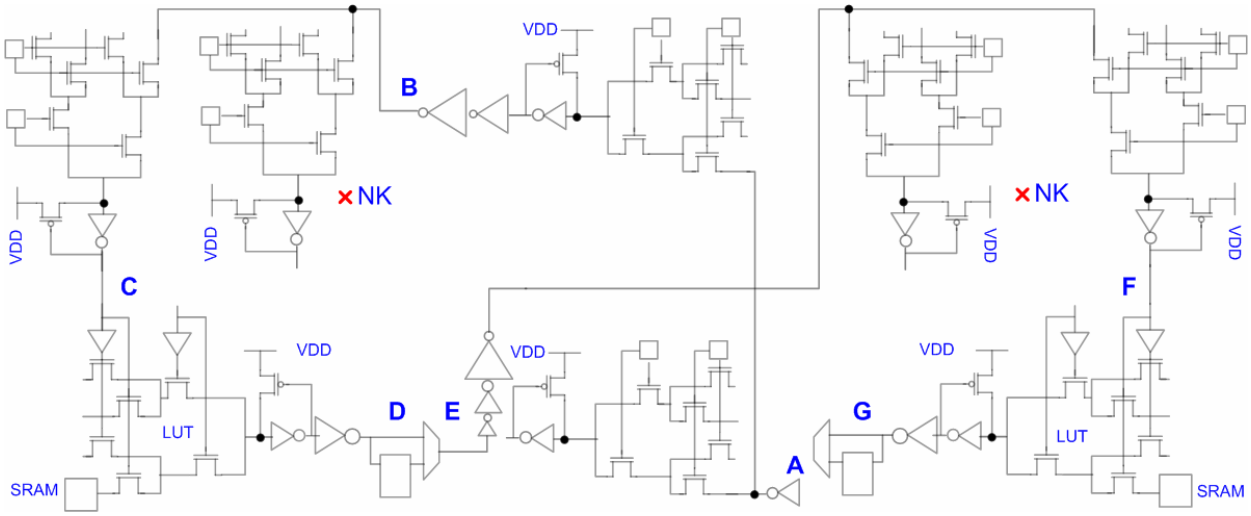


Figure 5.15: illustration of SPICE simulation setup to measure logic block delay

During the experimental setup for speed characterization, it was important to ensure that we simulated the loading effects due to cells or cell groups that were not replaced with tactical standard cells. For example, the track buffers (see Figure 5.8) in the original standard cell implementation were of fixed drive strength for ***all*** eFPGA implementations. In the current implementation, these buffers are not replaced with tactical cells and so it was important to account for this in the SPICE simulations. Therefore, track buffers used in experiments depicted in Figures 5.15 and 5.16 are the same ones used in the original standard cell implementation. There are other instances where a similar approach was required, and so special scripts were created to extract this information from the netlist and create a database. This was needed because cells used to implement the same function can differ depending on the value of N, for example, or timing constraints. Creating a database of different implementations and the cells used in each case, made it possible to determine the worst case loading constraints for use in all the SPICE simulations. All tactical standard cell SPICE simulations were done under nominal process conditions for 0.18um process technology node.

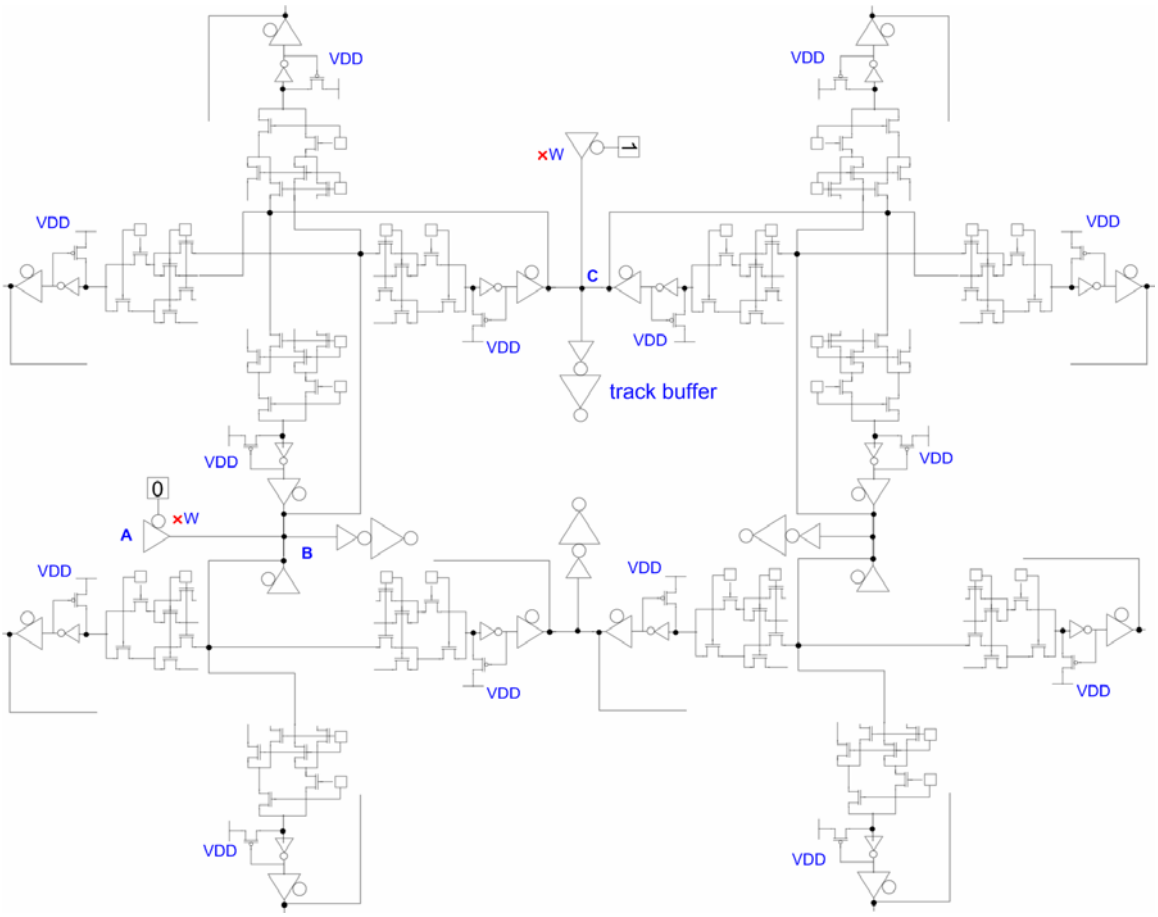


Figure 5.16: SPICE simulation setup to measure the routing switch element delay

Once all paths of interest had been characterized for timing using ***extracted layouts*** of the tactical standard cells, the data was used to back-annotate static timing reports generated during static timing verification of the original eFPGA implementations. Scripts were created to annotate and recalculate timing for all previously reported critical paths for different eFPGA implementations.

The implementation based on customized tactical standard cells shows an ***average circuit speedup of 40%*** on the critical path, compared to an eFPGA implementation using generic standard cells. For the benchmark circuits considered, (includes some not shown in Figure 5.17) the range of circuit speedup was between 28% and 48%. Results for 9 MCNC benchmark circuits are shown in Figure

5.17 below as well as comparisons with equivalent full-custom implementations. Our timing results for full-custom implementations of the same architecture are also based on estimates from a version of the VPR CAD tool that was characterized for 0.18um process. Furthermore, the timing results achieved with our custom tactical standard cells are within 10% of a full-custom implementation.

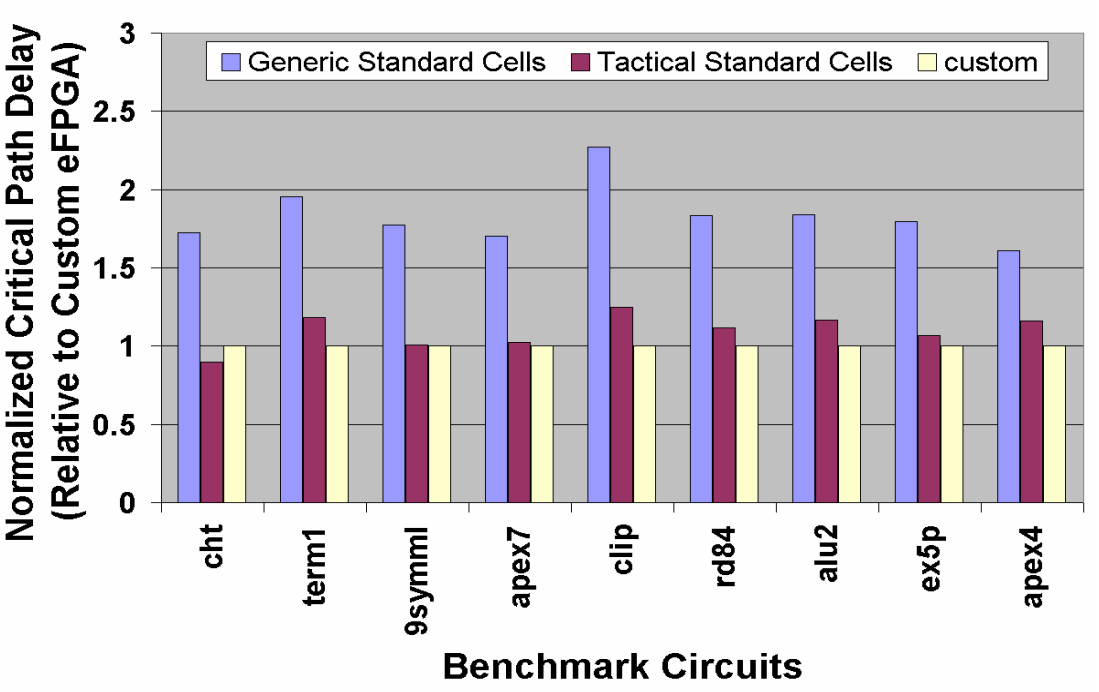


Figure 5.17: delay comparison of customized tactical standard-cell-based eFPGA implementations with generic standard cell, and full-custom implementations

Although the current work has not focused on power reduction *per se*, it is reasonable to assume power savings have been achieved as a side-effect of the work done here. For example, it is reasonable to assume that since the overall cell area has been more than halved, gate capacitances and interconnects would also experience similar reductions, and thereby reducing power dissipation.

5.6 Comparison to GILES

It was also useful to compare the standard cell approach with the GILES approach [68] [69] [70]. GILES is essentially an automated, cell-based, semi-custom design strategy for implementing programmable logic tiles. This approach is similar to the standard cell approach used here except, a new custom suite of "FPGA-aware" tools were developed for this purpose. Specifically, an entirely new back-end suite of tools for cell placement and routing was developed. Furthermore the cells used to implement programmable logic tiles do not adhere to the standard cell format. For example, cells are not necessarily the same height and so Vdd and Gnd lines may not always be abutted.

In [68] the authors reported that their approach resulted in a Virtex-E FPGA tile implementation that was within 36% of the full-custom implementation of the same tile. Their results also suggested that a standard cell based implementation of the Virtex-E tile using custom SRAMs and pass transistors in the output connection block [69], resulted in a tile implementation that was 75 % larger than the full-custom version. The authors do acknowledge in [68] [69] that wide-fan-in multiplexers in the standard cell implementation were not replaced with pass-transistor based multiplexers. The multiplexers in the GILES implementation used pass-transistors. Our work has already shown that this can have a significant impact on the densities achievable with standard cells.

In order to evaluate the impact of wide-fan-in multiplexers on the standard cell implementation of the Virtex-E tile, it would have been ideal to have access to the netlist used to implement the tile. However, since this information is not readily available, a reverse approach was used: rather than improve the standard cell implementation with pass-transistor-based wide-fan-in multiplexers, the GILES implementation is made worse by "bloating" the multiplexers in the "netlist" so that the cell

areas are at "pre-optimization levels". In other words the cell areas are increased so that they equal the area of a generic standard cell implementation of the same multiplexer. This was possible because the database of ***all*** cells used in the GILES implementation were provided by [69] [70].

To scale the multiplexers in the GILES implementation appropriately, the areas of equivalent generic standard cell implementations had to be determined. For this purpose, the graphs in Figures D.1 through D.3 were used. The plots labeled "X" in all three figures are based on a database of area results from the synthesis of different size multiplexers used throughout our research. For the synthesized multiplexers, each data point is the average synthesis area for that size of multiplexer. In addition, the plot labeled "Y" in Figure D.1 gives the scaling trend for pass-transistor-based multiplexers relative to the number of inputs, and is based on actual layout results. Comparing our layout results with the layout results for the GILES cells showed the area results are more-or less identical. For example, based on our layout results (see plot "Y" in Figure D.1), a 25-input multiplexer has a layout area of 140um^2 while an equivalent GILES cell has an area of 174um^2 (based on supplied, unpublished data for the GILES cells). Figures D.2 and D.3 also use data based on synthesis results and actual layout results to predict the scaling trends. A different plot was needed for LUTs because full-custom layouts of LUTs include buffers for fast select inputs (Figure D.2).

Table 5.2 shows the layout area and scaling factors for our tactical cell layouts and GILES cells, versus the generic standard cell multiplexers and LUT implementations obtained using logic synthesis. In all synthesis experiments, constraints were set to give the best balance between total cell area and overall delay. The multiplexers listed in Table 5.2 are the same ones used in the GILES implementation of the Virtex-E architecture. We have excluded 2-input multiplexers because there are very few of them and they are fairly small. Based on these scaling factors, the area of the GILES

implementation of the Virtex-E architecture increased by a factor of 1.62X. Given the initial results in [68] that suggested the standard cell implementation was 1.55X the GILES cell area, the new results obtained here suggest the tactical standard cell implementation is ***0.96X the GILES area***.

Table 5.2: The scaling factors for multiplexer logic bloating in GILES Virtex-E tile

multiplexer inputs	LUT size (K)	tactical cell area um^2	GILES cell area um^2	generic cell area um^2	scaling factor
	4	149.02	107.59	810.78	5.44
12		70.13	71.87	333.85	3.10
16		91.48	107.59	448.93	4.17
25		139.51	174.24	707.86	4.06
26		144.85	181.21	736.63	4.07
4		27.44	24.39	103.69	3.78
6		38.11	39.20	161.23	4.11
8		48.79	52.71	218.77	4.15

The results above would seem to place a tactical cell eFPGA within 36% of a full-custom design, which also contradicts estimates obtained from VPR. For this reason a methodology to estimate the area overhead was devised and is explained in Appendix D. Our estimates (see Table D.1) place a tactical standard cell eFPGA at between 1.66X and 3.11X. However, the more likely estimate is probably between 1.66X and 2.84X (closer to 1.66X) since these estimates are based on an architecture (Virtex-II) that is similar to the island-style architecture used in this research work.

It should be noted that these results are sensitive to the synthesis constraints used to obtain the generic standard cell areas. As a result, the scaling factor used here may be less or higher depending on the design emphasis (speed, area, or both) of the original GILES implementation. This is not clear from the literature that has been published on this topic. However, since all synthesis constraints were tuned for area ***and*** delay optimization, it is likely that our scaling factors are reasonable. It is not possible to investigate this any further within the scope of the current work.

5.7 Sensitivity Case Study

In this section ***architecture sensitivity to standard cell libraries*** is investigated. For example, the product-term architecture [15] [16] benefits from the fact that its product-term block (roughly 76% of core area) is based on logic gates like NANDs and NORs that are already well tuned in most commercial standard cell libraries. This is not the case for the island-style architecture. Therefore, in this experiment, tactical cells are used to implement some parts of the island-style architecture. In particular, the logic component of the island-style architecture (contents of CLB excluding program flip-flops) can benefit from the inclusion of tactical standard cells. This is essentially equivalent to implementing the product-term blocks in [15] [16] with NANDs and NORs. The routing architecture for the island-style architecture (connection blocks and switch blocks) and product-term architecture (wide fan-in routing multiplexers) [15] are left unchanged. Figures 5.18 to 5.20 show the experimental results. Island-MUX and Island-Tribuf are island-style architectures with multiplexed and tri-state routing switches respectively, and PTB is the product-term-based architecture [15] [17].

The results in Figure 5.18 show that for the set of benchmarks considered, the product term architecture is smaller than the LUT-based island-style architecture in all but one case. On average (arithmetic), the product term architecture is 34.85 % smaller than the island-style architecture. This is almost identical to the result obtained from comparisons in [15] [16] with a different LUT-based architecture [02] [14]. This would appear contrary to expectations since it has been observed that the LUT-based island-style architecture requires less area on average than the current implementation of the LUT-based gradual architecture used in comparisons in [15]. Hence, one would expect a smaller (less than 34.85%) average area differential between the island-style and product-term architectures.

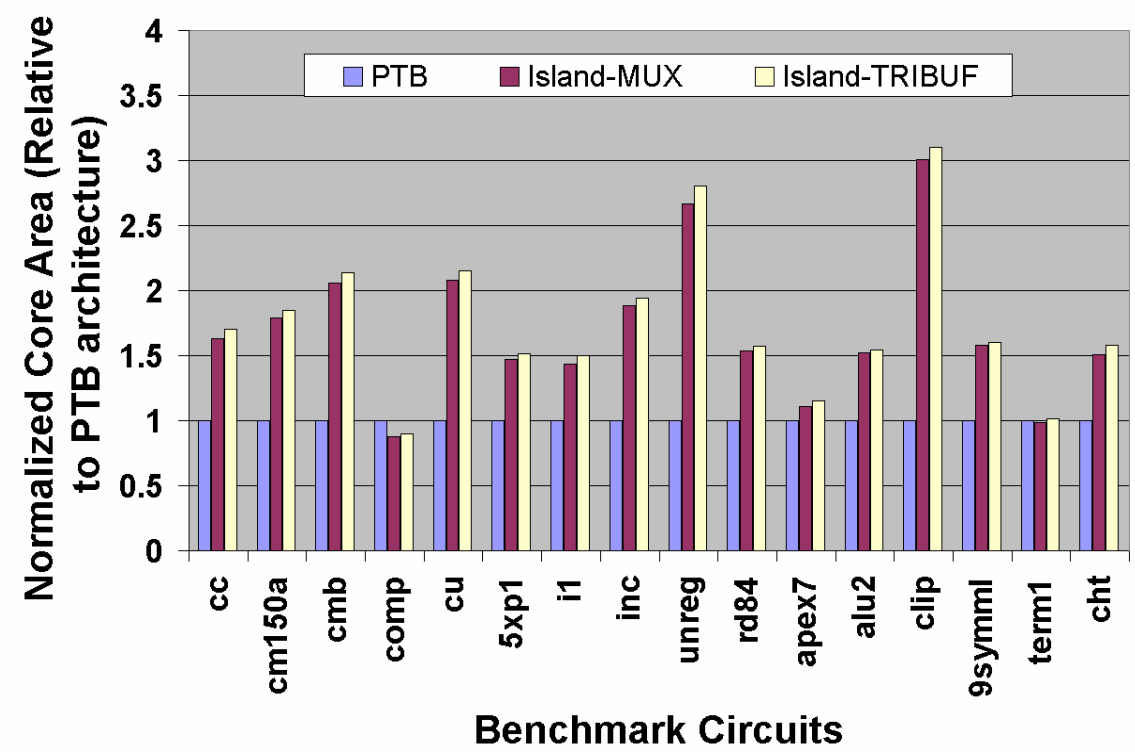


Figure 5.18: Core area comparison of product-term and island-style architectures

Furthermore, for this experiment, most of the benchmarks used are those for which the average area overhead of the LUT architecture [02] relative to the product-term architecture is very high on average (54.33%). Furthermore, analysis of the LUT-based island-style architecture showed that the area of the core is reduced by an average of 23.5% when pass transistor based multiplexers are used in the logic architecture. Therefore, the 34.4% area differential reported is in-line with expected results. This means that if all the benchmarks used in experiments in [15] had been considered here, the average area differential between the two architectures would likely be less than 34.85%. In addition to demonstrating architecture sensitivity to cell libraries, these numbers demonstrate that results obtained for different architectures are also sensitive to the benchmarks used in experiments.

The delay results in Figure 5.19 show that the island-style architecture with multiplexed switch elements [20] (Figure 3.2) is faster than the product-term architecture for most of the benchmarks considered. These are the results obtained after the inclusion of tactical cells in the logic architecture. On average (arithmetic), and for the benchmarks considered, the LUT-based island-style architecture was 19.7% faster than the product-term core. The tri-buffered switch architecture was on average 37.4% faster. Although not shown, it was also found that even without the use of pass transistor multiplexers in the logic architecture of the island-style architectures, Island-MUX and Island-TRIBUF are respectively 8.8 % and 29.4% faster than the product-term architecture. These results suggest a significant change from the results presented in [17], even-though the LUT based architecture used in [17] for comparisons is different from the one used in these experiments. This could be related to the fact that delay results used for the LUT-based architecture [02] are based on non-functional timing paths relative to the corresponding benchmarks. In other words, the fabric was not programmed. Furthermore, it has been noted previously that the programmed and un-programmed critical path delays results for the product-term architecture are very closely matched. However, this is not the case for the gradual LUT-based architecture [02] that was used in [16] [17].

As a result of the significant differences in delay between the programmed and un-programmed versions of the LUT-based fabric used in [17], the speed gap between the product-term and the LUT-based architecture is much larger than it should be. This explains (in part) why results obtained here seem like a huge reversal in the delay trends of LUT-based versus product-term architectures.

Figure 5.20, compares the area-delay-product for the product-term and island-style architectures. For the set of benchmarks considered, the island-land style architecture, Island-TRIBUF, has on average (geometric) an area-delay-product that is 0.99X that of the product-term architecture in [17].

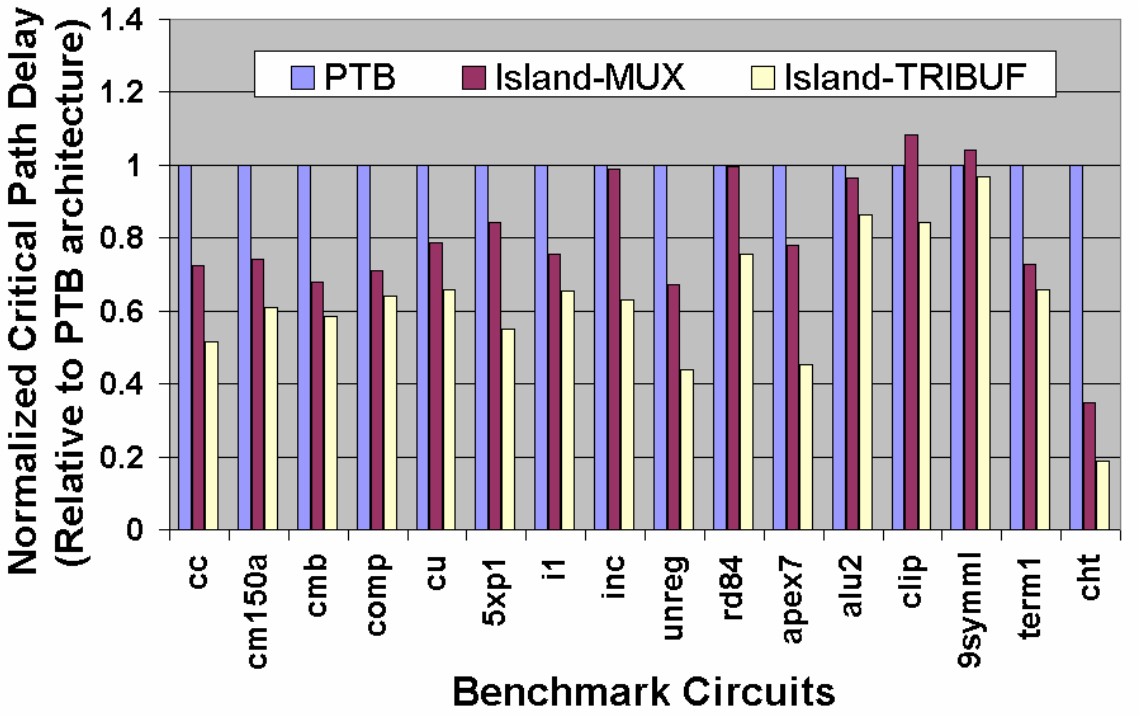


Figure 5.19: delay path comparison of product-term and island-style architectures

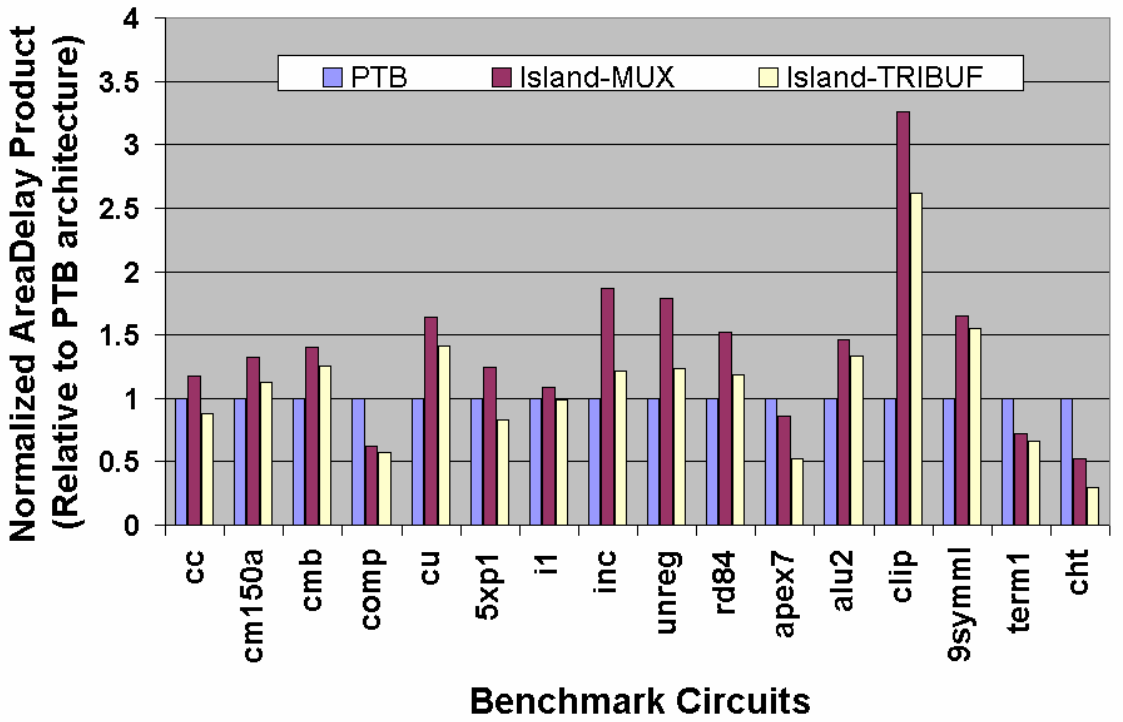


Figure 5.20: Area-delay-product comparison of product-term and island-style core

The arithmetic average of the area-delay-product scaling factor for Island-TRIBUF is 1.11X relative to the product-term architecture. Further, Island-MUX has geometric and arithmetic averages for area-delay-product of 1.26X and 1.38X respectively relative to "PTB". These results her show that the area-delay product of Island-TRIBUF is comparable to the product term architecture [15] [16].

5.8 Mux Switch Evaluation

Finally, "Improved multiplexer switch 2" was evaluated as a possible candidate for tactical cell implementation in a future architecture implementation. This design is somewhat of a variant of "multiplexer switch 1" because it also aims to speedup horizontal and vertical routes through a switch element. To evaluate its potential, results from SPICE simulations were used to annotate some MCNC benchmarks as described earlier. SPICE simulations revealed that the fast switch route was 12.5% faster than the "slow" routes. For the smaller benchmarks this only translated to a 1% speedup. For larger benchmarks like ex5p and apex4 speedups of 3% and 6% respectively were achieved. It was anticipated that even larger speedups can be achieved for switch multiplexers with wider fan-ins since slower paths will have more levels of pass gates relative to the fast path with a single pass transistor (plus buffer delay for all paths). However, an *independent* study in [21] showed that speedups of about 6% were achieved in architectures of this type. It is not immediately obvious why the gains were not higher but one reason could be due to the fact that most of the delay reductions come from the use of longer wires (e.g. length 4 wires) in the architecture [21]. It would be interesting to see if the proportion of length 4 wires in the architecture could be reduced in favor of these switches without adding significantly to area, and also reducing dynamic power dissipation.

Chapter 6

The Implications for eFPGA IP Design

The improved area and performance results of the previous chapter have some important implications for current eFPGA design and implementation strategies. Although the research work presented in this thesis is an initial exploration of ***potential*** implementation and design strategies for eFPGA circuits, and largely independent of the larger question of how eFPGA IP should be delivered to the end-user, this thesis chapter provides a position on this larger research question.

At present, the popular method for delivering eFPGA IP to end-users is hard IP. However, as illustrated in Figure 2.12, restrictions on core size introduce significant inefficiencies. An alternative is the soft approach [14] [16] where a configurable RTL description of an eFPGA can be implemented using ASIC tools and generic standard cell libraries. This approach allows customization [48] that is impossible with hard IP (due to fixed hard IP core sizes and composition), but the reliance on ***generic*** standard cells introduces significant inefficiencies. A "middle-ground" approach that combines positive elements of hard and soft IP approaches in a manner that allows customization to user specification, and retains superior area and performance metrics is desirable.

The middle-ground approach suggested above would require an IP generator similar to commercial SRAM generators that exist today [18]. Assuming a generator of this kind existed for eFPGA IP and was based on customized libraries of standard cells, results in the following subsection are intended to illustrate how such a generator would compare to the current hard IP approach.

6.1.1 Some "Real world" Case Studies

In order to compare an IP generator approach (based on customized standard cell libraries) with the existing hard IP approach, 9 benchmark circuits were used. Each benchmark circuit can be viewed as part of a different embedded application that might need to be modified in the future (for bug fixes or upgrades). Seven of these circuits range from 56 to 200 LUTs, and the 2 largest benchmark circuits have 1112 and 1340 LUTs respectively. For this experiment, the LUT input size (K) and cluster size (N) were fixed at 4 so that fair comparisons could be made with existing commercial hard eFPGA IP cores [44]. Also, the impact of embedding FPGA cores in a Bluetooth baseband SoC is evaluated using reprogrammable Frequency hopping [74] and data encryption [74] modules.

Using data available from IBM and Xilinx [41], it was estimated that on average a hard eFPGA IP core can implement anywhere from 800 to 1,371 equivalent ASIC gates per mm^2 in 180nm process. This estimate was obtained after correcting for process scaling (90nm to 180nm). These estimates are somewhat similar to estimates published in [45]. However, these estimates assume that all the logic in the eFPGA is used to implement a target circuit or benchmark. This is often not the case since the logic fabric is sometimes underutilized. We express logic underutilization in Equation (6.1).

$$\textit{Underutilized Area per Logic Tile} = \left(\frac{\text{eFPGA core area}}{\text{logic clusters used}} \right) = \frac{1}{\textit{Effective Logic Density}} \quad (6.1)$$

Equation (6.1) measures logic inefficiencies in eFPGA IP. It normalizes the overall core area relative to the actual number of logic clusters that are used to implement a target application circuit. Therefore, the fewer logic clusters a circuit implementation requires, given a certain core size, the higher the underutilization. This equation is used as the basis for the comparisons in this section. In

essence, the goal is to compare logic inefficiencies that exist in both the IP generator approach and the hard IP approach. Results for 9 benchmarks are presented in Figure 6.1. In this figure, it is assumed that hard eFPGA core sizes of 512, 1024, and 2048 LUTs are available, such as in Actel's Varicore [11]. To calculate core area (mm^2) the reported ASIC gate capacity for each core [44] was multiplied with the estimated equivalent ASIC gates per mm^2 for 180nm process technology.

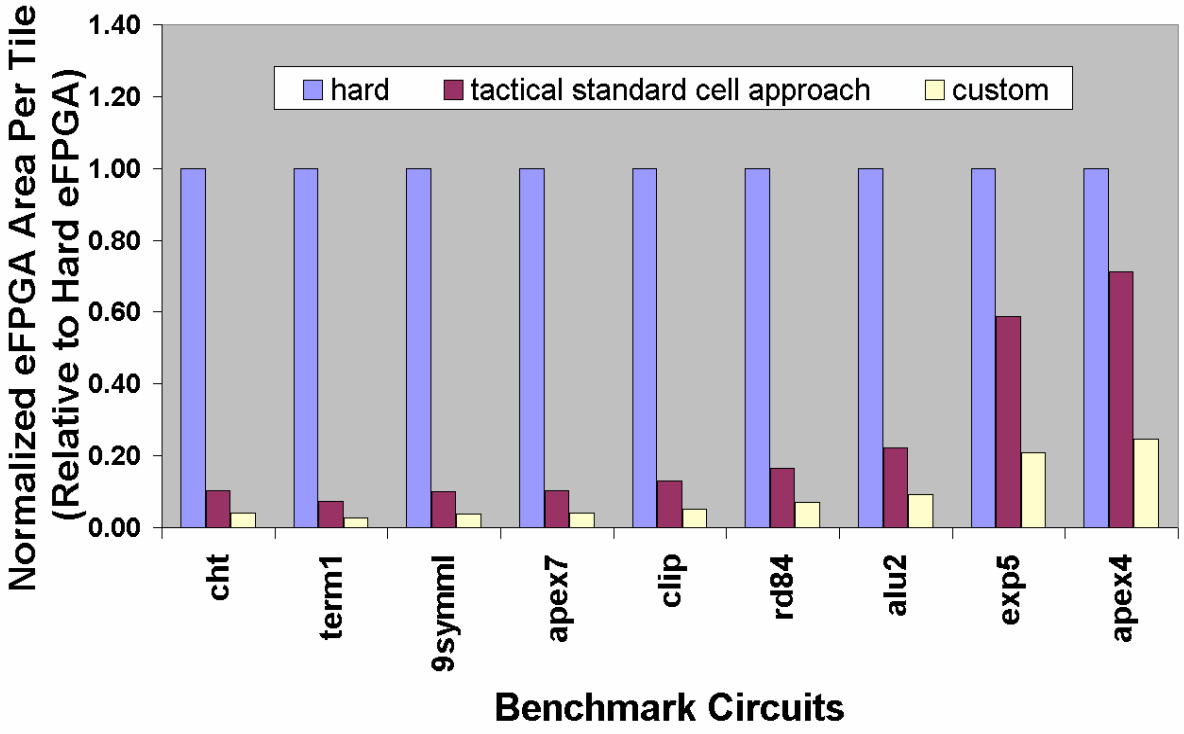


Figure 6.1: logic efficiency comparisons of a standard-cell-based eFPGA IP generator approach to a commercial hard IP approach for 9 MCNC benchmarks

The results in Figure 6.1 suggest that for small benchmark circuits, it is *very* important to match the *logic capacity* of cores to the circuit needs. Also, even for larger benchmarks such as ex5p and apex4 with over 1000 LUTs, the IP generator approach can still achieve effective logic densities that are higher compared to the hard eFPGA IP approach. From the results for the two largest benchmarks, the use of customized standard cell libraries makes the IP generator approach

competitive. Furthermore, the use of configurable architectures in the IP generator approach makes it possible to prune routing overhead to save area while hard IP tiles typically contain excess routing.

While an IP generator based approach using custom standard cell libraries generates cores that are larger than a full-custom eFPGA implementation of the same size and composition (Figure 5.14), it still can achieve effective logic densities (as defined in Equation (6.1)) that are superior or comparable to the best available hard eFPGA core (Figure 6.1). This stems from the fact that eFPGA vendors at present do not have the means to design full-custom eFPGAs for each application, or customize one based on some other efficient circuit implementation method. Instead, cores of fixed size and composition are built, and the inefficiencies predicted in Figure 2.12 occur.

The impact of including standard-cell-based eFPGA cores in a Bluetooth baseband SoC was also evaluated. Based on actual eFPGA layouts and estimates of improvements due to tactical standard cell substitution, an eFPGA that implements the baseband frequency hopping (FH) module would result in a ***core*** area increase of 31%. If instead the encryption module eFPGA is included, ***core*** area increases by 130% (2.3X). Both results assume an eFPGA with bidirectional routing and single segment wires. Assuming a tactical standard cell implementation with length 4 (L4) directional wires, the corresponding ***core*** area increase from the encryption module eFPGA is about 80% (1.8X). Its is not expected that the FH module eFPGA would benefit significantly in terms of area, from length 4 and directional wires. To put these results in perspective, consider that the smallest available hard IP core in 180nm technology with a ***similar architecture*** is about 6.25mm^2 based on data in [41] [44]. However, FH module eFPGAs based on generic and tactical standard cells are respectively 5mm^2 and 2.1mm^2 in area. Further, the larger encryption module eFPGA (784 4-LUTs) with length 4 wires and directional routing [20] has an estimated area of 5.3mm^2. This is still smaller than the smallest

available hard IP core [44] in 180 nm with a similar architecture. Lastly, the Bluetooth SoC is I/O limited, (51% of unused area in Figure E1) hence, the inclusion of the FH module eFPGA ***or*** encryption module eFPGA (L4 wire + directional routing + tactical cells) has no effect on ***die area!***

Finally, although the results above show that the tactical cell approach surpasses the ***effective*** logic density of the hard IP approach, there have to exist cases where this is not true (See Figure 2.12). However, with continued efforts to improve automated eFPGA generation methods (e.g. using other more efficient circuit building blocks), it may be possible to surpass ***effective*** logic densities of existing hard eFPGA IP cores [44] across all target circuit sizes. In essence, we advocate a departure from the existing commercial hard eFPGA IP approach towards a more adaptable and efficient methodology based on an eFPGA IP generator concept described herein. Previous work [01] [02], and the limited success of hard eFPGA IP, suggests the need for a paradigm shift in eFPGA design.

6.2 A New Paradigm for eFPGA IP Design

Despite the improvements reported here, there remains much work ahead if embedded eFPGA cores are to become mainstream. In particular, an efficient IP generator framework must be devised, so that users can tailor programmable fabrics to their own needs. The results here have shown using real world examples, that significant inefficiencies exist in current hard IP design approaches, due to the absence of application-specific customization. When users of hard eFPGA IP are restricted to IP with fixed sizes and resources, it is possible to end up with too much or too little of the resources needed for a given application. This has implications for area and speed, and presumably for power.

It has been shown in this thesis that standard cell libraries can be customized to improve the area and performance of standard-cell-based eFPGAs. However, augmenting an eFPGA RTL description with custom standard cell libraries is a deviation from the soft IP concept. For example, process independence, one of the hallmarks of the soft IP concept, would be sacrificed, because, new ***customized*** standard cell libraries have to be designed for each process migration. Moreover, it is desirable to hide the details of an architecture implementation, since this is to some extent what defines the competitiveness of a product in the market. Therefore, an IP-generator approach that is similar to SRAM generators [18] is proposed. Some of its proposed features are described next.

6.2.1 "Open" Architecture IP Library

Similar to the "open source" concept used in Linux© software development, it is desirable to have a programmable logic architecture IP library that allows programmable logic architects and designers to augment new architectures and related CAD software to an existing IP generator. This approach makes it possible for designers to select the eFPGA architecture that best suits their design needs. For example, the results in Table 5.18 to 5.20 showed that the LUT-based island style architecture is the better choice for certain benchmark circuits, while the product term architecture is better choice for others. Clearly, having the option to choose between architectures can be quite beneficial.

It is also expected that CAD infrastructure IP for architectures will include detailed area, speed and power models in addition to placement and routing software algorithms. This would allow users to evaluate the area, speed, and power characteristics of all architectures before deciding which to use.

6.2.2 Configurable Architecture IP

An eFPGA IP generator should contain architectures that are configurable. This means that in addition to having a pool of high-level architectures as described above, each architecture in the IP library must itself be configurable (i.e., it must have an associated set of modifiable parameters). For example, if using the clustered island-style eFPGA architecture described in this thesis and in [11] [13] [20] [22] [28] for an embedded application, users should have the option to change the routing segment length distribution, LUT input size (K), channel width (W), core dimension (Dx * Dy) or other architecture parameters as needed. The absence of such a flexible system in existing hard IP approaches [44] is the reason for the inefficiencies that were illustrated in this thesis and Figure 2.12. Configurable eFPGA architectures may be described in RTL or in "architecture files" [11] [68].

6.2.3 Domain-driven IP generation

Different parameterized high-level architectures in an architecture library, makes domain-driven IP generation possible. In particular, it is possible to construct an eFPGA with the best available architecture for a *specific* application (e.g., selecting a product term architecture over a LUT-based architecture) and with optimal parameter settings (e.g. with suitable channel width or core size).

The above approach is an improvement over existing approaches, because programmable logic designers in general implement programmable logic devices based on a large set of "golden" benchmarks, and do not cater to specific applications. This can result in large area overheads due to underutilized logic. Furthermore, it is expected that a large area overhead would also have a negative impact on speed and power (due to higher capacitance). To compensate, the authors of

[67] advocated architecture families comprised of sibling architectures that are in essence scaled down versions of the same "parent" architecture. The goal is that users will find an FPGA within the collection that minimizes overhead. However, inventory costs associated with having a large "family" limit the extent to which this can be done [67]. FPGA vendors use some of these same techniques and it is in a sense equivalent to having a collection hard IP core sizes as is done in [44].

Applications intended for an eFPGA are much smaller in number and better understood beforehand; therefore, inefficiencies associated with the usual design approach for standalone FPGAs and hard eFPGAs referred to above, can be avoided through domain specific customization [01] of eFPGA IP. Standalone FPGAs do not present the same opportunities because the potential application space is much larger and less is known about intended future applications.

Deciding on the most suitable eFPGA IP architecture implementation involves experiments with example circuits that are representative of the target application, and a series of parameter sweeps to determine the most suitable architecture implementation. In a SoC design for example, some of the target applications would be known in advance due to re-use of application IP, and therefore it is possible for SoC designers to tailor the generated eFPGA IP to the SoC application. It is expected that these optimization experiments, based on user-supplied design constraints, would be automated, and run within the proposed IP generator tool and therefore not visible to the user.

Finally, programmable logic architectures in general are simple enough that they are relatively very predictable. For example, the relative sizes of the logic used will change given a particular architecture implementation (e.g. different N or K values) but the logic function will be the same. Furthermore, there is no need to build multiplexing logic with different drive strengths, because,

minimum sized pass transistors have been found to give the best results in general [66] and multiplexer inputs and outputs can be buffered as needed. As a result, traditional and sometimes complex logic synthesis of the kind used in Chapter 3 and in [14] [15] is not required, and only gate resizing of input and output drivers is needed in addition to selecting the size of multiplexing logic (e.g. a 32:1 versus 16:1 multiplexer). Therefore, a reasonable approach could involve combining gate resizing with architecture optimization and using techniques similar to that used in [31] [35]. The end result would then be a circuit-level architecture netlist that has been resized and is ready for layout.

6.2.4 Automated Layout generation

A layout implementation framework is needed to compliment architectural flexibility, so that physical layouts of programmable logic architectures can be generated automatically. This could be achieved with a scripted and fully automated ASIC *backend* design flow with modifications for eFPGA design [14] [75] [78] , or a new custom design flow as in [68]. Such a design flow would typically include support for design partitioning that allows regular programmable logic architectures with repeated structures (e.g. eFPGAs) to be tiled and replicated (e.g. Cadence First Encounter [23]).

The need for an automated layout design flow that supports tile replication is based on the notion that architecture IP implemented in the proposed IP generator should have a regular structure that allows tile replication. Standalone FPGA chips and hard eFPGA cores are built using a structured approach that replicates a single well-optimized tile to form a programmable logic array or fabric. This approach allows tile re-use and improves the quality of design. A ***regular*** arrayed architecture like the island architecture makes it possible to build an eFPGA fabric from a single replicated tile.

In addition to facilitating tile re-use during design, a regular fabric also has a positive impact on the quality of results achieved with existing EDA tools. For example, it was observed that attempting "flat" logic synthesis and or place and route of a large standard cell eFPGA fabric can take several hours and produce less than desirable results, or in some cases lead to tool failure. However, a "divide and conquer" strategy allows synthesis to run till completion and achieves good results for the largest MCNC benchmark circuits (> 1,000 4-LUTS). A regular fabric structure makes this process straightforward, because a single tile (small in comparison to the fabric) can be mapped to a gate-level netlist rather easily, and then replicated at a higher level to form the programmable fabric. This same principle of localized design optimization can be applied to physical layout design as well. The fabric layout shown in Figure E.1 of Appendix E (784 LUTs) was implemented in this way.

A tiled regular fabric also simplifies eFPGA characterization and eFPGA CAD design because post-routing timing extraction and characterization needs to be done for a single tile (not the entire fabric) and then applied to all tiles since identical nets in each tile will have the same wire length. Otherwise, timing arcs in each tile must be extracted separately. This process can be time consuming and requires eFPGA CAD tools to store more complex databases. In a regular architecture like the island-style architecture with repeated tiles, the same results can be achieved in much less time.

Finally it was observed that a regular layout structure (see Figure E.1) reduced wire lengths by about 21% on average. However, because wire delays make up 10% of the overall delay on average, the wire length reductions translated to a 3% reduction in delay for benchmarks regardless of size. This result is not surprising since the architecture used here has only short wire (single segment length). It is expected that structure will have more of an impact in architectures with long wires

[11]. Nonetheless, this result is significant, because it suggests results similar to those obtained in [14] with a new CAD flow could have been achieved by imposing physical layout structure alone.

6.2.5 Summary

In the final analysis, the proposed eFPGA IP generator method which includes an "open" architecture IP library, individually configurable programmable logic architectures, domain-driven IP generation, and an automated layout generator that relies on *highly optimized custom cell libraries*, lies somewhere between the hard and soft IP approaches described previously, because, it aims to combine only the best properties of both these IP approaches to achieve the best results. We call the proposed approach the ***firm IP*** approach because it is neither truly soft nor hard. In Table 5.2 below, we summarize key distinguishing characteristics of the soft, firm, and hard IP approaches.

Table 6.1: Summary of Soft, Firm and Hard eFPGA implementation methodologies

Soft eFPGA	Firm eFPGA Approach	Hard eFPGA
Behavioral RTL	Structural RTL or GateLevel Netlist	Transistor-Level Design
ASIC flow	Custom ASIC flow	Full-custom flow
Generic standard cells	Custom tactical cells *and* generic standard cells	Full-custom design
Logic Synthesis Required	No Logic Synthesis Required	
Cells free to move	Regular, tiled structure	Regular, fixed-tile structure
Configurable architecture	Fixed architecture	
Flexible size, no fixed shape	Flexible size, shape can be fixed	Fixed size and shape
Mixed with cells used for rest of design (fixed logic)	Designed as a separate core and inserted	
Small Amount of Programmable Logic	Small-to-Medium Amount of Programmable Logic	Medium Amount of Programmable Logic

Chapter 7

Final Conclusions and Future Research Work

7.1 Conclusions

In this thesis, a generic island style eFPGA was successfully implemented using the ASIC flow. In the process, design issues were resolved that before now, excluded this class of eFPGA architectures from being implemented within the ASIC design flow. In particular, a "workaround" was devised to prevent the occurrence of combinational loops in eFPGA fabrics during logic synthesis and functional verification. As a result, new architectures that *could* enter states that create combinational feedback loops can be explored without any design or implementation issues. Furthermore, this solution was leveraged to minimize the occurrence of glitch power dissipation during programming.

Also, a novel technique for implementing I/O for island style eFPGAs that uses the switch blocks around the core periphery to implement I/O was introduced and successfully implemented. Two novel multiplexing switches that improve speed of eFPGA fabrics by as much as 24% for the set of benchmarks considered was also presented. A configuration scheme (similar to that used in commercial eFPGAs) that allows tiles in a fabric to be programmed individually or in a row was described and successfully implemented. This scheme has implications for testability programming power. For example, the ability to program tiles individually facilitates fault diagnosis during testing. Also, because only the clock networks of targeted tiles are activated, switching power in minimized.

In this thesis it was also shown that significant improvements in area and speed could be achieved by using FPGA-centric tactical cells to implement eFPGAs rather generic standard cells. An average

core area reduction of 58% was achieved for the set of benchmarks considered. Compared to area results reported by VPR for a full-custom equivalent, tactical cell based eFPGAs are about 2-3X larger. However, other estimates based on results for commercial fabrics show that this overhead lies somewhere between 1.66X and 3X. An average delay reduction of 40% was also achieved over the same set of benchmarks. These delays were found to be within 10% of a full-custom equivalent.

In addition, results achieved with tactical standard cells were compared to results achieved with a new approach called GILES, which uses non-standard cells and custom tools in a flow similar to the ASIC flow. Results obtained showed that cell densities achievable with tactical standard cells are comparable to those obtained for GILES. Consequently, it was also shown that with tactical standard cells, it is possible to achieve layout densities that are comparable to the GILES approach.

Next, the results obtained here were compared to the product term architecture in [15] [16]. The purpose of this experiment was to measure how having close to optimal building blocks for each architecture might affect conclusions. Since the NANDs and NORs used in the product-term architecture are already close to optimal in most standard cell libraries, this architecture is at an advantage relative to LUT-based architectures that rely more heavily on multiplexers. Multiplexers built with *typical* standard cell libraries are not optimal. In this experiment, it was found that including tactical multiplexers in the logic architecture of an island style ePFGA (equivalent to having NANDs and NORs in product-term blocks) resulted in a delay improvement of about 35% in the LUT-based island architecture relative to the product term architecture. It was also found that without tactical cells, the LUT-based Island architecture is still about 20% faster on average. In terms of area, the island-style architecture is roughly 35% larger. It is expected that this difference

would be even less if all the benchmarks used in [15] were considered. In terms of area-delay product, and for the set of benchmarks circuits considered, both architectures are comparable.

In Chapter 6 it was shown that eFPGAs based on tactical standard cells are competitive with commercial hard eFPGA cores. For the set of benchmarks considered, it was observed that higher *effective* logic densities could be achieved using tactical standard cell ePFGAs. This is possible because tactical cell eFPGAs can be tailored for a given application, and so, the extra routing and logic overcapacity that is associated with the hard eFPGA IP approach is avoided almost completely.

It was also shown using an actual Bluetooth baseband SoC design, that either of the baseband frequency hopping or data encryption modules could be replaced with their eFPGA equivalents without any impact in the overall die area of the Bluetooth SoC. This is possible because both modules together account for less that 0.7 % of the SoC design area. Furthermore, this SoC design is I/O limited and so a significant portion of the area within the I/O ring can be used at no cost.

Further, in Chapter 6, a new paradigm for eFPGA IP design was proposed to minimize the overhead that is normally associated with programmable logic fabrics and possibly make them more attractive to designers. In particular, a system based on an open source architecture library was proposed, so that designers are able to choose the most suitable architecture for a given application. The results in this thesis have already shown that some circuits, for reasons that are not yet fully understood, appear to "prefer" one architecture over another (other factors such as CAD tool algorithms cannot be ruled out as a factor [12]). Other recommended features include configurable architectures, such as the ones used here, and domain-driven IP generation to minimize overhead.

7.2 Future Work

The tactical standard cells developed in this work need to be fully incorporated into the ASIC design flow so that the results obtained in thesis can be further corroborated. Furthermore, it is also expected that this will result in further savings due to reductions in the layout area overhead. In thesis, a pessimistic estimate of layout area overhead for tactical standard cell eFPGAs was assumed.

Techniques to further minimize the SRAM cell area overhead should be investigated further, because SRAM constitute a large proportion of the fabric area, and are relatively easy to design. Data in [25] suggests that as SRAM cell area of roughly $11um^2$ is possible in 180nm process technology.

Given the workaround for combinational loops presented in this thesis, it would be interesting to see the improvements that can be made to previously implemented architectures in [02] [15] [16].

Finally a chip design identical to the one in [14] was implemented using the island architecture in 180nm process. It would therefore be useful to also implement the same design using customized tactical standard cells. This would present the chance to compare the power dissipation in both approaches, since it is difficult to obtain exact estimates of power consumption with available ASIC tools. Moreover, this would also be an excellent opportunity to corroborate current findings.

7.3 Contributions

- A parameterizable island-style architecture was implemented in RTL. In addition, a novel I/O design was developed that reuses excess routing resources around the eFPGA edges.

- A clever workaround was devised for the combinational loop problem that plagued previous architectures implemented in the ASIC flow. As a result, designers can now explore a broader range of architectures. For example, the dual-network architecture [15] could be improved because the work done here implies that an extra routing network in no longer needed. Also, the gradual architecture [02] can now be modified to support sequential logic.

- Two novel switch designs, "Improved multiplexer switch 1" and "Improved Multiplexer switch 2" were introduced. For the set of benchmarks considered, these switches resulted in speedups of between 1% and 24%. However "Improved multiplexer switch 1" results in an area overhead of 3% for generic standard cells and about 13% for tactical standard cells. Although maximum delay savings due to "Improved multiplexer switch 2" are not very high (6%) area overhead in minimal or non-existent compared to the original multiplexer switch.

- The design and implementation of tactical standard cells for the island-style architecture implemented as part of this project resulted in area and delay savings of 58% and 40% respectively. Compared to a full-custom equivalent, our delay results are within 10% and area results are somewhere between 1.66X and 3X based on our estimates and those from VPR

- It was shown that the densities achieved with the tactical standard cells developed as part of this work are comparable to cells used in the GILES. This is possible because a novel technique was devised to improve the layout area efficiency of pass-transistor-based multiplexers. As a result, with the standard tools of the ASIC flow, it was possible to achieve identical results to GILES which used custom-designed tools and a non-standard cell layout.

- It was shown that standard-cell-based architectures are very sensitive to the cells that are available in a library. Using a product term architecture and the LUT-based architecture developed here, it was shown that conclusions can change significantly when each architecture is implemented with cells that are relatively well optimized for each architecture.

- It was shown that an eFPGA fabric based on tactical standard cells developed as part of this work, are competitive with a commercial library of hard eFPGA IP fabrics. In the example considered, the tactical cell approach is better in all cases and for a range of circuit sizes.

- Using an actual "real-world" wireless platform design, it was shown that two modules that lend themselves to programmability, namely, the frequency hopping and data encryption modules, could be embedded in the SoC platform without any increase in the die area.

- It was demonstrated in this thesis that a regular eFPGA architecture has numerous benefits. For example, it was possible to implement a fabric with 784 4-LUTs in relatively short time (in roughly 3 hours versus 17 hours for a flat design) by exploiting the regularity of the island-style eFPGA during synthesis and layout. This approach also resulted in a 21% reduction in wire lengths. Although this only translated into a 3% speedup of fabrics in general, the implication is quite significant, because it suggests the same results obtained in [14] could have been achieved by imposing layout structure to better manage wire lengths.

- A novel paradigm for eFPGA design based on an open architecture library was proposed to enable designers select the best architecture for a given application and minimize overhead.

Appendix A

Standard Cell Based eFPGA implementation Results

A.1

In order to verify the RTL implementation of the island-style architectures described in Chapter 3, a reference design with an embedded core was implemented. This is the same reference design that was used in previous work [14] [77]. The design is a test interface for embedded core testing [76]. In essence, this design is an adaptor that allows an embedded IP core in a SoC design to "plug" into an on-chip test network and receive test packets depending on the destination address of the packet.

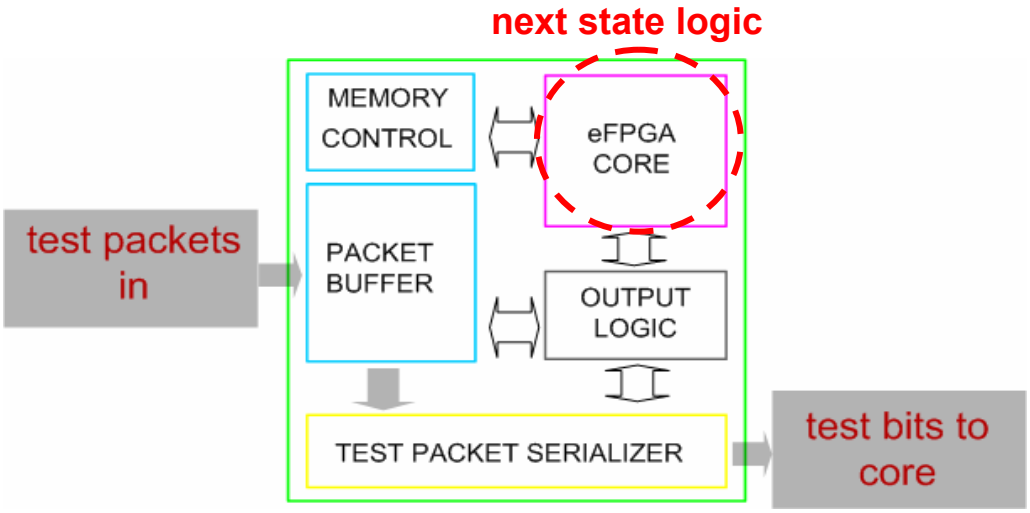


Figure A.1: Block level diagram of the test interface module with an eFPGA fabric

As in previous work [14] the next state logic of the *primary controller* in this interface was replaced with programmable logic, leaving only the output logic of the controller as fixed logic (see Figure A.1). This design was successfully implemented and Figure A.2 shows a screen capture of gate-level simulation waveform for the design. In Figure A.2 the states transition as expected (output of embedded fabric) and the fixed output logic module responds to the state transitions as expected.

Finally, the measured *critical path delay* through the eFPGA core (based on the original multiplexer switch) was 33.2ns and 34.24ns for the entire chip. The eFPGA core area alone was 389,550um^2.

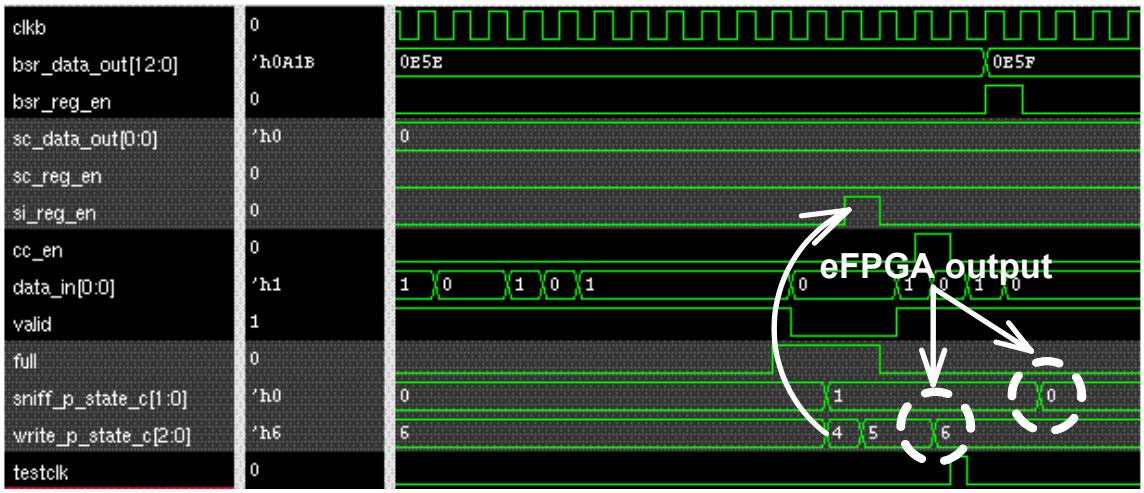


Figure A.2: simulation waveform capture of eFPGA state transitions and response

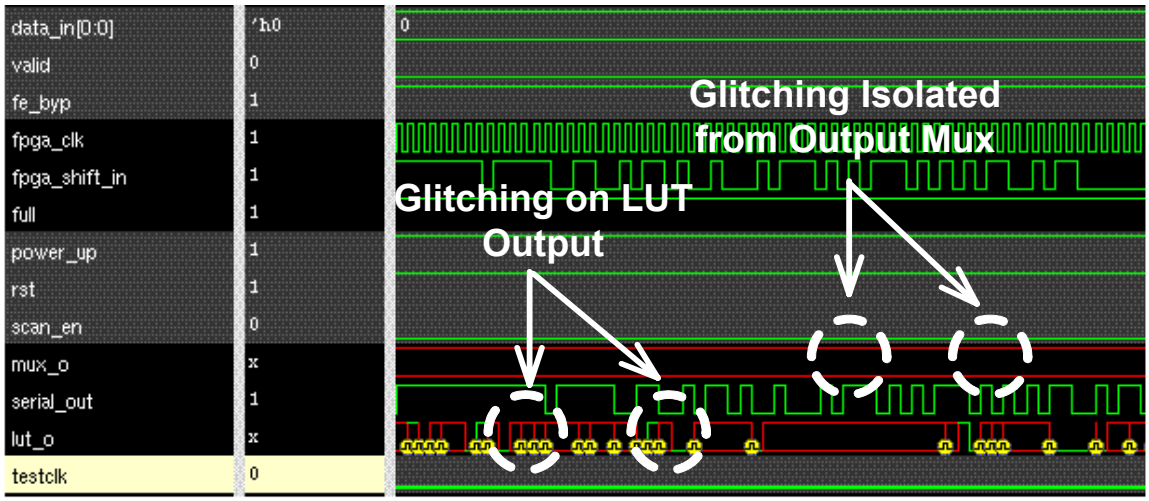


Figure A.3: simulation waveform capture of glitches and glitch isolation in a BLE

Furthermore, all of the architecture, CAD, and design issues highlighted in the previous chapter and in this chapter were resolved. For example, Figure A.3 shows a simulation screen capture from the eFPGA configuration phase. As the figure shows, several transitions and glitches occur at the output of the LUT during configuration as predicted (see Figure 4.7). Figure A.3 also shows that these glitches have been isolated from the multiplexer output ("mux_o" in Figure A.3) using the scheme illustrated in Figure 4.7. The power savings that result from combining this glitch isolation approach

with tile-based and row-based configuration has not been measured here. This is left to future work when the fabricated test interface chip is tested, and measurements of power can be obtained.

A.2

The data presented in Tables A.1 through A.3 were used to generate the plots shown in Chapter 4 of this thesis. Synthesis results were obtained using Synopsys Design Compiler. Layout was done using Cadence First Encounter and critical path delays were obtained using Prime Time Static Timing Analysis tool from Synopsys. All the results are for a 180nm process node standard cell library [24]. All critical path delay results in Tables A.1 through A.3 are for the ***worst case process corner***.

Table A.1: Design results for 18 eFPGA cores with tri-state buffer based switches

Circuit	estimated area (um^2)	synthesis area (um^2)	% area difference	layout area (um^2)	critical path delay (ns)
Cc	439018	471158	7.32	612505	7.74
Cm150a	229030	248074	8.32	322496	8.91
cmb	258836	268481	3.73	349025	8.54
comp	455733	487945	7.07	634328	16.30
cu	317032	300631	-5.17	390820	8.25
5xp1	381288	376979	-1.13	490073	5.14
I1	289621	279649	-3.44	363543	9.66
inc	381288	376979	-1.13	490073	5.40
unreg	600952	635851	5.81	826606	6.88
rd84	2570245	2464741	-4.10	3204163	20.92
apex7	1546660	1519160	-1.78	1974908	11.23
alu2	2792540	2984645	6.88	3880039	23.87
clip	2018818	2113094	4.67	2747022	18.05
9symml	939896	859022	-8.60	1116729	12.79
term1	1249090	1266233	1.37	1646102	15.92
cht	1275526	1353331	6.10	1759330	5.39
ex5p	23446224	22485924	-4.10	29231701	42.00
apex4	37571447	39263454	4.50	51042490	65.37

Table A.2: Design results for 18 eFPGA cores with original multiplexer switches

circuit	Estimated area (um^2)	synthesis area (um^2)	% area difference	layout area (um^2)	critical path delay (ns)
cc	419158	451298	7.67	586687	10.43
cm150a	221086	240130	8.61	312169	10.28
cmb	248244	257889	3.89	335256	9.76
comp	443320	475532	7.27	618192	17.74
cu	306440	290039	-5.35	377051	9.92
5xp1	370696	366387	-1.16	476303	7.61
I1	277208	267236	-3.60	347407	11.30
inc	370696	366387	-1.16	476303	8.29
Unreg	569176	604075	6.13	785298	10.12
rd84	2506527	2401023	-4.21	3121330	27.98
apex7	1483108	1455608	-1.85	1892290	18.37
alu2	2744876	2936981	7.00	3818075	28.73
Clip	1955266	2049542	4.82	2664405	23.69
9symml	929304	848430	-8.70	1102959	14.53
term1	1216817	1233960	1.41	1604148	18.58
Cht	1211974	1289779	6.42	1676713	9.55
ex5p	23128464	22168164	-4.15	28818613	61.59
apex4	36441909	38133916	4.64	49574091	98.27

Table A.3: Design results for 18 eFPGAs using the speedy multiplexer switch 1

Circuit	estimated area (um^2)	synthesis area (um^2)	% area difference	layout area (um^2)	critical path delay (ns)
Cc	434740	466880	7.39	606944	9.07
cm150a	227319	246363	8.38	320272	9.96
Cmb	256554	266199	3.76	346059	8.86
Comp	453059	485271	7.11	630852	15.99
cu	314750	298349	-5.21	387854	9.10
5xp1	379006	374697	-1.14	487107	6.31
I1	286947	276975	-3.48	360067	10.72
inc	379006	374697	-1.14	487107	6.49
unreg	594107	629006	5.87	817708	8.50
rd84	2556519	2451015	-4.13	3186320	24.49
apex7	1532970	1505470	-1.79	1957112	15.52
alu2	2782273	2974378	6.90	3866691	26.19
clip	2005128	2099404	4.70	2729226	19.25
9symml	937614	856740	-8.63	1113763	14.16
term1	1242138	1259281	1.38	1637065	17.51
cht	1261836	1339641	6.17	1741534	7.25
ex5p	23377776	22417476	-4.11	29142719	47.23
apex4	37328135	39020142	4.53	50726185	77.20

Appendix B

Area Model for Standard Cell based eFPGA area Estimation

Area breakdown of a tile in a standard-cell-based island-style embedded programmable logic fabric.

The Routing Switch Block

$Area_{sblock} = 1318 \times W$ (original multiplexer switch), $Area_{sblock} = 1483 \times W$ (tri-buffered switch design)
$Area_{sblock} = 1447 \times W$ (improved multiplexer switch 1 design)

Configurable Logic Block

$$Area_{lutflipflops} = a \times N \times 2^K + Area_{lutinputmultiplexers} = N \times \left(\left\lfloor \frac{2^K - 1}{3} \right\rfloor f + b \left(\mathrm{mod} \frac{2^K - 1}{3} \right) \right) + Area_{lutoutputmux} = N(a) +$$

$$Area_{selflop\ ,lutoutflop} = N(b + c) + Area_{lutinputmuxes} = KN\left(\left\lfloor \frac{2^{\lceil \log_2 (3N+2) \rceil} - 1}{3} \right\rfloor f \right) + KN \left(b \times \mathrm{mod} \frac{2^{\lceil \log_2 (3N+2) \rceil} - 1}{3} \right) +$$

$$Area_{reset\ \log ic\ (2-inputORgat\ es),reset\ \log icflipflop\ s} = 2N(a + d) + Area_{lutinputmu\ ltiplexerflops} = \lceil \log_2 (3N + 2) \rceil KNa$$

Tile Connection Blocks

$$Area_{xtionblkout} = \lceil WF_o \rceil N(t + a + e) + Area_{buffarrayInter} = \lceil WF_i \rceil \left\lfloor \frac{2N+2}{4} \right\rfloor s + \lceil WF_i \rceil \left(\left\lfloor \frac{2N+2}{4} \right\rfloor + \left\lfloor \frac{(2N+2) \bmod 4}{3} \right\rfloor \right) s +$$

$$Area_{buffarrayIntra} = \lceil WF_i \rceil \left\lfloor \frac{2N+2}{4} \right\rfloor q + \lceil WF_i \rceil \left(\left\lfloor \frac{2N+2}{4} \right\rfloor + \left\lfloor \frac{(2N+2) \bmod 4}{5} \right\rfloor \right) q +$$

$$\lceil WF_i \rceil \left\lfloor \frac{2N+2}{4} \right\rfloor q + \lceil WF_i \rceil \left(\left\lfloor \frac{2N+2}{4} \right\rfloor + \left\lfloor \frac{(2N+2) \bmod 4}{6} \right\rfloor \right) q + Area_{clbinputmu\ xflops} = \lceil \log_2 \lceil WF_i \rceil \rceil a(2N + 2) +$$

$$Area_{clbinputmu\ xes} = (2N + 2) \times \left(\left\lfloor \frac{2^{\lceil \log_2 \lceil WF_i \rceil \rceil} - 1}{3} \right\rfloor f + b \times \mathrm{mod} \frac{2^{\lceil \log_2 \lceil WF_i \rceil \rceil} - 1}{3} \right)$$

Area of Peripheral Blocks

$$Area_{buffarray\,left+bottom} = \lceil WF_i \rceil \left\lfloor \frac{2N+2}{4} \right\rfloor s + \lceil WF_i \rceil \left(\left\lfloor \frac{2N+2}{4} \right\rfloor + \left\lfloor \frac{(2N+2) mod 4}{3} \right\rfloor \right) s + Area_{switches\,left+bottom} = 2770W +$$

$$Area_{cornerblk} = 11080 \; Area_{tri\text{-}buffs\,left} = \lceil WF_o \rceil \left(\left\lfloor \frac{N}{4} \right\rfloor + \left\lfloor \frac{(N) mod 4}{3} \right\rfloor \right) t + Area_{tri-buffs\,bot} = \lceil WF_o \rceil \left(\left\lfloor \frac{N}{4} \right\rfloor \right) t +$$

$$Area_{clk\,logic\,left+bottom} = 6 \times r + Area_{sel\,logic\,left+bottom} = 2m +$$

$$Area_{tri\text{-}buffs\;\;left\,flip\;\;flops} = \lceil WF_o \rceil \left(\left\lfloor \frac{N}{4} \right\rfloor + \left\lfloor \frac{(N) \bmod 4}{3} \right\rfloor \right) a + Area_{tri-buffs\;\;bot\;\;flipflops} = \lceil WF_o \rceil \left(\left\lfloor \frac{N}{4} \right\rfloor \right) a$$

Configure and Clock logic

$$Area_{regulartile\;clk\,logic} = 3 \times r + Area_{regulartileselectlogic} = m + Area_{edgetile\;clk\,logic} = 3 \times r + Area_{edgetileselectlogic} = m$$

a = cell area of program flop; b = average cell area of 2:1 CMOS multiplexer; c = cell area of LUT output flip-flop;

d = area of 2-input OR drive 1; e = area of 2-input OR drive 2; f = area of 4:1 CMOS multiplexer;

q = area of intratile track buffers; s = area inter tile track buffers; t = average tri-state buffer cell area;

r = area of clock gating logic 2 input AND; m = area of tile selection logic OR2D1 + MUX2D1;

Appendix C

Details of Circuit Analysis Solutions for Multiplexer Circuits

C.1

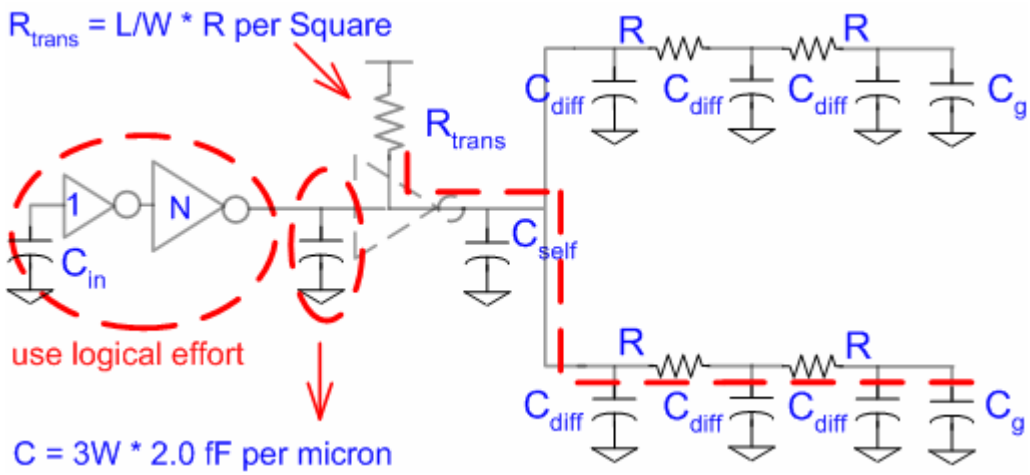


Figure C.1: RC network representation of shared buffering scheme in an eFPGA

$$Delay = (B-1)\left[(D-1)R_{trans}\,2C_{diff} + R_{trans}\left(C_{diff} + C_g\right)\right] + Elmore\ Delay\ of\ active\ path \quad \text{(C.0)}$$

$$Delay\ of\ active\ path\ = R_{trans}\left(2C_{diff} + C_{self}\right) + \left(R_{trans} + R\right)2C_{diff} + \left(R_{trans} + 2R\right)\left(C_{diff} + C_g\right) \quad \text{(C.1)}$$

Equation (C.1) is based on the general Elmore delay equation: $\sum_{i=1}^{D+1} C_i \sum_{j=1}^{i} R_j$ where D is the depth of the pass transistor tree, i is the number of nodes from source to sink, j is the number of resistors from the source to the current node (i^{th} node), and B is the number of branches in the pass tree.

Combining Equations (C.0) and (C.1) with $B, D = 2$, $R\ per\ square = 27\ K\Omega$, $\frac{W}{L} = \frac{0.5um}{0.18um}$, $R_{trans} = \frac{L}{W} \times R\ per\ square$, $C_{diff} = C_{eff} W_{pass}, C_g = C_g^* 3W_{\min}$; $C_{eff}, C_g^* = 1.0,\ 2.0 \frac{\text{fF}}{\text{um}}$ and solving for W (width of PMOS transistor network) in Figure 5.11 gives the results in Equation (C.2) below:

$$Delay_{target} = \frac{49\times10^{-18}}{W} + 78\times10^{-12} \;\therefore \text{W (of PMOS in buffer)} = \frac{49\times10^{-18}}{Delay_{target} - 78\times10^{-12}}\,\text{m} \qquad \text{(C.2)}$$

The value of W obtained in Equation (C.2) depends on the ***required target delay*** ($Delay_{target}$). Therefore, based on the results in Equation (C.2), this target delay must be greater than $78\times10^{-12}\,s$ because, this represents the smallest delay that is achievable in the design example considered here.

C.2

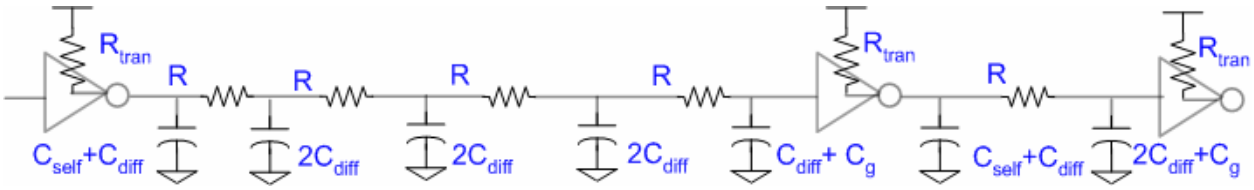


Figure C.2: RC network representation of repeater insertion in an eFPGA 5-LUT

$$Delay = \frac{n}{m}\left(R_{tran}\left[C_{self} + C_{diff} + (m-1)2C_{diff} + C_{diff} + C_g\right] + R\left(C_{diff} + C_g\right)m + \frac{m(m-1)}{2}RC_{diff}\right) \text{(C.3)}$$

Where m is the number of NMOS pass transistors between repeaters and n is the depth of the pass transistor tree. This translates to $\frac{m}{n}$ sections of m pass transistors for the entire pass transistor tree.

Setting $\frac{\partial Delay}{\partial m} = 0$ gives $\frac{-n}{m^2}R_{tran}\left(C_{self} + C_g\right) + \frac{n}{2}RC_{diff} = 0$ (C.4)

Rearranging equation (5.4) and solving for m gives $m_{opt} = \sqrt{\frac{2R_{tran}\left(C_{self} + C_g\right)}{RC_{diff}}}$ (C.5)

Where m_{opt} is the optimal number of pass transistors needed between repeaters to minimize delay.

Finally, Equation (C.5) evaluates to approximately 4 pass transistors if a $\frac{W_{nmos}}{W_{pmos}}$ ratio of 1.5 is used.

Notice that this analytical result is identical to the result obtained experimentally through SPICE.

Appendix D

Area estimation method for GILES and Full-custom eFPGAs

D.1

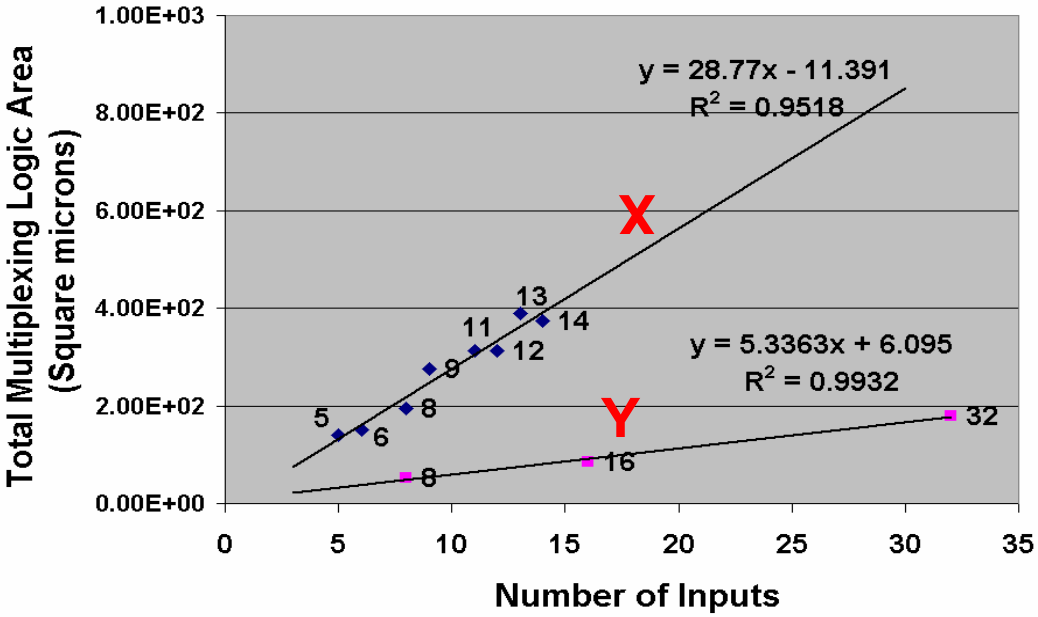


Figure D.1: plot of layout area vs. inputs for general-purpose eFPGA multiplexers

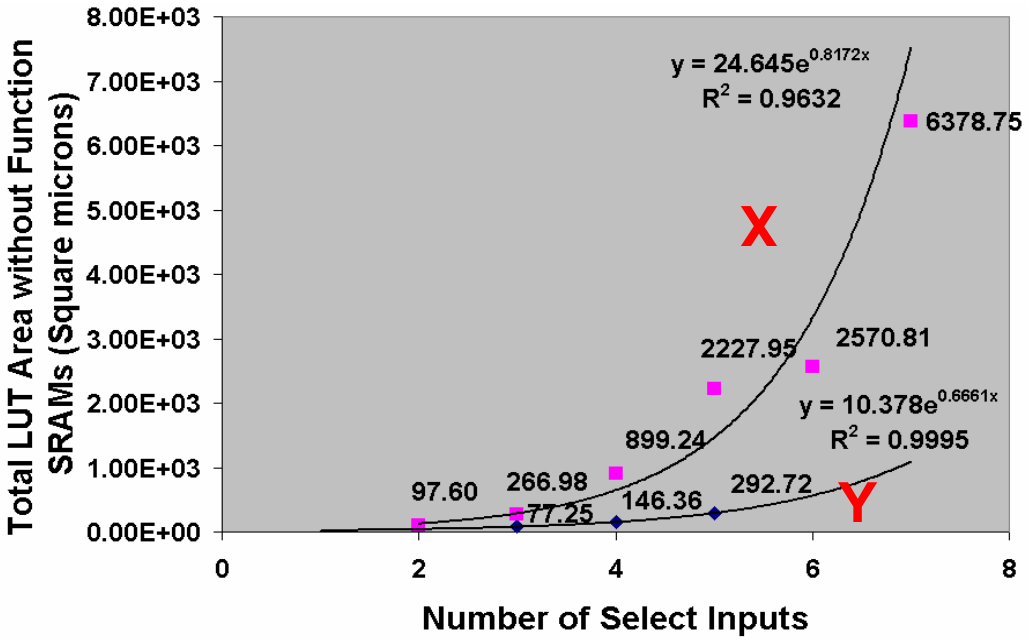


Figure D.2: Plot of layout area (excluding the SRAMs) vs. select inputs for LUT

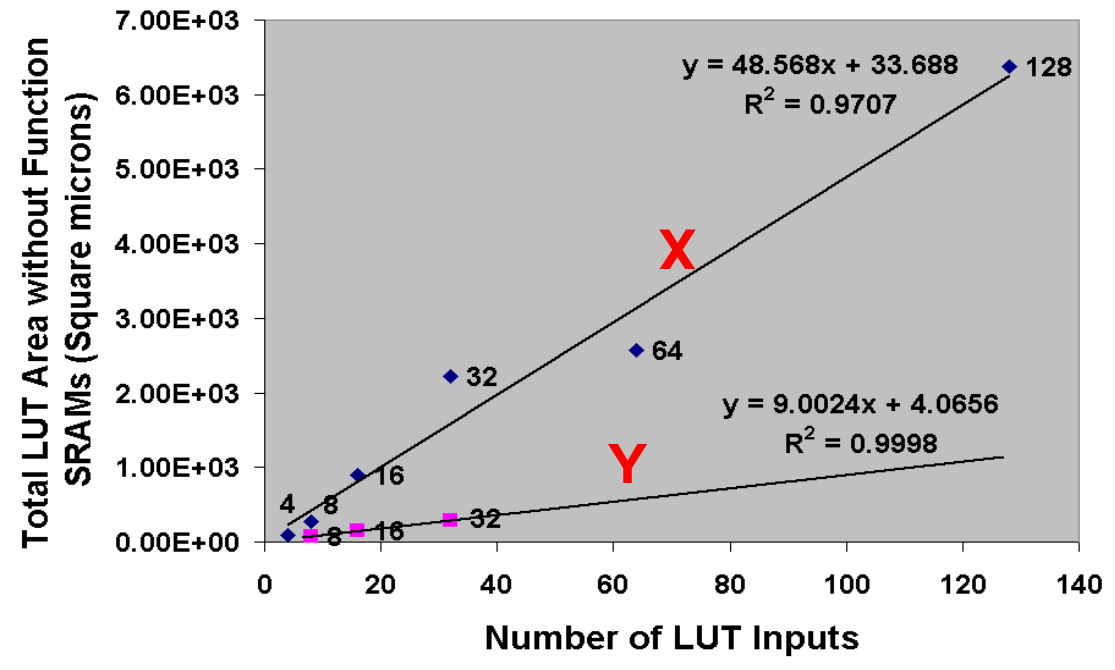


Figure D.3: Plot of layout area (excluding the function SRAMs) vs. inputs for LUT

D.2

Earlier area estimates obtained from the VPR tool show that a tactical standard cell implementation of a generic island-style eFPGA architecture is about 2.86X the size of the equivalent full-custom implementation. However, the results in [68] [69] have suggested the area overhead could be as low as 1.36X. We next propose an alternative method to estimate the area overhead of a standard cell eFPGA relative to a full-custom device and then compare our results with previous estimates.

To improve the quality of results obtained, it was necessary to consider a larger set of benchmark circuits. In particular, it was necessary to add more circuits from the "golden 20" MCNC benchmarks so that a more realistic area estimates might be obtained. This is needed because the very large benchmarks in the MCNC suite require a much larger routing infrastructure compared to the logic components (LUTs). In other words, the routing infrastructure of an FPGA scales at a faster rate than the logic with increasing size of the benchmarks. A large routing interconnect degrades the area efficiency of programmable devices. Therefore, in order to capture this effect in our estimates, it was necessary to include more golden 20 benchmarks to the initial set of benchmarks used in this work. Figure D.4 shows the distribution of logic and routing for the standard cell implementations of all the benchmarks considered. The general trend in this figure confirms that routing in more dominant as the size of logic implemented in an eFPGA increases.

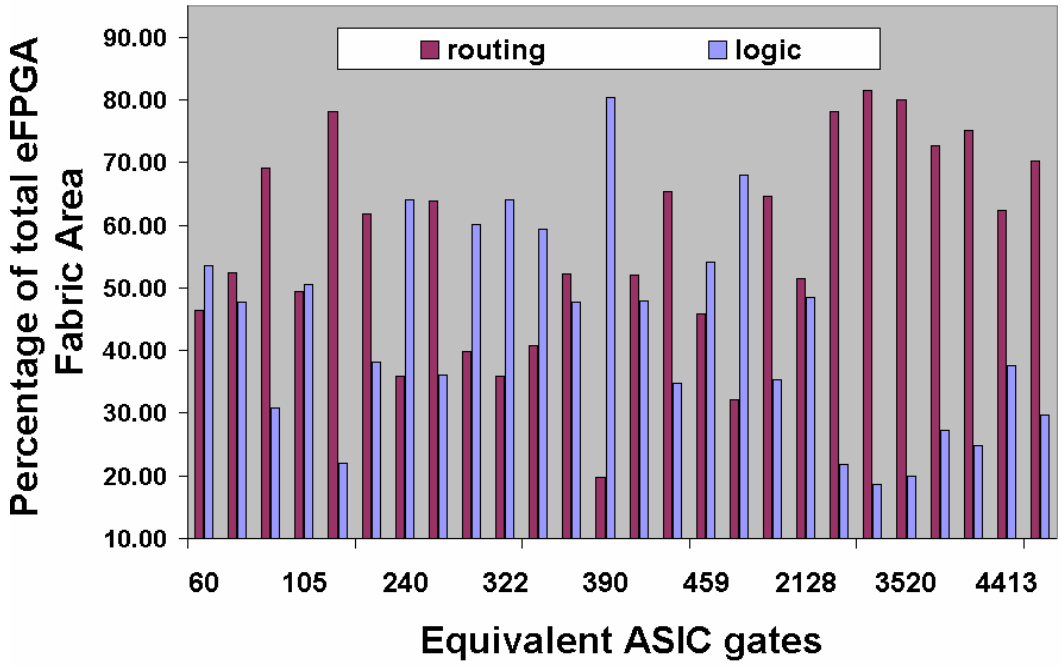


Figure D.4: Area distribution between logic and routing components of eFPGA

The proportion of routing and logic was more or less split evenly for smaller benchmarks (less than 2128 ASIC gates in Figure D.4). For the larger benchmarks, routing made up 70% of the total area.

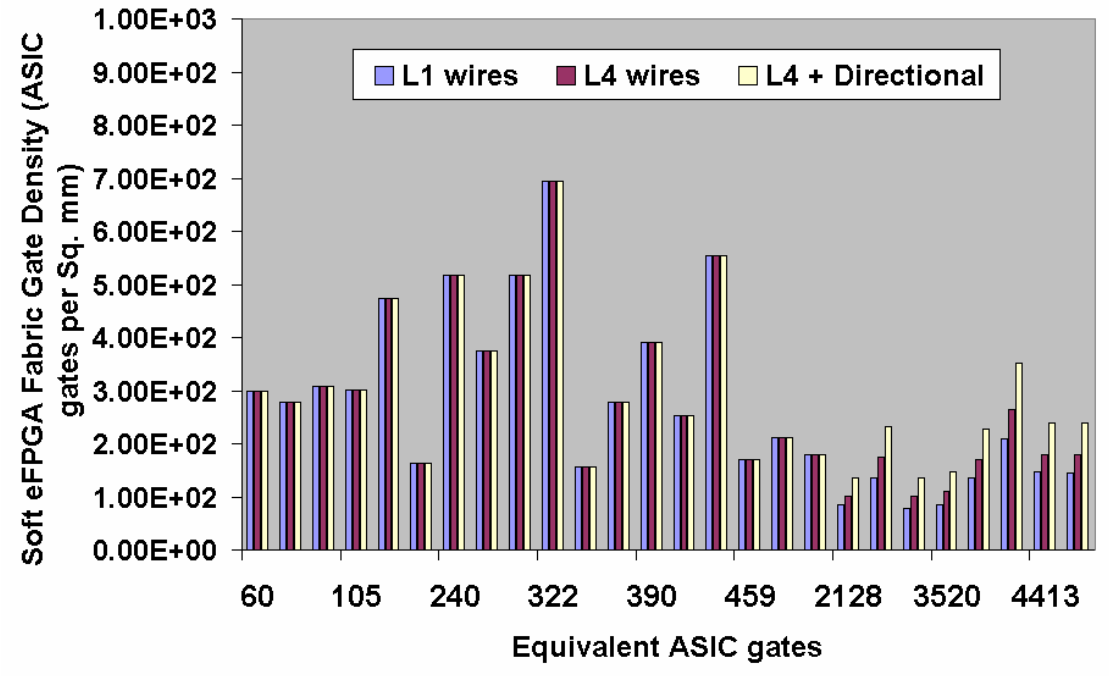


Figure D.5: ASIC gate density scaling after estimated improvements to routing

Figure D.5 shows the ASIC gate density (the equivalent ASIC gates in an ASIC implementation of a benchmark circuit divided by the area in mm^2, of smallest eFPGA needed to implement the circuit) for the standard cell eFPGAs used to implement each benchmark. "L1" in Figure D.5 refers to an island architecture with single segment or "length 1" [11] wires. "L4" refers to an island architecture with "length 4" wires and "L4 + Directional" refers to island architectures with "length 4" directional wires as described in [20]. As expected, the density of the eFPGA fabrics degrades as the circuit size increases. For example, in L1 eFPGAs, the equivalent ASIC gate density drops from 304 ASIC gates per mm^2 for the smaller benchmarks to 121 ASIC gates per mm^2 for larger benchmarks. This drop is made worse by the use of L1 wires because larger circuits require longer wires than L1 wires and so restricting the architecture to L1 wires introduces excess routing logic in the eFPGA.

To correct for the inefficiency introduced by L1 wires, all wires were changed to L$ wires. However, because our current implementation of the generic island architecture supports only L1 wires, a good estimate of the effect of switching from L1 to L4 wires was needed. Estimates for the area savings that result from changing from L1 to L4 wires is provided in [11] for the same set of golden 20 benchmarks used here. Based on the results in [11] an average overall area saving of 17% is achieved. This correction has been applied to large circuits alone since smaller benchmarks do not benefit much from L4 wires. Our change to L4 wires is a valid one because most modern island-style architectures use an abundance of L4 wires. Furthermore, a commercial eFPGA [41] used in this comparison is based on the Virtex-II architecture [80] which has an abundance of L4 wires.

Next, directional wires were factored into the architecture because recent eFPGA architectures (including the Virtex-II) use directional wires as opposed to bidirectional wires. Work done in [20] gives an estimate of total area savings (25%) due to directional L4 wires versus bidirectional L4

wires. After accounting for L4 and directional wires, the average ASIC gate density of the eFPGA fabrics used to implement the largest MCNC benchmarks improved to 203 ASIC gates per mm^2. With the inclusion of tactical cells the density rises to 483 ASIC gates per mm^2 for large circuits.

To make comparisons to a full-custom eFPGAs, we use the gate densities reported in [45] as well as detailed information in [41] about the ASIC gate capacity of specific core sizes. For example, in [41] the equivalent ASIC gate capacity for three core sizes (including core area) is provided. From this data it is easy to calculate the equivalent ASIC density for each core. Although this data is for a 90nm process, estimates for 180nm process are easily obtained. Table D.1 shows area overhead results (scaling factors) for tactical cell eFPGAs relative to full-custom eFPGAs in 180nm process.

Table D.1: estimated area overhead for tactical cell eFPGA relative to full-custom

Circuit range	relative to IBM-LOW	Relative to IBM-MEAN	relative to IBM-HIGH	relative to eASIC estimate
All circuits	1.24	1.58	2.12	2.32
small only	1.09	1.39	1.87	2.04
Large only	1.66	2.11	2.84	3.11

Table D.1 shows scaling factors relative to industry estimates of equivalent ASIC gate densities for full-custom FPGAs. Based on data provided for three cores by IBM and Xilinx in [41], their smallest eFPGA core has the lowest equivalent ASIC gate density (800 ASIC gates per mm^2). This is called "IBM-LOW" in Table D.1. The biggest of the three cores has the highest equivalent ASIC gate density (1371 ASIC gates per mm^2). This is referred to as "IBM-HIGH" in Table D.1. The next biggest core has an equivalent ASIC gate density of 960 ASIC gates per mm^2. The average of all three gate densities is referred to as "IBM-MEAN" in Table D.1. The ASIC gate density quoted by eASIC in [45] is 1500 ASIC gates per mm^2. This is the number used to calculate the overhead in the

last column of Table D.1. The calculated overhead for only the small benchmark circuits in Table D.1 is not considered reflective of the true scaling and is presented simply for reference purposes.

The scaling factors when only large circuits are considered, reveals some interesting trends. For example, the overhead relative to IBM-HIGH is almost identical to the VPR estimate obtained earlier. The overhead relative to the eASIC estimate is also close although it is not clear what type of architecture was used to obtain this number. This is an important factor in these estimates because achievable densities are sensitive to the type of architecture. For example, in [04] densities higher than 1500 ASIC gates per mm^2 have been reported. However, because the architecture is quite different from the island-style architecture used in our analysis, it has not been used in here.

The comparison to IBM-LOW for large circuits is closer to estimates obtained in [68] for the GILES implementation of the Virtex-E architecture. The difference could be due to the fact that in our estimates we have not taken full advantage of the interconnect richness in the Virtex-II architecture that helps keep area down and density high. On the other hand, the GILES implementation copied almost exactly the interconnect of the Virtex-E and so achieves higher density. However, based on the data from our analysis we place the area overhead of a tactical standard cell eFPGA relative to a full-custom equivalent at somewhere between 1.66X and 2.84X.

Appendix E

A sample structured eFPGA layout and Bluetooth SoC Floorplan

E.1

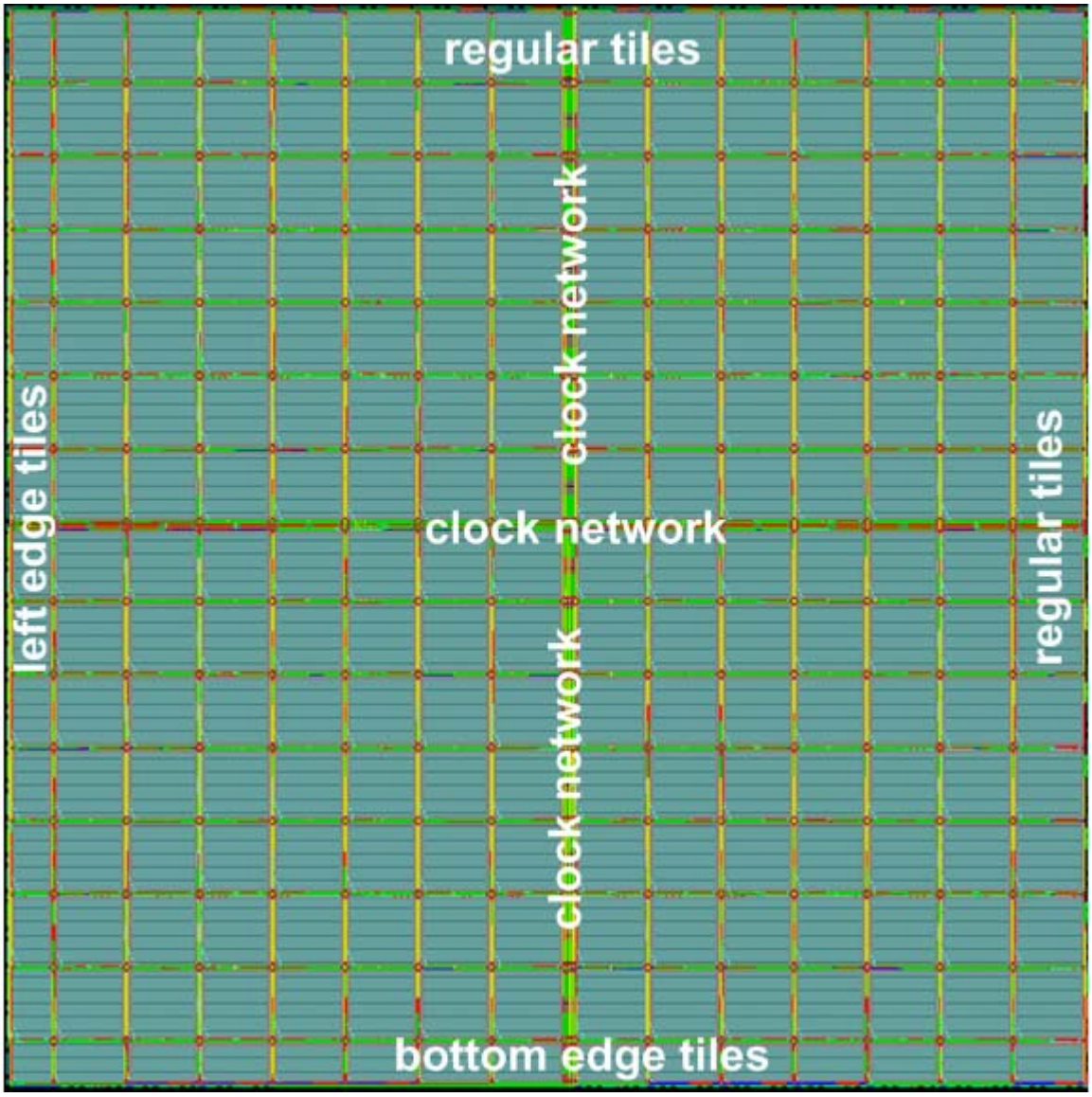


Figure E.1: Structured eFPGA layout for Bluetooth Baseband Encryption Module

E.2

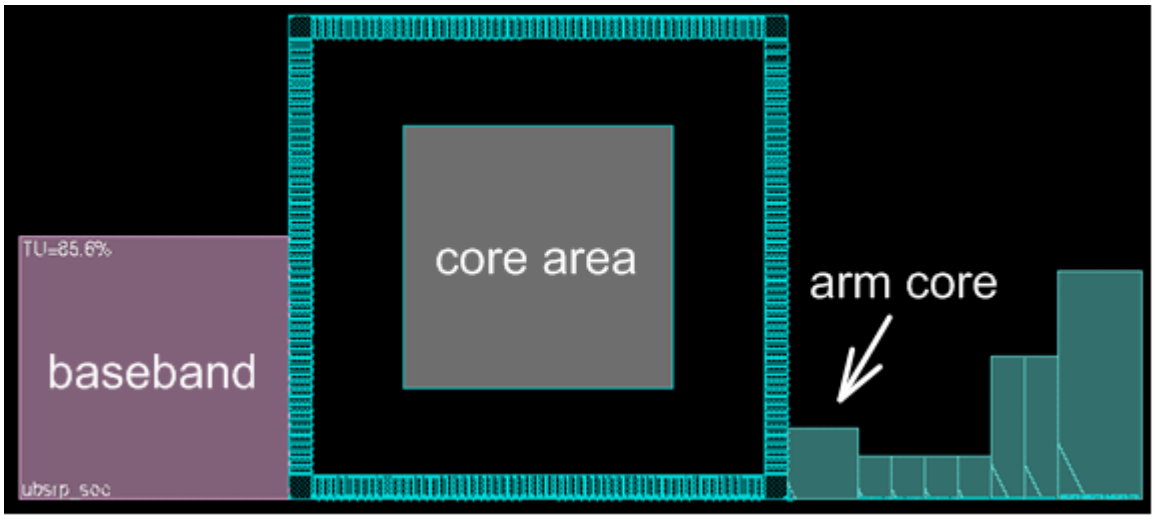


Figure E.2: Bluetooth baseband SoC showing underutilized area around the core

Bibliography

[01] S. Phillips, S. Hauck “Automatic Layout of Domain specific Reconfigurable Subsystems for System-on-Chip” FPGA 2002.

[02] 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, Feb 2003.

[03] S.J.E. Wilton, R. Saleh, “Programmable Logic IP Cores in SoC Design: Opportunities and Challenges”, Proceedings of the 2001Custom Integrated Circuits Conference, pp. 63-66.

[04] 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.

[05] T. Vaida, “PLC Advanced Technology Demonstrator TestChipB”, Proceedings of the 2001 Custom Integrated circuits conference, pp. 67-70, May 2001.

[06] S. Knapp, D. Tavana, “Field configurable system-on-chip device architecture” Proceedings of the IEEE 2000 Custom Integrated Circuits Conference, pp. 155–158, May 2000.

[07] Keating, Michael, and Pierre Bricaud. Reuse Methodology Manual. Boston: Kluwer Academic Publishers, 1999.

[08] M2000, Inc, “M2000 FLEXEOStm Configurable IP Core”, http://www.m2000.fr.

[09] E. Ahmed and J. Rose, “The effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density”, ACM International Symposium on Field-Programmable Gate Arrays, pp. 3-12, 2001.

[10] Altera Corp., *Stratix II Device Handbook*, Vol. 1, San Jose, CA, pp. 2-6 – 2-7, 2004

[11] V. Betz, J. Rose, and A. Marquardt, “Architecture and CAD for Deep-Submicron FPGAs”, Kluwer Academic Publishers, 1999.

[12] 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*, Monterey, CA, Feb. 2002, pp. 147-156.

[13] 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.

[14] J.C.H. Wu, "Implementation Considerations for "Soft" Embedded Programmable Logic Cores", M.A.Sc. thesis, October 2004.

[15] A. Yan, S.J.E. Wilton, “Sequential Synthesizable Embedded Programmable Logic Cores for System-on-Chip”, IEEE Custom Integrated Circuits Conference, Orlando, FL, October 2004

[16] A. Yan, S.J.E. Wilton, “Product Term Embedded Synthesizable Logic Cores”, in the IEEE International Conference on Field-Programmable Technology, Tokyo, Japan, Dec. 2003, pp. 162-169

[17] A. Yan, "Product-Term Based Synthesizable Embedded Programmable Logic Cores", M.A.Sc. thesis, January 2005.

[18] “ASAP”, http://www.viragelogic.com/upload/documents/product_broch_asap_mem4.pdf

[19] Victor O. Aken’Ova, “A Parallel Core Access Interface for Test”, EECE 578 Internal Report, April 2002.

[20] G. Lemieux, E. Lee, M. Tom. A. Yu, “Directional and Single-Driver Wires in FPGA Interconnect”, IEEE International Conference on Field Programmable Technology, December 2004.

[21] D. Lewis et al, “The Stratix II logic and Routing Architecture”, in the *ACM International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, Feb. 2005, pp. 14-20.

[22] A. Marquardt, V. Betz, and J. Rose, “Using Cluster-based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density”, ACM International Symposium on Field-Programmable Gate Arrays, pp. 37-46, February 1999.

[23] “SoC Encounter Data Sheet”, http://www.cadence.com/datasheets/socencounter_ds.pdf

[24] Virtual Silicon Technology Inc., “Silicon Ready Product Information Diplomat-18 High Performance Standard Cells”, Sunnyvale, CA, 1998.

[25] D. A. Hodges, H. G. Jackson and R. Saleh, *Analysis and Design of Digital Integrated Circuits*, Third Edition, McGraw-Hill, 2003

[26] K. Poon, A. Yan, S.J.E. Wilton, “A Flexible Power Model for FPGAs”, to appear in 12th International Conference on Field-Programmable Logic and Applications, September 2002

[27] I. Kuon, A. Egier, J. Rose, “Design, Layout and Verification of an FPGA using Automated Tools”, in the ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, pp. 215 – 226, February 2005

[28] J. Lamoureux, S.J.E. Wilton, “On the interaction between power-aware FPGA CAD Algorithms”, in the International Conference on Computer Aided Design, pp. 701 – 708, November 2003

[29] A. Singh, and M. Malgorzata, “Efficient Circuit Clustering for Area and Power Reduction in FPGAs”, Proc. ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, pp. 59-66, February 2002.

[30] C. Souza, 2002, “IBM and Xilinx advance ASIC design”, Electronic supply and Manufacturing. http://www.my-esm.com/showArticle?articleID=2915800

[31] H. Tennakoon, Carl Sechen, “Gate Sizing Using Lagrangian Relaxation Combined with a Fast Gradient-Based Pre-Processing step” IEEE/ACM International Conference on Computer Aided Design, pp. 395 – 402, 2002

[32] R. Mahmud, “An FPGA primer for ASIC designers”, EEdesign 2004. http://www.eedesign.com/article/showArticle.jhtml?articleId=18901725

[33] K. Compton, “Research Focuses on application-specific reconfigurable blocks”, EE Times 2002. http://www.eet.com/article/showArticle.jhtml?articleId=18307591

[34] A. Marquardt, V. Betz, J. Rose, Speed and Area Tradeoffs in Cluster-based FPGA Architectures, IEEE Transactions on VLSI Systems, pp 84 – 93 vol 8, Feb. 2000

[35] E. Yoneno, P. Hurat, “Power ans Performance Optimization of Cell-based Designs with Intelligent Transistor Sizing and Cell creation” IEEE Electronic Design Processes Workshop, IEEE Press, Piscataway, N.J., 2001, pp. 155-162.

[36] P. Holmberg, “Domain-Specific Platform FPGAs”, FPGA Journal. http://www.fpgajpurnal.com/articles/platform_xilinx.htm

[37] C. Rowen, “Configurability or Reconfigurability”, EE Times 2002. http://www.eetimes.com/article/showArticle.jhtml?articleId=18307590

[38] J Gabay, “Technology Advances Reshape Programmable Logic Offering”, EE Product Center 2004. http://www.eeproductcenter.com/showArticle.jhtml?articleId=20300674

[39] J. Gabay, “Second phase of FPGA and ASIC alternative positions itself to offer the best of bothe Worlds” EE Product Center June 7, 2004. http://www.eeproductcenter.com/showArticle.jhtml?articleId=20300674

[40] K. Padalia, R. Fung, M. Bourgeault, A. Egier, J. Rose “Automatic Transistor and Physical Design of FPGA Tiles From An Architectural Specification”, FPGA 2003.

[41] P. Zuchowsky et. al “A Hybrid ASIC and FPGA architecture”, ICCAD 2002, pp187–194

[42] J. Rubenstien, P Penfield, M. Horowitz, “Signal Delay in RC Networks” IEEE Transactions on Computer Aided Design, vol. CAD-2, pp 202-211, July 1983

[43] E. Elmore, “The transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers” Journal of Applied Physics, pp 55-63, January 1948

[44] Actel Corp., "Varicore™ Embedded Programmable Gate Array Core 0.18mm Family", Tech. Datasheet Rel. 2.2, Dec. 2001.

[45] L. Cooke, "Use of Configurable Cores in Platform based SoCs, eASIC corporation white paper, eASIC web site, 2004.

[46] W. Dally, A. Chang, "The Role of Custom Design in ASIC Chips", Design Automation Conference 2000.

[47] S. Yang, "Logic Synthesis and Optimization Benchmarks, Technical Report Version 3.0", MCNC, 1991

[48] D. Chinnery, K. Keutzer,, Closing the Gap between ASIC and Custom, Kluwer Academic Publishers, 2002.

[49] G. Lemieux, D. Lewis, Design of Interconnection Networks for Programmable Logic, Kluwer Academic Publishers, 2003.

[50] V. Betz, J. Rose, A. Marquardt, Architecture and CAD for Deep Submicron FPGAs, Kluwer Academic Publishers, 1999.

[51] Y. Khalilollahi, "What platform ASICs are and when to use them", EEdesign 2005. http://www.eedesign.com/article/showArticle.jhtml?articleId=57700299

[52] L. Simsic, "Accelerating algorithms in hardware", Embedded.com, 2004. http://www.embedded.com/showArticle.jhtml?articleId=17500157

[53] D. Bhattacharya, "Design-Specific standard Cells yield custom performance", EEdesign, 2004. http://www.eedesign.com/article/showArticle.jhtml?articleId=20301126

[54] R. Wilson, "The constantly shifting promise of reconfigurability", EE Times 2002. http://www.eet.com/article/showArticle.jhtml?articleId=18307592

[55] K. Scott, K. Keutzer, "Improving Cell Libraries for Synthesis", IEEE Custom Integrated Circuits Conference, 1994 pp. 128 – 131

[56] P. Chow, S. Seo, J. Rose, K. Chung, G. Paez-Monzon, "The Design of a SRAM-Based Field Programmable Gate Array – Part II: Circuit Design and Layout", IEEE Transactions of Very Large Scale Integration Systems vol. 7, No. 3, Sept. 1999.

[57] P. Chow, S. Seo, J. Rose, K. Chung, G. Paez-Monzon, "The Design of a SRAM-Based Field Programmable Gate Array – Part I: Architecture", IEEE Transactions of Very Large Scale Integration Systems vol. 7, No. 3, Sept. 1999.

[58] P. Osler, "Placement Driven Synthesis Case Studies on Two Sets of Chips: Hierarchical and Flat", ISPD April 2004 pp. 190 – 197

[59] R. Panda, A. Dharchoudhury, T. Edwards, J. Norton, D. Blaauw, “Migration: A new technique to improve synthesized designs through incremental customization”, Design Automation Conference June 1998 pp. 338 – 391.

[60] L. Pileggi, H. Schmidt, A. J. Strojwas, P Gopalakrishnan, V. Kheterpal, A. Koorapaty, C. Patel, V Rovner, K.Y. Tong, “Exploring Regular Fabrics to Optimize performance – Cost Trade-Off”, Design Automation Conference June 2003 pp. 782- 787.

[61] C. Chen, C. Chu, D. F. Wong “Fast and Exact Simultaneous Gate and Wire Sizing by Lagrangian Relaxation”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 18, No. 7, July 1999

[62] R. Wilson, “Panel debates merits of embedded FPGA megacells”, EE Times UK 2001. http://www.eetuk.com/datenet/news/showArticle.jhtml?articleID=17407115

[63] K. Atasu, L. Pozzi, P. Ienne, “Automatic Application-Specific Instruction-set Extensions Under Microarchitectural Constraints”, Design Automation Conference June 2003

[64] M. Wang, A. Ranjan, A. Raje, “Multi-Million Gate FPGA Physical Design Challenges”, IEEE Conference on Computer Aided Design November 2003, pp. 891 – 898.

[65] B. Zahiri, “Structured ASICs: Opportunities and Challenges”, International Conference on Computer Design 2003

[66] D. J. M. Rabaey, “Digital Integrated Circuits: A design perspective”, First Edition, Prentice Hall, Inc., 1996

[67] V. Betz and J. Rose, “Using Architectural Families to Increase FPGA Speed and Density”, “ACM/SIGDA International Symposium on Field-Programmable Gate Arrays”, Monterey, CA, 1995, pp. 10 - 16

[68] Ian Kuon, Aaron Egier, and Jonathan Rose, "Transistor Grouping and Metal Layer Trade-Offs in Automatic Tile Layout of FPGAs", Poster, International Symposium on Field Programmable Gate Arrays (FPGA), Feb 2004.

[69] Automated FPGA Design, Verification and Layout" Ian Kuon, M.A.Sc. Thesis, University of Toronto, 2004

[70] "Enhancing and Using an Automatic Design System for Creating FPGAs" Aaron Egier M.A.Sc. Thesis, University of Toronto, 2004.

[71] Partha Pratim Pande, Cristian Grecu, André Ivanov, Res Saleh, "Design of a Switch for Network on Chip Applications," IEEE International Symposium on Circuits and Systems, ISCAS 2003, Vol. V, pp. 217-220, Bangkok, Thailand.

[72] Spectrum Signal Processing's SDR-3000 Digital Transceiver Subsystem http://www.rapidio.org/news/newsletter/2002/10/01/news

[73] Canadian Microelectronics Corporation, "Tutorial on CMC's Digital IC Design Flow" Document ICI-096, Part of Tutorial Release V1.3, May 2001.

[74] The Wireless directory of Bluetooth Products and services http://www.thewirelessdirectory.com/Bluetooth-Overview/Bluetooth-Specification.htm

[75] Synopsys, "Chapter 8 Optimizing the Design", *Design Compiler User Guide*, version2002.05

[76] Mohsen Nahvi, Andre Ivanov, "A Packet Switching Communication-Based Test AccessMechanism for System Chips", IEEE European Test Workshop, 2001, pp. 81 – 86

[77] 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 CICC 2003

[78] M. Satarini, "Prolific Improves standard cell generator", EEdesign 2002. http://www.eedesign.com/article/showArticle.jhtml?articleId=17407697

[79] Xilinx Virtex-E 1.8V Field Programmable Gate Array product specification July 2002 DS002-1 (v2.3)

[80] Xilinx Virtex-II Platform FPGAs: Complete Data Sheet Product specification June 24 2004 DS031 v3.3