ARCHITECTURES AND ALGORITHMS FOR
SYNTHESIZABLE EMBEDDED PROGRAMMABLE
LOGIC CORES

by

Noha Kafafi
B.A.Sc, University of British Columbia, 2002

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

Master of Applied Science

in

The Faculty of Graduate Studies
Department of Electrical and Computer Engineering

We accept this thesis as conforming to the required standard:

The University of British Columbia

November 2003

© Noha Kafafi, 2003
Library Authorization

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Name of Author (please print)  
NOHA KAFAFI

Date (dd/mm/yyyy)  
22/01/04

Title of Thesis: Architectures and algorithms for synthesizable embedded programmable logic cores

Degree: MASC  
Year: 2004

Department of Electrical and Computer Engineering

The University of British Columbia  
Vancouver, BC Canada
ABSTRACT

ARCHITECTURES AND ALGORITHMS FOR SYNTHESIZABLE
EMBEDDED PROGRAMMABLE LOGIC CORES

As integrated circuits become more and more complex, the ability to make post fabrication changes will become more and more attractive. This ability can be realized using programmable logic cores. Currently, such cores are available from vendors in the form of a "hard" layout. In this thesis, we focus on an alternative approach: vendors supply a synthesizable version of their programmable logic core (a "soft" core) and the integrated circuit designer synthesizes the programmable logic fabric using standard cells. Although this technique suffers increased speed, density and power overhead, the task of integrating such cores is far easier than integrating "hard" cores into an ASIC. For very small amounts of logic, this ease of use may be more important than the increased overhead. This thesis presents three synthesizable programmable logic core architectures. The place and route algorithms developed for the various architectures are also described. These algorithms have been integrated in the Versatile Place and Route (VPR) CAD tool, a widely used CAD tool for FPGA architectural studies. We compare the architectures to each other, and to a "hard" programmable logic core. We also show how these cores can be made more efficient by creating a non-rectangular architecture, an option not available to "hard" core vendors. Finally, we evaluate various approaches to improve the area performance of our architectures by considering several architectural enhancements.
# TABLE OF CONTENTS

**ABSTRACT** .......................................................................................................................... ii

**TABLE OF CONTENTS** .......................................................................................................... iii

**LIST OF FIGURES** ................................................................................................................ vii

**LIST OF TABLES** ................................................................................................................... viii

**ACKNOWLEDGMENTS** ........................................................................................................... ix

1 **INTRODUCTION** ................................................................................................................. 1

1.1 *Motivation* ......................................................................................................................... 1

1.2 *Research Goals* .................................................................................................................. 4

2 **BACKGROUND AND PREVIOUS WORK** ........................................................................ 7

2.1 *Overview of FPGA Architectures* ................................................................................... 7

2.1.1 Logic Resources ............................................................................................................. 9

2.1.2 Routing fabric ............................................................................................................... 10

2.2 *CAD for FPGAs* ............................................................................................................... 12

2.2.1 Placement ..................................................................................................................... 14

2.2.2 Routing ......................................................................................................................... 16

2.3 *Embedded Programmable Logic Cores in SoC* ............................................................... 19

2.4 *Synthesizable Programmable Logic Cores* .................................................................... 20

2.5 *Focus and Contributions of this Thesis* ......................................................................... 21

3 **METHOD FOR CONSTRUCTING AN SoC CONTAINING A SYNTHESISIZABLE**
**PROGRAMMABLE LOGIC CORE** ....................................................................................... 23

3.1 *Design Process* ............................................................................................................... 23
3.2 Advantages and Disadvantages of the Soft Core Approach ........................................ 25

4 NOVEL FPGA ARCHITECTURES .................................................................................. 27

4.1 Directional Architecture ......................................................................................... 27

4.2 Gradual Architecture ............................................................................................ 31

4.3 Segmented Architecture ......................................................................................... 33

4.4 Summary .................................................................................................................. 35

5 PLACE AND ROUTE ALGORITHMS .................................................................... 36

5.1 Placement Algorithms ............................................................................................ 37

5.1.1 Directional Architecture ..................................................................................... 38

5.1.2 Gradual Architecture .......................................................................................... 39

5.1.3 Segmented Architecture ..................................................................................... 46

5.2 Routing Algorithm .................................................................................................. 53

5.3 Summary .................................................................................................................. 54

6 ARCHITECTURE PARAMETERS AND EVALUATION RESULTS .......................... 55

6.1 Experimental Framework ......................................................................................... 56

6.2 Optimal sizing for the Gradual Architecture ......................................................... 58

6.2.1 Gradual I/O Results ............................................................................................ 58

6.2.2 Gradual LUT size Results .................................................................................. 59

6.3 Gradual Architecture vs. Directional Architecture ................................................. 61

6.4 Gradual Architecture vs. Segmented Architecture ............................................... 62

6.5 Soft vs. Hard Programmable Logic Cores ............................................................... 64

6.6 Sensitivity of Results .............................................................................................. 65

6.7 Summary .................................................................................................................. 66
LIST OF FIGURES

Figure 2.1: Conceptual FPGA models [26] .................................................................................. 8
Figure 2.2: 4-input look up table [27] ................................................................................... 9
Figure 2.3: Island-style FPGA [32] ....................................................................................... 11
Figure 2.4: Different kinds of programmable switches in SRAM FPGAs [32] ......................... 12
Figure 2.5 FPGA CAD flow .................................................................................................. 13
Figure 2.6 Generic simulated annealing pseudo-code [32] ...................................................... 15
Figure 2.7: Pathfinder algorithm pseudo-code [42] ................................................................. 18
Figure 3.1: Method for constructing an IC containing a soft core .......................................... 24
Figure 4.1: Directional Architecture ...................................................................................... 28
Figure 4.2: Area breakdown of a 4x4 Directional Architecture core ....................................... 29
Figure 4.3: Inputs and outputs in the Directional Architecture ............................................. 30
Figure 4.4: Gradual Architecture .......................................................................................... 32
Figure 4.5: Inputs and outputs in the Gradual Architecture .................................................. 33
Figure 4.6: Segmented Architecture ....................................................................................... 34
Figure 5.1: Ease of routing and manhattan distance ............................................................... 40
Figure 5.2: Placement algorithm pseudo-code for the Gradual Architecture ......................... 42
Figure 5.3: Good placements on the Gradual Architecture .................................................... 43
Figure 5.4: Example placements on the Gradual Architecture .............................................. 44
Figure 5.5: Example placements on the Gradual Architecture: Sinks in many adjacent rows . 45
Figure 5.6 Placement algorithm pseudo-code for the Segmented Architecture ..................... 48
Figure 5.7: Good placements on the Segmented Architecture .............................................. 50
Figure 5.8: Example placements on the Segmented Architecture ......................................... 52
Figure 5.9: Poor placements resulting in congestion .............................................................. 53
Figure 6.1: Experimental methodology .................................................................................. 57
Figure 6.2: Gradual Architecture average area for various I/O ratio values ......................... 59
Figure 6.3: Gradual Architecture average area for various LUT sizes .................................... 60
LIST OF TABLES

Table 6.1: Directional and Gradual Architecture results .............................................. 62
Table 6.2: Gradual and Segmented Architecture results .............................................. 64
Table 6.3: Sensitivity results ......................................................................................... 66
Table 7.1: Reduced LUT input multiplexers Gradual Architecture results ..................... 76
Table 7.2: Further reduced Gradual Architecture results .............................................. 79
ACKNOWLEDGMENTS

First and foremost, I would to thank Dr. Steve Wilton for being a great supervisor. He was a constant source of encouragement and advice. His constant feedback, help and support is very much appreciated. Throughout my stay at UBC, I worked with a great research team: Kim, Kara, Dana, Julien, Martin, Tony, Andy and Steve. I would like to thank them for making my masters experience an enjoyable one. I am also very fortunate to be a member of the UBC System-on-Chip research group. I would like to extend my thanks to the SoC research group members for providing a friendly and pleasant environment.

This research project was funded by Altera Corporation, Micronet R&D, and the Natural Science and Engineering Research Council of Canada. Their financial support is greatly appreciated. I would also like to thank Dr. Vaughn Betz for providing the Versatile Place and Route CAD tool.

Last but not least, I dedicate this thesis to my parents, for being wonderful and supportive parents. They have been a constant source of motivation and support during the course of my studies. Without their support I would not have been able to accomplish what I have accomplished today. I would also like to thank my great brothers, for their continuous support. Finally, I am very grateful to my friends, who were always very supportive.
Chapter 1

1 INTRODUCTION

1.1 Motivation

Recent years have seen impressive improvements in the achievable density of integrated circuits. In order to maintain this rate of improvement, new design techniques are needed to handle the increased complexity inherent in these large chips. One such emerging technique is the System-on-a-Chip (SoC) design methodology. In this methodology, pre-designed and pre-verified blocks, often called cores or intellectual property (IP), are obtained from internal sources or third-parties, and combined onto a single chip. These cores can include embedded processors, memory blocks, and circuits that handle specific processing functions. The SoC designer, who would have only limited knowledge of the internal structure of these cores, can then combine them onto a chip to implement complex functions.

With this increasing complexity of integrated circuits, the task of designing and testing them is becoming unmanageable. As the transistor count reaches 30 million in typical SoC designs today, ensuring that the design is error-free has become virtually impossible. No matter how seamless the SoC design flow is made, and no matter how careful an SoC designer is, there will always be some chips that are designed, manufactured, and then deemed unsuitable. This may be due to design errors not detected by simulation or it may be due to a change in the design requirements. This problem is not unique to chips designed using the SoC methodology. Currently, the only way to solve such a problem is using an additional fabrication run. With the average cost of an all layer mask set revision approaching $750,000 in 0.13μm technology
and taking at least several months, this can lead to increased time-to-market, lost market-share, and increased development costs. Integrated circuit design and verification complexity is likely to become the limiting factor as more and more functionality is demanded in electronic products.

One solution is to circumvent the fabrication step. Field-Programmable Gate Arrays (FPGA’s) are off-the-shelf integrated circuits which can be configured to implement virtually any electronic circuit without the need for a fabrication plant. As changes in requirements arise or errors are discovered, the FPGA can be re-programmed virtually an unlimited number of times in seconds. This leads to improved design flexibility and reduced time-to-market. Unfortunately, electronic circuits implemented using FPGAs run slower and are less dense than their counterparts implemented using full custom ASIC solutions. In fact, a circuit implemented on an FPGA is typically at least 10 times less dense and three times slower than an equivalent implementation on an ASIC [2]. In many applications, particularly communications applications, FPGAs do not provide the required speed or density.

By combining the advantages of ASIC technology and FPGA technology onto a single chip, however, the best of both worlds can be delivered. This is possible by viewing a block of FPGA logic as a piece of intellectual property (IP). Then, the suppliers can integrate the programmable logic (FPGA) block with the custom circuitry and other IP blocks. In this way, companies can enjoy the benefits of flexibility and configurability by creating a custom integrated circuit (using traditional design techniques), and incorporating an FPGA programmable logic block, or core. Parts of the chip that are unlikely to change can be implemented using fixed circuitry, while the parts of the chip that may change can be implemented in the programmable logic core.
Several companies, including Actel, eASIC, Elixent, Atmel, Lattice, QuickLogic and Leopard Logic already provide programmable logic cores [3][4][5][6][7][8][9][10]. Yet, the use of these cores is still far from mainstream. There are a number of reasons for this:

1. Positioning the programmable logic core and connecting the core to the rest of the chip is not easy, using existing computer-aided design tools. Although the current tools have this capability, only the most experienced designers can use the tools to tackle these designs. This is somewhat of a chicken-and-egg problem: existing tools will not be modified to support the incorporation of programmable logic cores until this design technique becomes mainstream, and the design technique will not become mainstream until the tools are modified to support programmable logic cores. The new method, described in this dissertation, overcomes this hurdle.

2. Often, an integrated circuit would prefer to have many very small regions of programmable logic, rather than a single (or a handful of) large programmable logic regions. Often, circuits contain control logic that coordinates the operation of the rest of the chip. It would be beneficial to map selected parts of this control logic to programmable logic, rather than the entire control logic subcircuit.

3. Programmable logic cores come in fixed sizes. The integrated circuit designer must choose a programmable logic core that is closest to the desired size; this could lead to wastage of chip area.

4. The integrated circuit designer cannot modify the internal structure of the programmable logic core.
In this thesis, we describe an alternate method for incorporating programmable logic cores into an SoC. Rather than providing “hard” layouts, core vendors would provide “soft” descriptions of their programmable logic cores. These descriptions would typically be written in VHDL or Verilog. The integrated circuit designer could then incorporate the HDL description into their own HDL (for the fixed part of the chip) and synthesize the entire chip using existing synthesis techniques [11][12].

1.2 Research Goals

The goal of this research is to evaluate the feasibility of using synthesizable programmable logic cores and to show that including programmable logic cores in SoC is viable. In addition, we investigate suitable architectures and CAD algorithms. Although there has been considerable research on CAD algorithms and architectures for FPGAs, most of it has been targeted towards stand-alone FPGAs. It is very likely that different architectures and algorithms are more suitable for programmable logic cores that are synthesized and embedded rather than for “hard” programmable logic cores.

Specifically, the objectives of this research are as follows:

1. To propose the use of a synthesizable embedded programmable logic core in an SoC design environment.

2. To design novel embedded programmable logic core architectures that are synthesizable using commercial synthesis tools.

3. To develop the associated CAD algorithms required to map user circuits on these architectures.
4. To use the CAD tools to evaluate and optimize the various architectures.

The second objective listed above is to create FPGA architectures that can be used in embedded programmable logic cores. Traditionally, FPGAs are formed by hand-optimizing a single tile, and then cascading several tiles to form the FPGA. In the soft core methodology, however, the optimization and layout stages are performed by CAD tools, rather than by hand. Thus, it is important that the architectures can be easily incorporated into the ASIC design flow. In addition, the fact that our architecture is built with standard cells rather than a custom layout implies that the optimum architecture may be different than that for a hand-optimized FPGA. Thus, we present three new architectures which are synthesizable, and have been designed specifically for standard cell implementation.

In order to use our architectures, the CAD tools must be able to map circuits to them. The third objective of our research is to develop new CAD algorithms and integrate them into the Versatile Place and Route CAD suite [13], which is an existing FPGA place and route tool. It is expected that existing placement and routing algorithms will not suffice, since our architectures contain a unique routing structure.

The final goal of this research is to study the density of the different architectures, and to present enhancements to improve the density. We consider three architectures: Directional Architecture, Gradual Architecture and Segmented Architecture. Using the enhanced placer and router, we quantify area results of the three architectures and hence determine the penalty of using an embedded programmable logic core implemented using these architectures compared with a non programmable logic core implementation.
1.3 Thesis Organization

The next chapter provides an introduction to existing FPGA architectures, as well as an overview of the FPGA CAD flow. Relevant background information and previous work is also presented in this chapter. In Chapter 3 we describe, in detail, our new method for constructing an integrated circuit containing a programmable logic core. Chapter 4 describes novel synthesizable FPGA architectures. More specifically, it describes the Directional, Gradual and Segmented architectures. In Chapter 5, the CAD algorithms that were designed specifically for our novel architectures are explained. Experimental results are presented in Chapter 6. In particular, we measure and compare the area performance of the three architectures. Moreover, we determine the optimal values for various architectural parameters in the Gradual Architecture. We also quantify the area penalty incurred by using a synthesizable core rather than a hard core. In addition, a sensitivity analysis of our results is included. We then explore the possibility of making further enhancements to the Gradual Architecture in Chapter 7. Finally, the thesis concludes with a brief summary, and possible direction of future work, as well as the contributions of this research. Parts of this work have been published in [14]. The architectures and the techniques for creating synthesizable programmable logic cores have also been patented [15] and have been licensed by Altera Corporation.
Chapter 2

2 BACKGROUND AND PREVIOUS WORK

This chapter begins with a brief overview of FPGAs. More specifically, it introduces the architecture of the routing fabric and logic blocks used in existing FPGAs. The overview is followed by a description of the CAD flow used to map user circuits onto FPGAs. In particular, the placement and routing algorithms are discussed. It then summarizes various reasons that embedded programmable logic cores are attractive, and describes the previous work done in the area of synthesizable programmable logic cores. Finally, the focus and contributions of this research project are presented.

2.1 Overview of FPGA Architectures

Field-Programmable Gate Arrays are Integrated Circuits (ICs) that can be programmed after fabrication. Over the past years, the logic capacity of FPGAs has increased from thousands of gates to millions of gates [16][17]. As a result of this significant increase in capacity, FPGA applications are now more diverse and include entire system implementations. In fact, the FPGA market has dramatically expanded with annual sales totaling over $3 billion. Among the several vendors that dominate the FPGA market are companies such as Xilinx, Altera, and Actel [18][19][20].

Typically, FPGAs attain post-fabrication programmability through the use of three main components: (1) configurable logic blocks, (2) configurable routing fabric and (3) I/O blocks. Configuring the logic blocks and the routing fabric, according to a configuration bit stream, allows virtually any digital user circuit to be implemented on an FPGA. Recent additions to
these fundamental blocks in modern FPGAs include embedded memories and special feature blocks such as DSP blocks and processors [16][17][21]. Figure 2.1 shows conceptual representations of legacy as well as modern FPGAs.

Configurability is achieved within the FPGA by writing to configuration memories embedded in the FPGA. The values written to these memories dictate the functions of the logic blocks and the states of the routing interconnect switches, hence allowing user circuits to be implemented. Several technologies to achieve programmability exist, some of which are the antifuse technology [22] employed by Actel [20], floating gate transistors [23] and the SRAM cell technique [2][16][17][24][25]. By far, the most common approach used in industry to date, is to implement the configuration memory using SRAM cells. Thus, programmability is achieved by configuring pass transistors, multiplexers or tri-state buffers using CMOS SRAM cells. Vendors such as Xilinx and Altera [18][19] exploit such an approach.

Architectures illustrated in Figure 2.1 are referred to as island-style architectures, since the logic blocks resemble islands embedded within a sea of reconfigurable interconnect. These are the
most common architectures in industry today. Usually, logic blocks are arranged in a grid and are surrounded by vertical and horizontal routing tracks. In the following sections, the logic and routing resources are discussed in more detail.

2.1.1 Logic Resources
In order to provide the ability to implement any circuit, the FPGA’s logic resources must be flexible enough to implement many different logical functions. Most FPGAs use look-up tables (LUTs) as the basic element to implement logic. A $k$-input look-up table (k-LUT) has $k$ inputs and a single output. Such a $k$-input LUT contains $2^k$ configuration bits, and is capable of implementing any logical function of the $k$ inputs. Thus, a $k$-input LUT can implement $2^{2^k}$ distinct logic functions. Figure 2.2 shows an implementation of a 4-LUT.

Figure 2.2: 4-input look up table [27]
Most commercial FPGAs use a 4-LUT as their basic logic resource element [16][17][19][21][27]. However, for the work presented later in this thesis, we have demonstrated experimentally that a 3-LUT is more suitable. Hence, we use 3-LUTs in the remainder of this thesis.

2.1.2 Routing fabric

The routing fabric in an FPGA provides connectivity between the various internal components which include logic blocks and I/O blocks. An island-style architecture, such as that shown in Figure 2.3, is employed for routing between logic blocks. Three major components constitute the basic island-style architecture:

1. Connection blocks: Programmable connections to connect logic blocks to adjacent tracks.

2. Switch blocks: A set of programmable connections providing connectivity between routing channels.

3. Metal wires: Metal wires run vertically and horizontally throughout the FPGA to form routing channels.

As shown in Figure 2.3, connection blocks are adjacent to all four sides of the logic block. They provide a means of connecting routing tracks to logic block input pins, as well as connecting the logic block output pin to adjacent routing tracks.

At the intersection of each horizontal and vertical channel resides a switch block. Switch blocks allow wires in vertical and horizontal channels to be connected, in addition to allowing a horizontal-to-horizontal connection or a vertical-to-vertical connection. Providing fully
connectivity in the switch block would require a significant number of programmable switches and hence is not feasible. Several switch block topologies have been developed \cite{28}\cite{29}\cite{30}\cite{31}\cite{32}, in which some connections have been removed to reduce the number of programmable switches required. In our work, we use a modified version of the Disjoint switch block.

![Island-style FPGA](image)

Figure 2.3: Island-style FPGA \cite{32}

A variety of ways exist to implement the programmable connection switches employed by the connection blocks and switch blocks. Figure 2.4 shows three different ways to implement theses programmable switches. The majority of FPGAs today are SRAM-based \cite{16}\cite{17}
[21][27]; thus SRAM cells are used to control the pass transistors, multiplexers and tri-state buffers that make up the routing.

![Figure 2.4: Different kinds of programmable switches in SRAM FPGAs [32]](image)

Although each type of programmable switch is favored in certain situations, for the purposes of our work, we use multiplexers to provide connectivity in all the architectures presented later in this thesis.

### 2.2 CAD for FPGAs

Computer-Aided Design (CAD) tools are required to configure an FPGA. Since an FPGA contains millions of configuration memory bits, it is crucial to determine the values of these bits using CAD tools. The CAD tools convert a high-level description of the circuit to be implemented into a configuration bit stream; the bit stream defines the state of the FPGA routing switches, and the functionality of the lookup tables. This conversion is achieved through a series of stages, as illustrated in Figure 2.5.
The first step is for the user to provide an HDL description of the desired circuit to be implemented. During the high-level synthesis step, the high-level circuit description is converted into a netlist of basic logic gates. Followed by this step is the technology independent logic optimization step, in which all redundant logic is removed from the netlist. The next step is to map the logic gates to lookup tables using a technology-mapping algorithm.
After technology mapping, the lookup tables are packed into logic blocks. Each logic block and I/O block is then given a specific physical location in the FPGA using a placement algorithm. Once all the logic and I/O blocks have been placed, the connections between them are routed on the available routing tracks via a routing algorithm. At this point all logic block and I/O pad locations are known, as well as which routing tracks are used to connect them, therefore a configuration bit stream can be generated.

All steps prior to placement and routing are to a large extent independent of the FPGA architecture used. They are only influenced by whether or not lookup tables are used in the architecture. Since we use lookup tables in our architectures, we need not make any changes to these steps. However, the placement and routing steps are heavily dependent on the routing structure used in the FPGA architecture. Therefore, we need to change these algorithms to accommodate the unique routing structure used in our architectures. In Chapter 5, we detail the placement and routing algorithms used, and we show that indeed new algorithms are needed to make efficient use of our architectures.

2.2.1 Placement

Once the HDL description of the user circuit has been mapped into lookup tables and a netlist describing the connections between these blocks is available, a placement algorithm is used to determine the physical location of each logic block on the FPGA. Generally, placement algorithms strive to simultaneously minimize the wire length required to route nets, congestion, critical path delay, and power. Several placement algorithms have been developed. They can be classified as one of the following: min-cut [33][34][35], analytic [36][37] or simulated annealing placement algorithms [13][31][32][38][39][40][41][49]. Of these three, we focus on the latter approach since VPR (the place and route tool used in this research)
employs a simulated annealing placement algorithm [13]. An attractive characteristic of
simulated annealing algorithms is that they can very easily be used for new architectural
investigations by only modifying the underlying cost function to support new optimization
goals. In addition, they produce very high quality results.

Simulated annealing algorithms originate from the concept of the industrial annealing process,
where molten metal is gradually cooled to produce high quality metal objects [38]. Pseudo-
code for a generic simulated annealing placement algorithm is shown in Figure 2.6. A cost
function, which is designed based on the required optimization goals, is used to evaluate each
placement.

```plaintext
current_placement = RandomPlacement() /* Get initial random placement */
temperature = InitialTemperature() /* Get initial temperature */

while(ExitCriterion == False) {
    for (num_inner_loop_iterations) {
        /* Generate new placement by randomly swapping blocks */
        new_placement = TrySwap(current_placement)

        /* Calculate incremental cost of move */
        delta_cost = Cost(new_placement) - Cost(current_placement)

        r = random(0,1)
        if (r < \( e^{-\delta \text{cost} \cdot \text{temperature}} \)) {
            current_placement = new_placement /* Accept move */
        }
    }
    temperature = UpdateTemperature() /* Decrease temperature */
}
```

Figure 2.6 Generic simulated annealing pseudo-code [32]

Initially, the algorithm begins by generating a random placement. Several iterations are then
performed in which the algorithm attempts swapping random blocks. For each move, the
incremental cost is calculated by determining the change cost resulting from the new placement. A swap is automatically accepted if it results in a placement that improves the cost. If the swap increases the placement cost, however, there is a chance that the move will be accepted. The probability of accepting a move is $e^{-\delta_{\text{cost}}/\text{temperature}}$; where “$\delta_{\text{cost}}$” is the incremental cost of the move and “temperature” is the temperature parameter. At the beginning, the temperature is set a very high value so that virtually all moves are accepted. As the algorithm progresses, the temperature value gradually decreases allowing fewer and fewer moves that will degrade the cost of the placement to be accepted. Some seemingly bad moves are allowed in the beginning to allow the algorithm to escape local minima in the cost function. Eventually, only placements with better costs will be accepted.

The initial temperature and exit criterion to be used, as well as the number of times the inner loop should execute, and the amount by which the temperature should be decreased over the course of the anneal is termed the annealing schedule. The annealing schedule used in VPR is discussed in detail in [32].

2.2.2 Routing

Upon the completion of the placement phase, the physical location of each logic block is known. Using this information, connections can be made between the logic blocks according to a netlist which was generated at an earlier stage. The process of establishing paths for nets that connect logic blocks, is referred to as routing. Routing algorithms dictate which resources are used by which nets. Two optimization goals common across most routing algorithms are: (1) to minimize congestion so that a valid path to route each net is found [41][42], and (2) to minimize critical path delay.
Generally, routing algorithms can be classified as two-step routing algorithms [43][44][45][49] or combined global-detailed routing algorithms [41][42]. We only consider combined global-detailed routing algorithms, in which a complete path from source to sink is found, since this has been shown to work better in FPGAs [43].

In most cases, routing algorithms start by building a directed graph [41][42] called a routing resource graph to model the routing resources available in the FPGA. The nodes in this graph represent logic block pins or wires, and the edges represent possible connections. In addition, most routing algorithms are based on a sophisticated version of a maze router [46] that essentially runs Dijkstra's algorithm [47] to find the shortest path between a source and sink. However, such an approach used in isolation is likely to yield unroutable solutions due to the contention of routing resources. Rip-up and re-route techniques [48] are often used to overcome this problem. Such techniques rely on performing several routing iterations, and at the end of each iteration some nets are ripped-up and re-routed in a different order with hope to resolve congestion issues.

One such algorithm, which also focuses on congestion avoidance and delay minimization, is the Pathfinder algorithm [42]. The key concept of this algorithm is that it initially allows routing resources to be shared by different nets. As the algorithm proceeds, the penalty for sharing resources increases. Thus, in future iterations, this increase in cost of sharing resources gradually resolves congestion. The pseudo-code of the Pathfinder algorithm is shown in Figure 2.7.
Criticality(i,j) = 1 for all nets i and sinks j

while(OverusedResources() == True) {
    for each net i {
        ripup routing paths and update usage of affected resources
        for each sink j sorted in decreasing Criticality(i,j) order {
            PathQueue = all nodes, m, in path connected from source node sorted by cost (m)
            while(sink(i,j) is not found) {
                remove lowest cost node (low_cost_node) from PathQueue
                for(all nodes, n, connected to low_cost_node) {
                    PathQueue = node n sorted by cost(n)+cost(low_cost_node)
                }
            }
            update routing path of sink j and update resource usage
        }
    }
    update historical congestion of all nodes
    perform timing analysis and update Criticality(i,j) for all nets i and sinks j
}

Figure 2.7: Pathfinder algorithm pseudo-code [42]

In this algorithm, the cost of adding a node n to a connection path (i,j) is [32]:

\[
Cost(n) = Criticality(i, j) \cdot delay_{improvement}(n) + (1 - Criticality(i, j)) \cdot b(n) \cdot h(n) \cdot p(n)
\]

The first term emphasizes delay minimization by giving higher priority to delay as a connection becomes closer and closer to being on the critical path. The second term focuses on congestion minimization using a base cost \( b(n) \), a historical congestion term \( h(n) \), and a present congestion term \( p(n) \). Using this function, a decreasing criticality value shifts the focus of the algorithm from delay to congestion. This means that nets on or close to being on the critical path are routed for speed, while other nets try to avoid highly congested areas.
2.3 Embedded Programmable Logic Cores in SoC

The System-on-Chip design flow allows system designers to integrate various kinds of third-party Intellectual Property onto a single chip. In this flow, system designers obtain pre-verified IP blocks from various suppliers that specialize in the design of IPs used in different areas, and integrate them. As a result, the design and verification time, which directly translates to time-to-market, is significantly reduced.

There are many reasons as to why incorporating an embedded programmable logic core would be beneficial [50][51]:

1. Some design details can be left until late in the design cycle. In a communications application, for example, the development of a chip can proceed while standards are being finalized. Once the standards are set, they can be incorporated into the programmable portion of the chip. In other words, it allows designers to rapidly and inexpensively adapt to changes in standards or add specific features.

2. As products are upgraded, or as standards change, it may be possible to incorporate these changes using the programmable part of the chip, without fabricating an entirely new device.

2. In many cases, it may be possible to fabricate a single chip for an entire family of devices; the characteristics that differentiate each member of the family can be implemented using the programmable logic. This would, in effect, amortize the cost of developing the integrated circuit over several products. It also provides a mechanism for IP maintenance and upgrade.
3. Having a programmable logic core on chip can be very valuable from a testing perspective. Consider a faulty chip, it is possible to implement various test scenarios to test different parts of the chip and locate the error. As errors are discovered, the programmable logic core can be utilized to apply diverse test scenarios as necessary.

4. Moving risk-prone or critical areas of the design to the programmable logic core allows errors to be fixed without the costly refabrication process. Thus, the cost is decreased by avoiding increasingly costly mask set changes and silicon re-spins. This also saves the time overhead introduced by re-fabrication.

These advantages are compelling, and have already promoted several designers to employ programmable logic cores in their designs [52][53].

2.4 Synthesizable Programmable Logic Cores

Programmable logic cores such as those described in the previous section are, or have been, available from several vendors [3][4][5][6]. In each case, these vendors supply a “hard core” which contains the actual physical transistor layout information. In this thesis, we introduce the concept of a “soft” programmable logic core, in which the vendor provides a description of the behaviour of the core written in a hardware description language.

Synthesizable FPGAs have been described before. In particular, [11] describes the prototyping of an FPGA using standard cells. This is different than our work. In our work, the synthesized FPGA is the “end product”, while in [11], it is only a prototype of what will eventually be implemented as a hard core. As will be shown in the next chapter, synthesized cores have a number of advantages, including reduced design and test time, and increased flexibility, that makes them suitable as an “end product” rather than just a prototype. As far as we know, this
work (and the associated publication [14] and patent [15]) is the first work that proposes using these cores in this way.

2.5 Focus and Contributions of this Thesis

As described in the previous section, we propose using synthesizable programmable logic cores in an SoC environment. Chapter 3 will discuss this in more detail, and will provide a detailed flowchart showing the proposed flow.

Although it is possible to describe a standard FPGA architecture in a hardware description language and synthesize it, Chapter 4 will argue that the optimum architecture for a synthesizable core may differ greatly from that of a hard programmable logic core. Chapter 4 will then describe three novel architectures that have been designed specifically to be used as a synthesizable core.

Chapter 5 will present new placement algorithms that can be used to map user circuits to our new programmable logic architectures. Since the routing fabric in our architectures is significantly different than in a standard FPGA, existing placement programs will not be sufficient.

Chapter 6 presents experimental results that compare a soft programmable logic core to a hard core, and that compare the three architectures described in Chapter 4.

Finally, Chapter 7 returns to the architectures of Chapter 4, and shows how they can be further optimized to improve the density of the core.

Thus, the contributions of this thesis are:
1. A method of constructing an SoC using a synthesizable programmable logic architecture is introduced.

2. Three novel architectures for synthesizable cores are presented.

3. Placement algorithms for our new architectures are described.

4. Enhancements to improve the density of our three architectures are presented.

5. The architectures and enhancements are evaluated experimentally, and the density of the resulting implementations is compared to that if a hard core had been employed.
Chapter 3

3 METHOD FOR CONSTRUCTING AN SoC CONTAINING A SYNTHESIZABLE PROGRAMMABLE LOGIC CORE

As described in the introduction, system designers that wish to use a programmable logic core typically receive a “hard core” which contains the actual physical transistor layout information. The size and shape of the core is fixed; the only freedom the designer has is where to position the core on the chip. Using the alternate scheme described in [11],[12], however, the designer receives the core in the form of a “soft core”. A “soft core” is one in which the designer obtains a description of the behaviour of the core, written in a hardware description language. Note that this is distinct from the behaviour of the circuit to be implemented in the core, which is determined after fabrication. Here, we are referring to the behaviour of the programmable logic core itself.

3.1 Design Process

Since the designer receives only a description of the behaviour of the core, he or she must use synthesis tools to map the behaviour to gates. These synthesis tools can be the same ones that are used to synthesize the fixed (ASIC) portions of the chip. Figure 3.1 shows the new method we propose for constructing a chip that contains a “soft” core.
Partition the chip into fixed and programmable logic

Describe the fixed logic using a HDL

Obtain HDL description of a programmable logic core

Combine the two previous HDL descriptions to form the HDL description of the entire system

Synthesize chip level HDL description

Layout the chip

Fabricate the chip

Configure the programmable logic core to implement desired functionality

Figure 3.1: Method for constructing an IC containing a soft core
First, the integrated circuit designer partitions the design into functions that will be implemented using fixed logic and programmable logic, and describes the fixed functions using a hardware description language. Next, the designer obtains a description of the behaviour of a programmable logic core. This behaviour is also specified in a hardware description language. The designer then merges the behavioural description of the fixed part of the integrated circuit and the behavioural description of the programmable logic core, creating a behavioural description of the entire chip. Standard ASIC synthesis, as well as place and route tools are used to implement the behavioural description of the complete chip. In this way, both the programmable logic core and fixed logic are implemented simultaneously. The integrated circuit is then fabricated. Once the fabricated chip is received, the designer configures the programmable logic core to implement any desired functionality.

3.2 Advantages and Disadvantages of the Soft Core Approach

The primary advantage of the new method is that existing ASIC tools can be used to implement the chip. No modifications to the tools are required, and the flow follows a standard integrated circuit design flow that designers are familiar with. This will significantly reduce the design time of chips containing these cores. A second advantage is that this technique allows small blocks of programmable logic to be positioned very close to the fixed logic that connects to the programmable logic. The use of a "hard core" requires that all the programmable logic be grouped into a small number of relatively large blocks. A third advantage is that the new technique allows users to customize the programmable logic core to support his or her needs precisely. This is because the description of the behaviour of the programmable logic core is a text file which can be edited and understood by the user. Finally, as technology improves (as new fabrication processes are created, which happens roughly once
per year), no re-design of the integrated circuit is needed. It is possible that a "hard core" may not work well in the new fabrication process, and would need to be re-designed.

The primary disadvantage of the proposed technique is that the area, power, and speed overhead will be significantly increased, compared to implementing programmable logic using a hard core. Thus, for large amounts of circuitry, this technique would not be suitable. It only makes sense if the amount of programmable logic required is small. An envisaged application might be the next state logic in a state machine. In Chapter 6, we will quantify this tradeoff.
Chapter 4

4 NOVEL FPGA ARCHITECTURES

In this chapter, we describe three possible architectures for a synthesizable programmable logic core. We first introduce the Directional Architecture, in Section 4.1, which is based on the traditional island-style FPGA architecture. In Section 4.2 we present the Gradual Architecture, which was developed in an effort to improve the density of the core by removing some flexibility. Section 4.3 then describes the Segmented Architecture, which like the Gradual Architecture, contains fewer programmable switches than the Directional Architecture.

4.1 Directional Architecture

The most straightforward way to implement a synthesizable programmable logic core is to describe the behaviour of a standard FPGA at the RTL level using a hardware description language. Before doing so, we make the following observations:

1. It is important to note that synthesizable programmable logic cores are only feasible for small amounts of logic. An envisaged application would be the next state logic in a state machine.

2. Many of the synthesis tools, which will be used to synthesize the entire chip including our programmable logic core, will encounter timing difficulties if combinational loops exist.
These observations motivate us to modify a standard FPGA architecture. Consider the first observation. Since we are targeting small amounts of logic, we have decided that our architecture will only implement combinational logic, allowing us to remove all flip-flops. Flip-flops can be added at the inputs and outputs of the programmable logic core by the IC designer if desired. Removing flip-flops reduces area and simplifies timing analysis.

The second observation is a problem, since an un-programmed FPGA contains many combinational loops (a good designer will rarely configure the FPGA to contain combinational loops, but before configuration, these loops exist). Recall that one of the primary requirements of our programmable logic core is that it is synthesizable by standard tools. Thus, we have created a "directional" architecture, in which the flow between logic blocks can only flow from left to right. Since our architecture is only to implement combinational circuits, this will not cause a problem; any feedbacks that are required can be implemented outside of the core. Based on these observations, we have created the architecture shown in

![Directional Architecture](image)

![Close-up of Switch Block](image)

Figure 4.1: Directional Architecture
Figure 4.1(a). Each switch block is a standard switch block, with the right-to-left connections removed, as shown in Figure 4.1(b).

Two issues were encountered when developing this architecture. The first issue concerns the look-up table size that should be used. Four-input look-up tables are most commonly used in industry. However, this is not necessarily the best choice for our architecture. Since our core will be synthesized and our architecture is unique, the optimal look up table size could be different. Figure 4.2 shows the area breakdown of a 4x4 core. As can be seen from the figure, the logic resources constitute 59.8% of the total core area, whereas the routing resources contribute to 23.6% of the total area. On the other hand, in a hand optimized FPGA the bulk of the area is due to the routing resources; thus, using larger lookup tables does not incur significant area overhead. This indicates that the ratio of logic area to routing area is larger in a synthesized core than in a hand optimized core. Intuitively, this leads to the conclusion that a smaller LUT size would be more area efficient. As a result, we use three input LUTs in this architecture.

![Figure 4.2: Area breakdown of a 4x4 Directional Architecture core](image-url)
Next, we consider the issue of how many primary inputs and outputs the core should have. Similar to any FPGA architecture, there are I/O blocks surrounding the perimeter of the programmable logic core. Each I/O block can contain more than one primary input or output. It is necessary to address the issue of how many inputs or outputs per I/O block would be adequate for our architecture. The number of primary inputs or outputs per I/O block is referred to as the “I/O ratio”. In our case, using an I/O ratio of two was sufficient, meaning that each of the I/O blocks can contain up to two primary inputs or outputs. This is illustrated in Figure 4.3.

![Figure 4.3: Inputs and outputs in the Directional Architecture](image-url)
4.2 Gradual Architecture

We can create a more efficient architecture by making a few additional observations.

1. First we consider the flexibility offered by the architecture. Today, FPGAs are targeted to implement large system-sized circuits. Therefore, current architectures provide sufficient flexibility to accommodate the requirements of implementing such circuits. The directional architecture described in the previous section is based on a standard FPGA architecture; therefore the flexibility it provides could be overkill for implementing small circuits. It is conceivable that, since we are only implementing such small circuits, we can remove some flexibility.

2. Next, we investigate the issue of connecting the programmable logic core to the rest of the fixed logic in the chip. In particular, since the primary inputs and outputs will be hardwired to specific input and output pads, extra flexibility will be required on the inputs and outputs.

3. Finally, we note that unlike a hard FPGA layout, it is not critical that each tile be identical. In a hard layout, FPGA vendors do not wish to layout multiple tiles; in our case, the tiles are synthesized and laid out automatically by CAD tools.

These observations lead to the architecture in Figure 4.4, which we call the “Gradual Architecture”. Like the Directional Architecture, signals in the Gradual Architecture flow from left to right, and the logic resources consist only of 3-LUTs. The horizontal routing channels gradually increase in width from left to right. The vertical tracks are only accessible through LUT outputs (each vertical track can be driven by one LUT), and can be connected to horizontal tracks using a dedicated multiplexer at each grid point. Note that, except for this multiplexer, no switch blocks are required. Inputs to a LUT are selected from the horizontal
tracks above and below that specific LUT, and from the vertical tracks in the previous column.

The extension of this architecture to any number of rows and columns is straightforward.

![Gradual Architecture diagram](image)

Figure 4.4: Gradual Architecture

The routing multiplexers in the first column are different from the others. We have performed experiments and shown that the primary inputs are frequently required in many different columns. Thus, we have included routing multiplexers in each row (we will vary the number of these multiplexers in Chapter 7). Similarly, the inputs to the LUTs in the first column are selected from all the primary inputs.

The I/O blocks in this architecture are on the right and left hand side of the core. Primary inputs are placed to the left of the core and primary outputs are placed to the right of the core, as shown in Figure 4.5. Within each row, there are four output select multiplexers; each output
select multiplexer selects a signal to become a primary output of the core. Experimentally, we found that using an I/O ratio of four yields the best results. Hence, there are four inputs and four outputs per block in this architecture. The output multiplexers choose between the outputs of all LUTs located in the last column and any horizontal line located above or below that specific row. The exception to this is that only one routing multiplexer per row from the first column passes a signal to the output select multiplexers.

![Figure 4.5: Inputs and outputs in the Gradual Architecture](image)

4.3 **Segmented Architecture**

An alternate architecture motivated by the above observations is shown in Figure 4.6. In this architecture we also limit the propagation of signals to one direction, from left to right. The primary difference between this architecture and the Gradual Architecture is that, rather than extending each horizontal track to each logic block to the right of the track driver, the horizontal tracks span only one logic block. Each of the vertical tracks is driven by a LUT output. Connections between adjacent horizontal tracks or adjacent LUTs can be made through a routing multiplexer. At each grid point, there is a dedicated routing multiplexer used
to select between a specific LUT output and a specific horizontal track. Each of the routing multiplexers in the first column is used to select between a set of four primary inputs. Two of the LUT inputs are selected from the horizontal tracks below the LUT, while the third input is selected from the horizontal tracks above the LUT.

![Segmented Architecture Diagram](image)

Figure 4.6: Segmented Architecture

The I/O block structure in this architecture is similar to that of the Gradual Architecture. All the primary input blocks are to the left of the core, and primary output blocks are to the right. Again, an I/O ratio of four was used in this architecture. Each of the primary outputs in a certain row is selected using an output select multiplexer, which chooses between the horizontal tracks above and below that row, and the outputs of the LUTs in the last column.
4.4 Summary

In this chapter, we presented three synthesizable programmable logic cores. We first introduced the Directional Architecture, which is similar to a standard island-style FPGA architecture, except for the fact that signal flow is uni-directional (from left to right). However, keeping in mind that standard FPGA architectures might provide more flexibility than necessary, we developed the Gradual Architecture. In the Gradual Architecture we reduce the flexibility within the core by taking advantage of the fact that we can have more than one type of tile in a synthesizable architecture. Finally, we propose the Segmented Architecture as another possible architecture to implement synthesizable programmable logic cores. Similar to the Gradual Architecture, the flexibility of the segmented architecture is also less than that of a standard FPGA. Compared to the Gradual Architecture, the Segmented Architecture will require shorter wires, and each tile may be smaller. Intuitively, however, it will be less routable than the Gradual Architecture. In Chapter 6, we compare these architectures experimentally.
Chapter 5

5 PLACE AND ROUTE ALGORITHMS

Before we can assess the density of the presented architectures, it is essential to determine the size of the programmable logic fabric required to implement a given user circuit, in addition to the actual physical area of such a core. A fair estimate of the overall area efficiency of the architectures can only be achieved using the combination of these two pieces of information. For instance, for a given core size, one architecture could have a smaller physical area than an alternate architecture. However the routing structure employed by this architecture might be very restrictive, resulting in a much bigger core being required to implement a given user circuit and in a significant area overhead.

Three things were needed in order to compare the logic fabric size needed by each of the architectures:

1. Several benchmark circuits.

2. A place and route CAD tool that serves as a vehicle for mapping a user circuit onto the architecture.

3. A placement algorithm, for each architecture, to be employed by the selected CAD tool. Since our architectures contain novel routing structures, it is likely that existing placement and routing algorithms will not suffice.

A set of benchmark circuits from the MCNC benchmark circuit suite was used. Since we do not intend to use our architectures in large cores and our architectures do not provide support
for sequential logic, we are primarily interested in small combinational circuits. Therefore, we selected circuits that were purely combinational, and contained between 3 and 173 logic blocks.

The CAD tool used is Versatile Place and Route (VPR) [13]. VPR allows a researcher to model FPGA architectures of various capabilities and sizes, and place and route user circuits in order to test each architecture. VPR takes a netlist which represents the user circuit, and a text file which describes the FPGA architecture. From these VPR can place and route the user circuit onto the architecture.

In this chapter, we present novel CAD algorithms that were developed in order to allow user circuits to be placed on the architectures described in Chapter 4. Section 5.1 contains three subsections describing the placement algorithms used for each of the architectures. In the first subsection we present the placement algorithm for the Directional Architecture. The next subsection describes the placement algorithm for the Gradual architecture. The placement algorithm for the Segmented Architecture is explained in the last subsection. Finally, Section 5.2 describes the routing algorithm used by all three architectures.

5.1 Placement Algorithms

Placement follows technology mapping and packing in the FPGA CAD flow. In this stage, all the logic blocks generated at the end of the previous stage are assigned physical locations in FPGA according to a specific algorithm. In the following three subsections we describe the placement algorithms used for each of the architectures previously presented in Chapter 4; namely, the Directional, Gradual and Segmented architectures.
5.1.1 Directional Architecture

The placement algorithm for the Directional Architecture described in Section 4.1 is based on the original simulated annealing placement algorithm from VPR [13]. The only change was to put a restriction on the placer, which stipulates that sources for all blocks must originate from the left of that block. In other words, the output of a block could only flow into blocks to its right. During the annealing and the initial random placement, we never allow a move which would result in an illegal placement. The cost function used in the VPR placement algorithm consists of a wiring cost and a timing cost, defined as follows:

\[
\text{Wiring Cost} = \sum_{n=1}^{N_{n=1}} q(n) \left[ \frac{b_{x}(n)}{C_{x,v}(n)} + \frac{b_{y}(n)}{C_{y,v}(n)} \right]
\]

\[
\text{Timing Cost} = \sum \text{Delay} \cdot \text{Criticality}^{\text{Criticality Exponent}}
\]

The algorithm uses these two cost components to minimize routing and critical path delay. As described in [13], \(b_{x}\) and \(b_{y}\) are the horizontal and vertical span of the bounding box of net \(n\). The terms \(C_{x,v}\) and \(C_{y,v}\) are the average channel capacities of the horizontal and vertical tracks, respectively [13]. A compensation factor, \(q(n)\), is used to compensate for the underestimated lengths of wires with more than three terminals [13].

Minimizing the critical path delay is achieved by considering the timing cost of a connection. In the above equations, \(\text{Delay}\) is the delay of the connection between the source and sink, \(\text{Criticality}\) is a measure of how close the connection is to being on the critical path, and the \(\text{Criticality Exponent}\) is a constant [40]. The total cost is the sum of the wiring cost and timing cost of all nets.
The cost function used in the VPR placement algorithm depends on the delay of potential connections as well as on the Manhattan distance between pins. In a synthesized core, the delay between pins depends on where the individual cells that make up the core are positioned; it may be that blocks adjacent in the conceptual representation may be positioned far apart in the actual silicon. Nonetheless, we base our placement cost function on the distances and delays in the conceptual representation.

5.1.2 Gradual Architecture

It was necessary to develop a new placement algorithm for the gradual architecture, for two key reasons:

1. VPR tries to place sources and sinks as close to each other as possible, assuming that an optimal placement is achieved by minimizing the Manhattan distance between sources and sinks. This is true for typical architectures, since such a placement would require less routing resources and hence be more routable. However, this is not necessarily true for the gradual architecture. In the gradual architecture, the ease of routing a circuit is not governed by the Manhattan distance between the sources and sinks. It could be that more routing resources are required to route sources and sinks that are closer together. Figure 5.1 illustrates such a scenario. In the first case, the source and sink are further apart and yet no shared routing resources are used. On the other hand, in the second case, despite the fact that the source and sink are close to each other, the use of a shared routing multiplexer is required.
2. Due to the limited routing resources available in this architecture, it is very difficult to obtain a routable placement if the availability of the routing resources is not taken into consideration during the placement phase. Poor placements can easily lead to unroutable implementations. Therefore, it is important to enhance the placement algorithm to make it aware of the routing path that will be taken.

Motivated by the above, we developed a placement algorithm for the gradual architecture which aims at minimizing the overuse of the routing resources. The algorithm is based on four concepts:

1. Since a logic block has a dedicated multiplexer to select each of its inputs, there is no cost associated with routing a signal to any block in the adjacent column (i.e. it is free). This is because the signal is available through dedicated multiplexers at each of the LUT inputs, and hence the use of routing multiplexers is not needed.

2. If a routing multiplexer is used to route a single net, it is assigned a minimal cost.
3. As the demand for a certain routing multiplexer grows, its cost is increased. This will force a net to use that routing multiplexer only if there is no other choice.

4. Once a signal is routed to a sink in a specific row, all other sinks which lie in the adjacent rows can get the signal at no extra cost.

The pseudo-code for the algorithm is shown in Figure 5.2.
/* Initialization*/
total_cost=0
for each mux m {
    mux_occupancy=0
    mux_capacity="Number of output signals that can be passed with this multiplexer"
}
for each row r {
    histogram[r]=0
}

/* Set mux occupancies */
for each net n {
    source_x="x-coordinate of the source signal"
    for each sink s {
        sink_y="y-coordinate of the sink s"
        histogram[sink_y]=1
    }
    for each row r {
        if histogram[r] = 1 then
            if histogram[r+1] = 1 then { /* signal is shred between two rows */
                mux_occupancy(sink_x)(r)+=1
                r++
            } else {
                mux_occupancy(sink_x)(r)+=0.5
                mux_occupany(sink_x)(r+1)+=0.5
                r++
            }
    }
}

/* Calculate cost for each mux */
for each mux m {
    if mux_occupancy < mux_capacity {
        single_mux_cost(m)=0
    } else {
        single_mux_cost(m)=[mux_occupancy(m) + 0.2 - mux_capacity(m)]
    }
}

/* Calculate total cost */
for each mux m {
    total_cost=total_cost + single_mux_cost(m)
}

Figure 5.2: Placement algorithm pseudo-code for the Gradual Architecture

We use a simulated annealing based algorithm, with a unique cost function, as described below. To illustrate the algorithm, consider the examples shown in
Figure 5.3. This figure shows two examples of a "good" placement on the Gradual architecture. In Figure 5.3(a), a logic block drives logic blocks in an immediately adjacent column. This net can be routed "for free" since no shared resources are required. Note that the multiplexer used to feed each input pin of a logic block is not a shared resource; there is one multiplexer per input pin. Any number of sinks in the column immediately adjacent to the source can be connected in this way.

![Figure 5.3: Good placements on the Gradual Architecture](image)

For nets which drive logic blocks that are not in the immediately adjacent column, the routing multiplexers must be used. Since these multiplexers are shared resources, we wish to minimize the number of multiplexers used by each net. In the example of Figure 5.3(b), a net drives four sinks, but only needs one routing multiplexer, since the sinks are all in two vertically adjacent rows (meaning the track between the two rows can be used to drive all sinks). Again note that the multiplexers used to feed the input pins of each logic block are not shared resources, and thus do not play into the cost of a given placement.

The cost function used in our placement algorithm directly relates to the overuse of routing multiplexers. The cost of a given placement on an C-column, R-row core is:
\[
    \text{Cost} = \sum_{r=0}^{R} \sum_{c=0}^{C} \left[ \text{MAX}(0, \text{Occ}(c,r) - \text{Cap}(c,r) + \gamma) \right]
\]

where \( \text{Occ}(c,r) \) is the occupancy of the routing multiplexer (defined below) at location \((c,r)\), \( \text{Cap}(c,r) \) is the capacity of the multiplexer (defined below) at location \((c,r)\) and \(\gamma\) is a small constant (experimentally, we have found 0.2 works well). The capacity of all routing multiplexers is 1, except for those in the first column, where the capacity is equal to the number of horizontal lines that can be driven by primary inputs (in Figure 4.4, this would be 3).

The occupancy of a routing multiplexer is an estimate of how many nets would like to use that routing multiplexer. We can write this as the sum of the estimated demand for that multiplexer by each net:

\[
    \text{Occ}(c,r) = \sum_{n \in \text{Nets}} \text{demand}(c,r,n)
\]

where \( \text{demand}(c,r,n) \) is the estimated demand for the routing multiplexer at \((c,r)\) by net \(n\). This number is between 0 and 1; 0 means there is no chance that net \(n\) will want to use this multiplexer, while 1 means that net \(n\) will definitely want to use this multiplexer.

![Figure 5.5: Example placements on the Gradual Architecture](image)
Consider the net in Figure 5.5(a). In this case, it is equally likely that the net will use the two indicated multiplexers; therefore, the demand term for this net for each of the two multiplexers is 0.5. In Figure 5.5(b), it is likely that the net will use the indicated multiplexer, since a single multiplexer can be used to feed all three sinks, so the demand term for that net is 1. Note that a valid routing could be found that does not use this multiplexer, however, such a route would require two routing multiplexers. During placement, we assume that this will not happen, and thus, set the demand term for all other routing multiplexers for this net to 0. Note that this does not mean the router is constrained to use this routing multiplexer (see Section 5.2).

Finally, Figure 5.6 shows a net that drives four vertically adjacent rows. In this case, we assume, during placement, that the two indicated routing multiplexers are used with probability 1. Experimentally, we have determined that this leads to better results than if we assign all five routing multiplexers the same value (which would be lower than 1). Again, note that the router is not constrained to actually use the indicated multiplexers.

Figure 5.6: Example placements on the Gradual Architecture: Sinks in many adjacent rows
5.1.3 Segmented Architecture

In this subsection, we describe a placement algorithm for the Segmented Architecture. In the segmented architecture, the routing resources are even less flexible than in the gradual architecture. As a result, the placement algorithm plays a more vital role in determining the routability of a circuit. Similar to the placement algorithm for the gradual architecture, the placement algorithm for the segmented architecture also strives to minimize the overuse of routing resources. It achieves this using the following concepts:

1. As in the previous algorithm, a small cost is assigned to a routing multiplexer if only a single net wishes to use it.

2. In this architecture, each LUT input pin has a dedicated multiplexer to select that input. However, the input pins are not all equivalent. There are two inputs that come from the tracks below a given LUT, and only one input that comes from the tracks above the LUT. In order to compensate for this, a higher cost is assigned to a routing multiplexer which will route a signal to the tracks above a LUT.

3. Once a signal is routed to a sink in a specific row, it can be shared by sinks in adjacent rows at no extra cost.

4. When a routing multiplexer in a certain row is allocated to route a signal, all the proceeding multiplexers along the same row are allocated to route that signal until the sink is reached.

The pseudo code for the algorithm is shown in Figure 5.7.
/* Find region boundaries */
for each swap of two blocks {
    max_x="Maximum x-coordinate"
    min_x="Minimum x-coordinate"
    max_y="Maximum y-coordinate"
    min_y="Minimum y-coordinate"
}

/* Find nets affected by the swap */
for each of the affected nets {
    for each pin of this net {
        x="x coordinate of the block to which this pin connects"
        y="y coordinate of the block to which this pin connects"
        max_x="Max between x and current value of max_x"
        min_x="Min between x and current value of min_x"
        max_y="Max between y and current value of max_y"
        min_y="Min between y and current value of min_y"
    }
}

/* Initialization*/
total_cost=0
for each mux m in region bounded by min_x,max_x,min_y,max_y {
    new_mux_occupancy=0
    old_mux_occupancy=0
    /* keep copy of occupancy in case move is rejected */
    temp_mux_occupancy=mux_occupancy
    mux_capacity=1 "Capacity is always one in this architecture"
}
for each of the affected nets {
    /* Calculate occupancies after move (new_mux_occupancies) */
    for each row r between min_y and max_y {
        histogram[r]=0
        travel_length[r]=0
    }
    source_x="x-coordinate of the source signal"
    source_y="y-coordinate of the source signal"
    for each sink s {
        sink_x="x-coordinate of the sink of the signal"
        sink_y="y-coordinate of the sink of the signal"
        histogram[sink_y]=1
        travel_length[sink_y]="Max horizontal distance from source to sink"
    }
    for each row r between min_y and max_y {
        if histogram[r]=1 then
            if histogram[r+1]=1 then /* signal is shared between two rows*/
                max_travel_length=Max between travel_length[r+1], travel_length[r]
                for i between sourcex and sourcex+max_travel_length
                    new_mux_occupancy(i)(r)(sourcey) += 1
                r++
            else
                for i between sourcex and sourcex+travel_length[r]
                    new_mux_occupancy(i)(r)(sourcey) += 0.34
                    new_mux_occupancy(i)(r-1)(sourcey) += 0.66
                r++
        
    }
for each of the affected nets {
    /* Calculate occupancies before move (old_mux_occupancies) */
    for each row r between min_y and max_y {
        histogram[r]=0
        travel_length[r]=0
    }
    source_x="x-coordinate of the source signal"
    source_y="y-coordinate of the source signal"
    for each sink s {
        sink_x="x-coordinate of the sink of the signal"
        sink_y="y-coordinate of the sink of the signal"
        histogram[sink_y]=1
        travel_length[sink_y]="Max horizontal distance from source to sink"
    }
    for each row r between min_y and max_y {
        if histogram[r] is 1 then
            if histogram[r+1] is 1 then /* signal is shared between two rows*/
                max_travel_length=Max between travel_length[r+1], travel_length[r]
                for i between source_x and source_x+max_travel_length
                    old_mux_occupancy(i)(r)(source_y) += 1
            else
                for i between source_x and source_x+travel_length[r]
                    old_mux_occupancy(i)(r)(source_y) += 0.34
                    old_mux_occupancy(i)(r-1)(source_y) += 0.66
        
    }
    /* Calculate change in mux occupancy*/
    for each mux m in region bounded by x_min, x_max, y_min, y_max
    mux_occupancy(m) += new_mux_occupancy(m) - old_mux_occupancy(m)
    /* Calculate cost for each mux */
    for each mux m
        if mux_occupancy < mux_capacity
            single_mux_cost(m)=0
        else
            single_mux_cost(m)=[mux_occupancy(m) + 0.2 - mux_capacity(m)]
    /* Calculate total cost */
    for each mux m
        total_cost=total_cost + single_mux_cost(m)
}

Figure 5.7 Placement algorithm pseudo-code for the Segmented Architecture
A fundamental difference between this placement algorithm and the Gradual Architecture placement algorithm described earlier is that the cost of a move is calculated incrementally. We have found that recalculating the cost for all the multiplexers throughout the core after every move substantially affects the runtime of the algorithm, making it infeasible to recalculate the cost in the same way it was done for the Gradual Architecture. Two key issues contribute to the increased runtime. First, since the routing multiplexers in this architecture are less flexible than in the Gradual Architecture, the routing structure is very restrictive. Therefore, a larger number of iterations are required to obtain a valid placement. Second, the Segmented Architecture contains more routing multiplexers than the Gradual Architecture. Hence, in this algorithm, the cost is calculated for all the routing multiplexers only after the initial placement. From that point on, the incremental cost is calculated for the routing multiplexers affected by a move. This results in the same total cost as if it was fully recalculated.

Again, we employ a simulated annealing algorithm with a modified cost function to suit the characteristics of our architecture. To illustrate the algorithm, consider the placement shown in Figure 5.8. In this placement, the source drives four sinks in different rows and columns. We can take advantage of the fact that the sinks lie in adjacent rows and use the horizontal tracks common to both rows to route the net to all the sinks. Assuming that such a placement will be used, each of the sinks can select the signal using one of the dedicated multiplexers at each of its input pins.
Figure 5.8: Good placements on the Segmented Architecture

The cost function used in the placement algorithm for the Segmented Architecture is similar to that for the Gradual Architecture. The primary difference is that, at each row and column intersection, there are several routing multiplexers instead of just one. These routing multiplexers are not all equivalent; each logic block can only drive one of the routing multiplexers. Thus, we cannot simply use the previous algorithm with the capacity value set to a larger number. Instead, we maintain an occupancy and a capacity value for each of the routing multiplexers. In addition, each input to a logic block is taken from one of the two adjacent horizontal rows (for example, in Figure 4.6, two of the logic block pins must be taken from the horizontal row below the logic block, while the third must be taken from the horizontal row above the logic block). In the Gradual Architecture, each logic block input can be driven by either horizontal row. A final difference is that in the Segmented Architecture, the horizontal span of a connection matters (since a longer span means more segments are used). In the Gradual Architecture, the horizontal span does not affect routability at all, since each track spans across all logic blocks to the right of the driver.
The cost function used in this placement algorithm directly relates to the overuse of routing multiplexers. The cost of a given placement on an C-column, R-row core is:

\[
\text{Cost} = \sum_{r=0}^{R} \sum_{c=0}^{C} \sum_{t=0}^{R} \left[ \text{MAX}(0, \text{Occ}(c,r,t) - \text{Cap}(c,r,t) + p) \right]
\]

In this case, we maintain a separate occupancy value for each multiplexer (there are R multiplexers per row/column intersection). The value of \(\text{Cap}(c,r,t)\) is 1 for all multiplexers.

As in the Gradual Architecture placement algorithm, the occupancy of a routing multiplexer is an estimate of how many nets would like to use that routing multiplexer. It is expressed as the sum of the estimated demand for that multiplexer by each net:

\[
\text{Occ}(c,r,t) = \sum_{n \in \text{Nets}} \text{demand}(c,r,n)
\]

where \(\text{demand}(c,r,n)\) is the estimated demand for a routing multiplexer at \((c,r)\) by net \(n\). This number is between 0 and 1; 0 means there is no chance that net \(n\) will want to use this multiplexer, while 1 means that net \(n\) will definitely want to use this multiplexer. Consider the placements shown in Figure 5.9. In the first case, the sink lies in the column adjacent to that of the source. This is the only sink; therefore the placer has the option of using any one of the selected routing multiplexers. However, due to the non-equivalence of the LUT input pins, it is more desirable to use the routing multiplexers which lie below the sink since two of the inputs are selected from the tracks below the sink.
Selecting an appropriate value for the demand term is very important. Figure 5.10 shows how easily congestion can occur if the placement algorithm makes a poor decision as to which routing multiplexers to use (by not assigning appropriate occupancies to the routing multiplexers). The figure shows two sources driving a sink in the adjacent column. Figure 5.10(a) illustrates how a placement can lead to an unroutable circuit if a poor occupancy is assigned to the routing multiplexers during the placement phase. In this case, there is a single sink and two sources, and if the placer considers the routing multiplexers above and below the sink as equally suitable choices, a placement such as that shown in Figure 5.10(a) could result. Since the sink can only select one of its inputs from the tracks above it, the router will not be able to perform the required connections. However the sink can select two of its inputs from the tracks below it. Thus, the problem can be overcome by using the routing multiplexers below the sink as shown in Figure 5.10(b). Thus, we wish to make it more desirable to use the routing multiplexers below a sink. We assume that it is twice as probable to use a routing multiplexer below a given sink. Clearly, such a problem can also be avoided by maintaining an occupancy value for each of the multiplexers selecting the LUT inputs. However, we have determined experimentally that this is not necessary.
5.2 Routing Algorithm

Routing is the process by which all connections between logic block inputs and outputs, required by the user circuit, are determined. We used the routing algorithm from VPR [13] for all three architectures. Essentially, it is a negotiated congestion algorithm based on the Pathfinder algorithm [42]. Initially, the algorithm routes a net using the shortest possible path without considering any overuse of routing resources (in other words, it assumes that it is acceptable for more than one net to share the same routing resource). Then, for a maximum number of iterations (in our case the router performs thirty iterations), the cost of overused resources is gradually increased. The cost function used in this algorithm is:

$$\text{Cost}(n) = \text{Criticality} \cdot \text{delay}_{\text{Elmore}}(n) + (1 - \text{Criticality}) \cdot b(n) \cdot h(n) \cdot p(n)$$

In this cost function, the first term is used to account for the delay; where Criticality is a measure of how close the currently routed net is to the critical path, and \( \text{delay}_{\text{Elmore}}(n) \) is the Elmore delay of a net. The later term uses three components \( b(n), h(n) \) and \( p(n) \) to determine the congestion cost. Where \( b(n), h(n) \) and \( p(n) \) refer to the base cost, historical congestion and present congestion of a net. As routing proceeds, the value of \( p(n) \) is
increased to penalize sharing resources. Once a legal solution is obtained, in which each routing resource is used by a single net, the algorithm terminates. A more complete and detailed explanation of the algorithm can be found in [42]. Although our architectures might not have required the use of such a complex routing algorithm, we found experimentally that it yields good results.

5.3 Summary

In this chapter, we presented a placement for each of the Directional, Gradual, and Segmented architectures. The placement algorithm for the Directional Architecture was based on the original placement algorithm from VPR [13], with the additional restriction of only placing sinks to the right of sources. New placement algorithms were developed for the Gradual and Segmented architectures, however. These algorithms employ unique cost functions which were designed to specifically optimize for our architectures. We used the original VPR routing algorithm [13] for all of our architectures. A brief description of the routing algorithm was presented.
Chapter 6

6 ARCHITECTURE PARAMETERS AND EVALUATION RESULTS

New architectures and CAD algorithms can only provide insight when they are accompanied by experimental evaluations and comparisons. In this chapter, we use the CAD algorithms described in the previous chapter to evaluate the architectures presented in Chapter 4. More specifically, we focus on the following:

1. First, we investigate the effect of varying several architectural parameters in the Gradual Architecture. In particular, we focus on determining the optimal values for the I/O ratio and the LUT size. These results are presented in Section 6.2.

2. Next, we compare the density of the Gradual and Directional Architectures. Following that, we compare the density of the Gradual and Segmented Architectures. Our evaluation metric is the average area required to implement a class of benchmark circuits on each of the architectures. A discussion of the obtained results is presented in Sections 6.3 and 6.4 respectively.

3. After we compare our architectures to each other, we compare the area of a Gradual Architecture core to a hard core of an equivalent size. Section 6.5 contains the details and result of this comparison.

4. Finally, several architectural and CAD tool parameters are varied and the sensitivity of our results to these variations is measured and presented in Section 6.6. We consider
the effect of changing the I/O ratio value, and running VPR with a "fast" option enabled during the placement and routing of the benchmark circuits.

Before we present our experimental findings, a brief description of the experimental methodology is provided in Section 6.1. Finally, the chapter concludes with a summary of all the results in Section 6.7.

6.1 Experimental Framework

In evaluating our architectures, we used an experimental approach. Our methodology involved mapping benchmark circuits onto the architectures using CAD tools. An essential requirement was that the benchmark circuits be representative of real-world designs. We used 19 small combinational circuits from the Microelectronics Center of North Carolina (MCNC) [56]. We chose small circuits since these are the type of circuits we expect to be used with our architecture; large circuits would likely be implemented by a hard programmable logic core.

Figure 6.1 depicts our experimental methodology. Note that prior to the placement stage, the CAD flow described in Section 2.2 was used to generate the netlist files. Further, details of modifications made to the VPR routing resource graph, I/O connections and graphics are not discussed in this dissertation.
Benchmark Circuit (netlist format)

Place circuit using VPR

Route circuit using VPR

Routing successful?

no

Increase core size

yes

Generate VHDL description for core of that size

Synthesize VHDL description using Design Analyzer

Report core area (in square microns) from Design Analyzer

Figure 6.1: Experimental methodology
For each circuit, we found the minimum-size square core on which the circuit can be placed and routed using VPR. Once we found the size of the fabric required to implement a user circuit, we created a VHDL description of a core of that size using each of the architectures. We then synthesized the VHDL description using Synopsys Design Analyzer. The technology library we used was a 0.18μm CMOS library available from TSMC. After synthesizing each core, the total cell area (in square microns) was reported from Design Analyzer. We have completed the physical design of some of the cores using Cadence tools, and have determined that there is a good fidelity between the Design Analyzer estimates and the final chip area, so we use the Design Analyzer estimates in this work.

6.2 Optimal sizing for the Gradual Architecture

In the Gradual architecture, there are various architectural parameters whose values can be varied. These parameters include the I/O ratio, the number of horizontal multiplexers used to route primary inputs, and the size of the LUT to be used in the architecture. Given that the values of these parameters can have a significant effect on the obtained area results, it is necessary to determine the best values for these parameters. Particularly, we consider two of these parameters: I/O ratio and LUT size. In the following subsections, we describe experiments to determine the I/O ratio value and LUT size, and we present the results.

6.2.1 Gradual I/O Results

The I/O ratio determines the number of input/output pins per row of logic. If the I/O ratio is chosen to be too small then the number of rows/columns of programmable logic necessary to implement a user circuit will become large (since the circuit is now input/output bound). This means that many of the logic blocks in the fabric will be unused, thus introducing area
overhead. Similarly, if the I/O ratio is chosen to be too large then the programmable logic will contain many unused input and output pins. This also causes a waste of area since the size of many internal multiplexers scales with the number of primary inputs and outputs.

We started with an educated guess of an I/O ratio of two (i.e. two inputs and two outputs per row). We varied the value of this parameter, and we followed the experimental methodology described earlier to determine the area required to implement each of the benchmark circuits. We then graphed the geometric average area for all benchmark circuits for a variety of I/O ratios. A fixed lookup table size of three was used in all the experiments. From Figure 6.2 it can be seen that the best value was found to be an I/O ratio of four.

![Figure 6.2: Gradual Architecture average area for various I/O ratio values](image)

**6.2.2 Gradual LUT size Results**

The LUT size (where a LUT size of \( k \), refers to a look-up table with \( k \) inputs) is another important parameter we consider. A LUT size of four is commonly used in industry and in most research work. However, the Gradual architecture is not similar to any of the existing
architectures. In the gradual architecture the LUT selection multiplexers, along with the configuration bits required to program them, constitute a significant amount of area. Consequently, if the LUT size used is too large, more of these multiplexers will be required. Yet many of them might not be used and this would result in substantial waste of area. On the other hand, if the LUT size is too small, the core size required to implement the user circuits can significantly grow, again resulting in an increase in area. Thus, the choice of LUT size is very important.

We limit our investigation to three LUT sizes. First, we followed the previously described experimental methodology to find out the core area required to implement each of the benchmark circuits using a LUT size of three. We then averaged all the obtained area measurements. This process was repeated for LUT sizes of four and five. The I/O ratio is fixed at four for all the experiments. Figure 6.3 shows the obtained results. As can be seen from the graph, a LUT size of three results in the most area efficient architecture.

Figure 6.3: Gradual Architecture average area for various LUT sizes
6.3 Gradual Architecture vs. Directional Architecture

As previously mentioned, we suspect that the allowable flexibility in the Directional Architecture is more than what is required to implement small circuits. Intuitively, this would mean that the Gradual Architecture may be more suitable for implementing small circuits. In this section, we seek to confirm this intuition by quantifying any area used to implement benchmark circuits in both the Gradual Architecture and the Directional Architecture. Since the previous results suggest that using an I/O ratio of four and a LUT size of three is most suitable for the Gradual Architecture, these values were fixed in our experiments. Recall that the I/O pads in the Directional Architecture are along the perimeter of the core. Whereas in the Gradual Architecture there are I/O pads only along the right and left sides of the core. In other words, the Directional architecture contains two rows of I/O pads (along the top and bottom of the core) more than the Gradual Architecture. To compensate for this, we use an I/O ratio of two for the Directional Architecture.

The results are tabulated in Table 6.1. The first four columns show the results for the Directional Architecture. For each benchmark circuit, we varied both the core size and the number of tracks in each channel, and chose the configuration which resulted in the minimum area; the chosen size and channel width is shown in Columns Two and Three of the table. Column four of the table shows the area required to implement the core described by the parameters in Columns Two and Three, as reported from Design Analyzer.

The final three columns show the results for the Gradual Architecture. In this case, we varied both the core size and the number of input multiplexers per row, and chose the configuration which resulted in the lowest area. These numbers are reported in Columns Five and Six of the table, and the synthesized cell area from Design Analyzer is shown in the final column. Examining the results, we see that they do indeed confirm our intuition. The results show that
when the Gradual Architecture is used, the geometric average of the area required to implement the circuits on the Gradual Architecture is 18.9% less than that required to implement the same circuits using the Directional Architecture.

<table>
<thead>
<tr>
<th>Benchmark Circuit</th>
<th>Directional Architecture</th>
<th>Gradual Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FPGA Core Size</td>
<td>Tracks per</td>
</tr>
<tr>
<td>cc</td>
<td>9x9</td>
<td>4</td>
</tr>
<tr>
<td>cm138a</td>
<td>5x5</td>
<td>3</td>
</tr>
<tr>
<td>cm150a</td>
<td>9x9</td>
<td>4</td>
</tr>
<tr>
<td>cm151a</td>
<td>5x5</td>
<td>3</td>
</tr>
<tr>
<td>cm152a</td>
<td>4x4</td>
<td>3</td>
</tr>
<tr>
<td>cm162a</td>
<td>5x5</td>
<td>4</td>
</tr>
<tr>
<td>cm163a</td>
<td>6x6</td>
<td>5</td>
</tr>
<tr>
<td>cm42a</td>
<td>5x5</td>
<td>4</td>
</tr>
<tr>
<td>cm82a</td>
<td>4x4</td>
<td>3</td>
</tr>
<tr>
<td>cm85a</td>
<td>6x6</td>
<td>4</td>
</tr>
<tr>
<td>cmb</td>
<td>7x7</td>
<td>3</td>
</tr>
<tr>
<td>comp</td>
<td>12x12</td>
<td>4</td>
</tr>
<tr>
<td>con1</td>
<td>4x4</td>
<td>3</td>
</tr>
<tr>
<td>count</td>
<td>12x12</td>
<td>5</td>
</tr>
<tr>
<td>cu</td>
<td>8x8</td>
<td>3</td>
</tr>
<tr>
<td>5xpl</td>
<td>11x11</td>
<td>5</td>
</tr>
<tr>
<td>l1</td>
<td>8x8</td>
<td>3</td>
</tr>
<tr>
<td>Inc</td>
<td>10x10</td>
<td>5</td>
</tr>
<tr>
<td>unreg</td>
<td>10x10</td>
<td>4</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geo. Avg.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Directional and Gradual Architecture results

6.4 Gradual Architecture vs. Segmented Architecture

In the previous section we demonstrated that the Gradual Architecture is more area efficient than the Directional Architecture. In this section, we compare the area required to implement the benchmark circuits using the Segmented Architecture to that required to implement the circuits using the Gradual Architecture. Again, the I/O ratio and LUT size are held constant (at four and three respectively) throughout our experiments.
In Table 6.2, the first four columns show the results for the Segmented Architecture. In this architecture the number of tracks per channel is always the same as the number of rows in the core. Hence, only the core size was varied. For each benchmark circuit, we found the minimum sized core on which the circuit could be implemented. The chosen size and channel width is shown in Columns Two and Three of the table. For each configuration, we then synthesized the architecture using Design Analyzer; the fourth column in the table shows the cell area required to implement the core.

The same results for the Gradual Architecture are shown in the final three columns of the table. As the table shows, the geometric average of the area required to implement the circuits on the Gradual Architecture is 25.3% less than that required to implement the same circuits using the Segmented Architecture. This substantial increase in area can likely be attributed to the very restrictive nature of the Segmented Architecture. Although the area of a Segmented Architecture tile is less than that of an equivalent Gradual or Directional tile, the routing structure is much more restrictive. This means that more tiles are required to implement a given benchmark circuit. The effect of this is more pronounced in larger circuits.

<table>
<thead>
<tr>
<th>Benchmark Circuit</th>
<th>Segmented Architecture</th>
<th>Gradual Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FPGA Core Size</td>
<td>Tracks per Channel</td>
</tr>
<tr>
<td>Cc</td>
<td>13x13</td>
<td>13</td>
</tr>
<tr>
<td>Cm138a</td>
<td>5x5</td>
<td>5</td>
</tr>
<tr>
<td>Cm150a</td>
<td>9x9</td>
<td>9</td>
</tr>
<tr>
<td>Cm151a</td>
<td>5x5</td>
<td>5</td>
</tr>
<tr>
<td>Cm152a</td>
<td>4x4</td>
<td>4</td>
</tr>
<tr>
<td>Cm162a</td>
<td>7x7</td>
<td>7</td>
</tr>
<tr>
<td>Cm163a</td>
<td>7x7</td>
<td>7</td>
</tr>
<tr>
<td>Cm42a</td>
<td>6x6</td>
<td>6</td>
</tr>
<tr>
<td>Cm82a</td>
<td>4x4</td>
<td>4</td>
</tr>
<tr>
<td>Cm85a</td>
<td>6x6</td>
<td>6</td>
</tr>
<tr>
<td>Cmb</td>
<td>7x7</td>
<td>7</td>
</tr>
<tr>
<td>Comp</td>
<td>11x11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>4x4</td>
<td>4</td>
</tr>
<tr>
<td>Count</td>
<td>18x18</td>
<td>18</td>
</tr>
<tr>
<td>Cu</td>
<td>8x8</td>
<td>8</td>
</tr>
<tr>
<td>5xpl</td>
<td>12x12</td>
<td>12</td>
</tr>
<tr>
<td>il</td>
<td>12x12</td>
<td>12</td>
</tr>
<tr>
<td>Inc</td>
<td>12x12</td>
<td>12</td>
</tr>
<tr>
<td>Unreg</td>
<td>15x15</td>
<td>15</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geo. Avg.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Gradual and Segmented Architecture results

6.5 Soft vs. Hard Programmable Logic Cores

As mentioned in Chapter 3, the primary disadvantage of using a “soft” programmable logic core is the reduced density, speed, and increased power consumption. In this section, we estimate the density of a soft core compared to a hard core (comparing the two in terms of speed or power is left as future work).

The most accurate way to compare the area required by soft and hard programmable logic cores would be to lay out (by hand) a hard core, and compare its area with the numbers in Table 5.1. This is a time-consuming task. Instead, we estimate the size of a hard core using a detailed transistor-count model, following the methodology described in [49]. We focus on a 4x4 Gradual Architecture with three input multiplexers per row (since the previous results have shown this to be the most efficient architecture). By estimating the number of Minimum Transistor Equivalents (MTE's) required to implement the circuit, and converting this to area in our 0.18μm technology, we estimate the layout of such a core to require 12 868μm². A soft core was generated using these same parameters, and the size (after synthesis using Design Analyzer and physical design using Cadence) was 81 092μm². Thus, the synthesized core is approximately 6.4x less dense than the hard core.
This number is significant. Clearly, for large programmable logic cores, our approach would not be suitable. However, if only small amounts of programmable logic are required, this density penalty may be acceptable. In addition, the use of a hard-core will usually require the selection of a core from a library. Since it is unlikely that a library would contain all sizes and shapes of cores, in most cases, a designer would end up choosing a larger core than is required. Using a soft core, the designer can create a core of any size. Thus, the penalty may not be as bad as the above number suggests.

We have also compared our sizes to commercial FPGA layouts using publicly available information from Chipworks. These comparisons yielded little insight, however, since the commercial devices contain far more tracks per channel, and contain additional elements such as flip-flops.

6.6 Sensitivity of Results

As described in [54], it is critical to analyze results for their sensitivity to experimental assumptions. Table 6.3 shows two of our sensitivity results for the data in Table 6.1. The first part of the table shows how the conclusions change if we alter the number of input/output connections per grid. In the experiments in Section 6.3, it was assumed that an \( n \times n \) Directional Architecture has \( 2n \) input/output connections along each of the four edges of the core, and that an \( n \times n \) Gradual Architecture has \( 4n \) input/output connections along the left and right edges of the core. We tried two other input/output ratios, and gathered the results in Table 5.3. Although the Gradual Architecture always gave better density than the Directional architecture, the margin by which the Gradual was better varied. According to the methodology in [54], we classify this experiment as sensitive to the input/output ratio, even though the conclusion was the same in all cases.
The second part of the table shows how a less aggressive placement schedule (fewer moves per temperature and larger temperature drops during the annealing) and routing schedule (fewer routing attempts) affects the conclusions. In this case, the margin was smaller, meaning the experiment was only slightly sensitive to the choice of algorithm.

<table>
<thead>
<tr>
<th>Half as many I/O connections</th>
<th>9.67 %</th>
<th>Percent Difference, baseline algorithms</th>
<th>18.9 %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>18.9 %</td>
<td>Percent Difference, fast algorithms</td>
<td>15.5 %</td>
</tr>
<tr>
<td>Twice as many I/O connections</td>
<td>2.33 %</td>
<td>Margin</td>
<td>3.4 %</td>
</tr>
<tr>
<td>Margin</td>
<td></td>
<td>Conclusion</td>
<td>Slightly Sensitive</td>
</tr>
<tr>
<td>Conclusion</td>
<td>Sensitive</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: Sensitivity results

6.7 Summary

In this chapter we presented the results of several experiments that were conducted. First, the experimental methodology was explained in detail. We then performed experiments to determine the optimum values for some parameters in the Gradual Architecture. More specifically, we conducted experiments to determine the best I/O ratio and LUT size values. Using the geometric average of the core area required to implement a set of benchmark circuits as an evaluation measure; our results show that using an I/O ratio of four and a LUT size of three yield the best results.

A comparison was also carried out between the Gradual and Directional Architectures, and we presented results that demonstrate that the Gradual Architecture requires 18.9% less area, (on average) to implement the same set of benchmark circuits on the Directional Architecture. We also compared the Gradual Architecture to the Segmented and we showed that implementing the same set of user circuits on the Segmented Architecture requires 25.3% more area than an equivalent implementation using the Gradual Architecture.
Further, we have shown that a 4x4 Gradual Architecture soft core is 6.4 times bigger than a hard core with an equivalent logic capacity.

Finally, we presented the results of experiments conducted to show how sensitive our experimental results are to various architectural parameters and CAD tool settings. We considered the area required to implement the set of benchmark circuits using half and double the baseline I/O ratio value (i.e. we determined the effect of using an I/O ratio of two and eight on our results). Based on the sensitivity calculation method in [54], we have found our results to be sensitive to variations in the I/O ratio by a margin of 9.21%. Another aspect we considered is the sensitivity due to the CAD tool settings. In this case we changed the VPR setting to a “fast” mode, in which fewer placement and routing iterations are performed. Again, as per the method described in [54], we found that our results are slightly sensitive to this variation by a margin of 3.4%.
7 FURTHER AREA OPTIMIZATIONS

In Chapter 6, we showed that the Gradual Architecture is more efficient than the Directional or Segmented Architectures. In this chapter, we present three enhancements to this architecture to further improve its density. These enhancements require enhancements to the associated placement algorithms; these are also described in this chapter. More specifically, we evaluate the following approaches to improve the area efficiency of the architecture:

1. As previously mentioned, a key benefit of using a soft core is the freedom to generate irregularly shaped cores. Since we are not restricted to a particular core shape, we explore the possibility of eliminating some logic blocks. We discuss the concept of creating non-rectangular cores in Section 7.1, and we present some experimental results.

2. In Section 7.2, we investigate the gains of reducing the size of the LUT input selection multiplexers throughout the core (with the exception of those in the first column). Since the placement algorithms dictate, to a large extent, how easily user circuits can be implemented on our architecture; we include the modifications to the placement algorithm necessary to accommodate this architectural change. A discussion of the experimental results is also included.

3. Recall that our architectures demand significant flexibility on the inputs of the core. As a result, the LUT input selection multiplexers in the first column are used to select between all the primary inputs; thus these multiplexers are larger than the LUT input selection multiplexers in the rest of the core. Therefore, in Section 7.3, we to attempt to further
improve density by reducing the size of the LUT input selection multiplexers in the first column.

Finally, the chapter concludes with a summary presented in Section 7.4.

7.1 Non-Rectangular Fabric

The grid of logic blocks in standard FPGAs is square or rectangular. From [55], however, logic circuits often have a "triangular" shape. In standard FPGAs, this is not a problem, since the routing resources are flexible enough that signals can be routed left, right, up or down, as shown in Figure 7.1(a). This means that in a standard FPGA, the physical implementation of a circuit need not match the shape of the circuit. In the architectures described in this thesis, however, the signal flow is restricted from left to right. As shown in Figure 7.1(b), this can lead to unused logic blocks if the circuit does not have a naturally square shape.

We can alleviate this problem somewhat by creating a programmable logic core that is not square. We have observed that in many implementations, several logic blocks in the rightmost columns remain unused. We can take advantage of this by removing logic blocks from the last few columns, as shown in Figure 7.1(c). We quantify the number of logic blocks removed using the parameter $\epsilon$, where $\epsilon$ is defined as the proportion of the logic blocks in the top row that have been removed. In Figure 7.1(c), $\epsilon$ is 2/3. In all cases, we remove blocks in a "triangular" fashion; if we remove $m$ blocks from column $i$, we remove $m-1$ blocks from column $i-1$. A value of 0 for $\epsilon$ indicates a rectangular core; a value of 1 indicates a triangular core. Note that a non-zero value of $\epsilon$ does not imply a non-rectangular layout. The diagram in Figure 7.1(c) is a conceptual representation; the core will be synthesized into gates, and the gates will be placed into rows regardless of the shape of the conceptual representation.
Intuitively, as $c$ is increased, the area of the implementation will go down. If $c$ is decreased too much, however, the area will rise, since a larger grid will be needed. This can be seen in Figure 7.2. Figure 7.2(a) shows how the implementation area depends on $c$ for each circuit implemented on the Gradual Architecture (one line per circuit). Because we had problems synthesizing large triangular cores using our synthesis tools, results are only shown for 11 of the 19 benchmark circuits. The geometric average over these 11 circuits is shown in Figure 7.2(b).

Although each individual circuit in Figure 7.2(a) shows the expected trends, the results in Figure 7.2(b) indicate that the gain obtained using a non-zero value of $c$ is small. From Figure 7.2(a), the “breakpoint” (the point at which a larger grid is needed) is not the same for each circuit. Thus, the average results show that only a modest improvement can be achieved. Overall, the value of $c$ that gave the lowest area was 0.6, which resulted in 11.1% lower area than a square core, averaged over all circuits.

![Diagram of circuits](image)

Figure 7.1: Implementing a circuit on a triangular core
7.2 Reducing LUT Input Multiplexers

From Figure 4.4, we see that the multiplexers selecting the inputs of the LUTs make up a significant amount of the core area. Furthermore, the architecture contains a considerable number of these multiplexers. Thus, we would expect that optimizing these multiplexers may lead to significant overall density improvements.

7.2.1 Architectural Modifications

Reducing the size of the multiplexers implies that the overall flexibility of the architecture will be reduced. However, a reduction in flexibility may increase the core size on which a circuit can be successfully placed and routed. It is essential to find a good compromise between flexibility and core size, such that the area gains from reducing flexibility are not overcome by the area increase resulting from using a larger core. We attempted to achieve such a compromise by changing two characteristics of the multiplexers. First, we reduced the flexibility of the LUT input multiplexers by only allowing their inputs to come from the horizontal tracks above the row containing the LUT. Second, we divide the vertical tracks between the three LUT input selection multiplexers. In other words, each LUT input
multiplexer has access to only one third of the vertical tracks in the previous column. This change is illustrated in the Figure 7.3.

![Figure 7.3: Reduced LUT input multiplexers Gradual Architecture](image)

**Figure 7.3: Reduced LUT input multiplexers Gradual Architecture**

### 7.2.2 Placement Algorithm Modifications

Architectural changes usually require a change in the associated CAD tools. It is meaningless to suggest any architectural enhancements if they will not be exploited by the CAD tools. In our case, the enhancement proposed in this section places new demands on the placement algorithm. In the architecture shown in Figure 7.3, each LUT can only route a signal to a LUT in the adjacent column via a single LUT input selection multiplexer. Further, the LUT input selection multiplexers are now shared resources and are susceptible to congestion. In a 4x4 core, for example, the first and fourth vertical tracks in each column will connect to the same
LUT input selection multiplexer. Hence, it is possible that two nets may compete to use the same multiplexer and there is no other option for either net. It is crucial to try to avoid such a situation. To ensure this, we need to reflect the overuse of the LUT input selection multiplexers in the placement algorithm.

Our cost function is similar to that described previously in Chapter 5. A primary difference though, is that the occupancy and capacity of the LUT input selection multiplexers are now accounted for in our cost calculation. Figure 7.4 illustrates how we consider the use of these multiplexers. Figure 7.5 shows the modified placement algorithm.

![Figure 7.4: Example placements on modified Gradual Architecture](image)

Figure 7.4: Example placements on modified Gradual Architecture
/* Initialization*/
total_cost=0
for each mux m {
    mux_occupancy=0
    mux_capacity="Number of output signals that can be passed with this multiplexer"
    for each lut input mux lut_m {
        lut_mux_occupancy=0
        lut_mux_capacity=0
    }
}
for each row r {
    histogram[r]=0
}

/* Set mux occupancies */
for each net n {
    source_x="x-coordinate of the source signal"
    for each sink s {
        sink_x="x-coordinate of the sink s"
        sink_y="y-coordinate of the sink s"
        if sink_x>source_x+1 {
            histogram[sink_y]=1
        }
        if sink_x=source_x+1 {
            lut_mux_occupancy[sink_x][sink_y][source_y%lut_size]+=1
        }
    }
}
for each row r {
    if histogram[r]=1 {
        mux_occupancy[source_x][r]+=1
    }
}

/* Calculate cost for each mux */
for each mux m {
    if mux_occupancy < mux_capacity {
        single_mux_cost(m)=0
    } else {
        single_mux_cost(m)=[mux_occupancy(m) + 0.2 - mux_capacity(m)]
    }
}

/* Calculate total cost */
for each mux m {
    total_cost=total_cost + single_mux_cost(m)
}

Figure 7.5: Placement algorithm pseudo-code for modified Gradual Architecture
7.2.3 Experimental Results

To evaluate the proposed enhancements, we used the same 19 MCNC benchmark circuits used earlier. An experimental approach identical to that described in Section 6.1 was used. For each circuit, we determined the minimum size square core on which the circuit could be successfully placed and routed using our enhanced placement algorithm. Then we obtained the area of a VHDL implementation of such a core, using the modified architecture, from Design Analyzer.

Table 7.1 shows the results. As was the case in Section 6.3, we vary the number of horizontal multiplexers for each circuit. The first four columns of the table show the results for the original Gradual Architecture. The remaining three columns contain the results for the modified architecture. Column Five shows the core size required to implement the user circuit. Column Six shows the number of horizontal multiplexers used and Column Seven shows the total cell area from Design Analyzer.

<table>
<thead>
<tr>
<th>Benchmark Circuit</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gradual Architecture</td>
<td>Modified Gradual Architecture</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Benchmark Circuit</td>
<td>FPGA Core Size</td>
<td>Input muxes per row</td>
<td>Cell Area ($\mu$m$^2$)</td>
<td>FPGA Core Size</td>
<td>Input muxes per row</td>
<td>Cell Area ($\mu$m$^2$)</td>
</tr>
<tr>
<td>cc</td>
<td>8x8</td>
<td>3</td>
<td>263 101</td>
<td>8x8</td>
<td>5</td>
<td>220 437</td>
</tr>
<tr>
<td>cm138a</td>
<td>5x5</td>
<td>1</td>
<td>78 319</td>
<td>5x5</td>
<td>3</td>
<td>72 412</td>
</tr>
<tr>
<td>cm150a</td>
<td>8x8</td>
<td>3</td>
<td>263 101</td>
<td>9x9</td>
<td>5</td>
<td>302 013</td>
</tr>
<tr>
<td>cm151a</td>
<td>4x4</td>
<td>1</td>
<td>43 932</td>
<td>4x4</td>
<td>3</td>
<td>43 026</td>
</tr>
<tr>
<td>cm152a</td>
<td>4x4</td>
<td>1</td>
<td>43 932</td>
<td>4x4</td>
<td>1</td>
<td>32 919</td>
</tr>
<tr>
<td>cm162a</td>
<td>5x5</td>
<td>2</td>
<td>89 614</td>
<td>5x5</td>
<td>5</td>
<td>86 203</td>
</tr>
<tr>
<td>cm163a</td>
<td>5x5</td>
<td>2</td>
<td>89 614</td>
<td>5x5</td>
<td>5</td>
<td>86 203</td>
</tr>
<tr>
<td>cm42a</td>
<td>5x5</td>
<td>2</td>
<td>89 614</td>
<td>5x5</td>
<td>2</td>
<td>65 618</td>
</tr>
<tr>
<td>cm82a</td>
<td>5x5</td>
<td>2</td>
<td>89 614</td>
<td>5x5</td>
<td>2</td>
<td>8 517</td>
</tr>
<tr>
<td>cm85a</td>
<td>6x6</td>
<td>2</td>
<td>128 822</td>
<td>6x6</td>
<td>2</td>
<td>93 358</td>
</tr>
<tr>
<td>cm58a</td>
<td>7x7</td>
<td>2</td>
<td>184 590</td>
<td>7x7</td>
<td>4</td>
<td>157 502</td>
</tr>
<tr>
<td>comp</td>
<td>11x11</td>
<td>2</td>
<td>542 489</td>
<td>13x13</td>
<td>4</td>
<td>669 383</td>
</tr>
<tr>
<td>con1</td>
<td>4x4</td>
<td>1</td>
<td>43 932</td>
<td>4x4</td>
<td>4</td>
<td>48 726</td>
</tr>
<tr>
<td>count</td>
<td>10x10</td>
<td>4</td>
<td>487 588</td>
<td>11x11</td>
<td>6</td>
<td>517 848</td>
</tr>
<tr>
<td>cu</td>
<td>8x8</td>
<td>2</td>
<td>244 676</td>
<td>8x8</td>
<td>4</td>
<td>206 630</td>
</tr>
<tr>
<td>5xpl</td>
<td>11x11</td>
<td>2</td>
<td>542 489</td>
<td>12x12</td>
<td>6</td>
<td>616 273</td>
</tr>
<tr>
<td>il</td>
<td>7x7</td>
<td>2</td>
<td>184 590</td>
<td>7x7</td>
<td>5</td>
<td>169 759</td>
</tr>
</tbody>
</table>
As shown in Table 7.1, the number of logic required to implement a specific circuit either increases or stays the same in the enhanced architecture. If the core size remains unchanged, the number of horizontal multiplexers usually increases. Reducing flexibility increases the likelihood of congestion; this may increase the size of the core required or increase the number of horizontal multiplexers required. Despite the increase in the number of logic blocks, we see that there is an 8.9% reduction in area. This is due to the fact that the reduction in the size of the LUT selection multiplexers makes up for the increase in the core size. The area gains are more pronounced in larger circuits (circuits that require a 7x7 core or larger using the original Gradual Architecture). There are two reasons for this. First, a larger core contains more LUT input selection multiplexers. Second, the size of these multiplexers is larger in larger cores. Hence, the effect of reducing these multiplexers is more noticeable in larger cores.

We notice that for smaller circuits the core size as well as the area required by the circuits actually increases. There are two reasons for this. The first reason is that smaller cores do not contain many LUT input selection multiplexers. Another reason is that the LUT input selection multiplexers in small cores are not very large in size. Therefore, the reduction in the area of these multiplexers is not very significant. In most cases, the increase in the horizontal multiplexers required to compensate for the reduced flexibility results in an overall increase in area.

<table>
<thead>
<tr>
<th></th>
<th>inc</th>
<th>10x10</th>
<th>2</th>
<th>424 445</th>
<th>12x12</th>
<th>5</th>
<th>287 940</th>
</tr>
</thead>
<tbody>
<tr>
<td>unreg</td>
<td>9x9</td>
<td></td>
<td>4</td>
<td>388 074</td>
<td>9x9</td>
<td>5</td>
<td>302 013</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>218 009</td>
</tr>
<tr>
<td>Geo. Avg.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>141 954</td>
</tr>
</tbody>
</table>

Table 7.1: Reduced LUT input multiplexers Gradual Architecture results
7.3 Reducing LUT Input Multiplexers In First Column

A natural extension to the enhancement proposed in the previous section, is to reduce the size of the LUT input selection multiplexers in the first column. Recall that these multiplexers are different from the LUT input selection multiplexers in the rest of the core. A fundamental requirement of our architectures is to have lots of flexibility on the inputs and outputs. Hence, these multiplexers have access to any of the primary inputs of the core. As a result, these multiplexers are large. Therefore, attempting to reduce the area of these multiplexers was the next optimization we considered.

7.3.1 Architectural Modifications

Figure 7.6 shows our enhanced architecture. As a compromise between flexibility and area, we chose to reduce the input multiplexer flexibility such that each LUT input in the first column can be taken from any of one third of the inputs.

![Input and Output Diagram](image)

Figure 7.6: Further reduced Gradual Architecture
7.3.2 Placement Algorithm Modifications
Again, our architectural change necessitates a change in the placement algorithms. Similar to the change described in Section 7.2.2, we maintain an occupancy and capacity value for each LUT input selection multiplexer and we include them in the cost calculation to account for the overuse of these multiplexers.

7.3.3 Experimental Results
In this subsection we present the experimental results to evaluate the effectiveness of the proposed architectural modification. Again, the same 19 circuits from the MCNC benchmark suite were used to evaluate the performance of the new limited Gradual Architecture. We employ the same experimental methodology described in Section 6.1.

For each circuit, we varied the core size and the number of horizontal multiplexers. We chose the smallest configuration on which the circuit could be implemented, and we generated a VHDL description of a core of such a configuration. The core description was synthesized using Design Analyzer and total cell area required to implement the core was obtained. In Table 7.2, the first four columns correspond to the original Gradual Architecture presented in Chapter 4. Columns Five and Six of the table show the core size and the number of horizontal multiplexers used, respectively, for the enhanced architecture shown in Figure 7.6. Finally, the last column of the table shows the cell area necessary to implement the circuit using the modified architecture.

<table>
<thead>
<tr>
<th>Benchmark Circuit</th>
<th>Gradual Architecture</th>
<th>Modified Gradual Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FPGA Core Size</td>
<td>Tracks per Channel</td>
</tr>
<tr>
<td>cc</td>
<td>8x8</td>
<td>3</td>
</tr>
<tr>
<td>cm138a</td>
<td>5x5</td>
<td>1</td>
</tr>
<tr>
<td>cm150a</td>
<td>8x8</td>
<td>3</td>
</tr>
</tbody>
</table>
An interesting observation is that the average area required to implement the set of benchmark circuits increases in the enhanced architecture. Although the input multiplexers are smaller, this is overshadowed by an increase in the number of LUTs required to implement each circuit. The area increase, averaged over all the benchmark circuits, is 12%. Based on this result, we conclude that this modification leads to a very restrictive architecture which in turn results in congestion requiring a significant increase in the core size or the number of horizontal multiplexers. As a result, this change results in an architecture with reduced density.

### 7.4 Summary

In this chapter, we presented three optimizations to the Gradual Architecture introduced in Chapter 4. First, we investigated the possibility of reducing the area of the architecture by removing unused logic blocks. This is an option available to us because we are not restricted to using a core of a certain shape, since we are synthesizing our cores. According to previous work done in [55], combinational circuits tend to be triangular in shape. Thus, we investigated

<table>
<thead>
<tr>
<th>cm151a</th>
<th>4x4</th>
<th>1</th>
<th>43 932</th>
<th>5x5</th>
<th>3</th>
<th>72 424</th>
</tr>
</thead>
<tbody>
<tr>
<td>cm152a</td>
<td>4x4</td>
<td>1</td>
<td>43 932</td>
<td>4x4</td>
<td>4</td>
<td>48 738</td>
</tr>
<tr>
<td>cm162a</td>
<td>5x5</td>
<td>2</td>
<td>89 614</td>
<td>6x6</td>
<td>4</td>
<td>112 970</td>
</tr>
<tr>
<td>cm163a</td>
<td>5x5</td>
<td>2</td>
<td>89 614</td>
<td>5x5</td>
<td>5</td>
<td>86 178</td>
</tr>
<tr>
<td>cm42a</td>
<td>5x5</td>
<td>2</td>
<td>89 614</td>
<td>5x5</td>
<td>2</td>
<td>65 618</td>
</tr>
<tr>
<td>cm82a</td>
<td>2x2</td>
<td>1</td>
<td>9 261</td>
<td>3x3</td>
<td>3</td>
<td>24 092</td>
</tr>
<tr>
<td>cm85a</td>
<td>6x6</td>
<td>2</td>
<td>128 822</td>
<td>6x6</td>
<td>4</td>
<td>112 970</td>
</tr>
<tr>
<td>cmb</td>
<td>7x7</td>
<td>2</td>
<td>184 590</td>
<td>7x7</td>
<td>5</td>
<td>169 759</td>
</tr>
<tr>
<td>comp</td>
<td>11x11</td>
<td>2</td>
<td>542 489</td>
<td>13x13</td>
<td>4</td>
<td>641 891</td>
</tr>
<tr>
<td>con1</td>
<td>4x4</td>
<td>1</td>
<td>43 932</td>
<td>4x4</td>
<td>3</td>
<td>43 050</td>
</tr>
<tr>
<td>count</td>
<td>10x10</td>
<td>4</td>
<td>487 588</td>
<td>11x11</td>
<td>5</td>
<td>468 378</td>
</tr>
<tr>
<td>cu</td>
<td>8x8</td>
<td>2</td>
<td>244 676</td>
<td>8x8</td>
<td>4</td>
<td>206 630</td>
</tr>
<tr>
<td>5xpl</td>
<td>11x11</td>
<td>2</td>
<td>542 489</td>
<td>12x12</td>
<td>6</td>
<td>588 331</td>
</tr>
<tr>
<td>i1</td>
<td>7x7</td>
<td>2</td>
<td>184 590</td>
<td>7x7</td>
<td>5</td>
<td>169 759</td>
</tr>
<tr>
<td>inc</td>
<td>10x10</td>
<td>2</td>
<td>424 445</td>
<td>16x16</td>
<td>5</td>
<td>1 068 051</td>
</tr>
<tr>
<td>unreg</td>
<td>9x9</td>
<td>4</td>
<td>388 074</td>
<td>9x9</td>
<td>5</td>
<td>302 013</td>
</tr>
</tbody>
</table>

| Average | 218 009 |  | | 255 380 |
| Geo. Avg. | 141 954 |  | | 158 885 |

Table 7.2: Further reduced Gradual Architecture results
triangular cores in Section 7.1. We showed that this technique resulted in an 11% improvement in area, averaged over our benchmark circuits.

In Section 7.2, we investigate the benefits of reducing the size of the LUT input selection multiplexers throughout the core, with the exception of those in the first column. We also described the necessary placement algorithm modification in order to evaluate this modified architecture. Using this enhanced architecture, we were able to reduce the average area by 8.9% compared to the baseline Gradual Architecture. In Section 7.3, we considered reducing the size of the LUT input selection multiplexers in the first column of the core. We showed that the resulting architecture becomes difficult to route, causing the area to increase. We have found such an architecture to increase the average area by 12% compared to the baseline Gradual Architecture.
Chapter 8

8 CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

In this thesis we have presented architectures and CAD algorithms for synthesizable programmable logic cores. With the growing demand for SoC design, it is desirable to include a programmable logic core on chip. Currently, this can be achieved by integrating a hand-layout of a programmable logic core ("hard" core) form a third party vendor with the rest of the design. However, if only small circuits are implemented on these cores, another viable solution is to use synthesizable programmable logic cores. Synthesizable programmable logic cores ("soft" cores) are different in that the cores are obtained as a HDL description, and synthesized using standard synthesis tools. The "soft" core approach is beneficial in many ways:

1. It is easy to integrate "soft" cores with the rest of the fixed logic.

2. A core of any shape or size can be generated.

3. Technology scaling does not require the hand-layout of a new core.

In Chapter 3, we detailed a method for constructing an integrated circuit that contains a soft core. In addition, we stated the advantages and disadvantages of this technique. We described three novel synthesizable FPGA architectures in Chapter 4. A distinguishing factor between our architectures and traditional FPGAs is that they only support combinational circuits, and
are directional, in that signals only flow in one direction through the fabric. This characteristic was incorporated to ensure the compatibility of our architectures with standard synthesis tools. In addition, the interconnect pattern is less flexible and the routing resources are less plentiful.

Existing CAD algorithms are not optimized to target our architectures, therefore new algorithms were required. Chapter 5 explains the placement algorithms developed to enable the implementation of user circuits on our architectures. VPR was the baseline place and route tool of choice. These algorithms were added to VPR to provide support for our architectures.

In Chapter 6, we compared the efficiency of the architectures in terms of area. We found the Gradual Architecture to be most efficient, requiring 18.9% and 25.3% less area (on average), to implement a set of benchmark circuits, compared to the Directional and Segmented Architectures respectively. Among the comparisons we performed is a comparison of a 4x4 Gradual Architecture core to an equivalent hard core implementation. We quantified the area penalty of using a “soft” core rather than a “hard” core, in this case, to be 6.4X. A sensitivity analysis showed that our experimental results are sensitive to the I/O ratio used and the choice of CAD tool algorithms by a margin of 9.21% and 3.4% respectively.

Further Gradual Architecture enhancements were considered in Chapter 7. We found that the average area can be further reduced by 8.9% when the flexibility of the LUT input selection multiplexers throughout the core (with the exception of those in the first column) is reduced. On the other hand, when we reduced the flexibility of the LUT input selection multiplexers in the first column, the architecture became very restrictive and the average area increased by 12%.
8.2 Future Work

As our results show, using a "soft" core incurs a significant area overhead. A main reason contributing to this overhead is that the available standard-cell libraries contain generic cells that are not necessarily optimized for any architecture. However, better synthesis results could be obtained by adding cells which are specifically optimized to implement our programmable logic fabric. We believe that if the standard-cell libraries were expanded to include optimized building blocks, better area results could be achieved. We have not considered this in this work, since our goal was to create architectures that can be implemented using the standard synthesis tools, cell libraries, and design flows that integrated circuit designers are already familiar with. Nonetheless, if this design technique was to become mainstream, specially-designed standard cells could be created.

Other issues that we have not investigated are the speed and power implications of our cores. A current limitation of our CAD algorithms is that we disregard the physical location of the logic blocks on silicon. In other words, we assume that logic blocks which are adjacent in the conceptual representation are equally separated. However, it is not necessarily true that logic block in the same column will be adjacent on silicon or even be the same distance apart. The physical location of the logic blocks is only accurately determined after the core is placed and routed using Cadence tools. Therefore, we suspect that to obtain good speed and power results, some sort of "back-annotation" of detailed routing information is required between Cadence and VPR. A project is currently ongoing at the University of British Columbia to further investigate these issues.

8.3 Summary of Contributions

The contributions of this work are summarized as follows:
1. We introduced the concept of using "soft" programmable logic cores in SoC applications and showed it to be viable. This idea is unique in the sense that an HDL description of the programmable logic core is provided and synthesized, rather than having third party vendors supply a hand-layout of the programmable core.

2. We proposed three novel architectures suitable for this approach; namely, the Directional, Gradual and Segmented Architectures. Our architectures are synthesizable, and contain unique routing structures.

3. New CAD algorithms were developed and integrated into VPR to allow user circuits to be efficiently placed and routed on our architectures.

4. An evaluation of the three architectures was presented, by comparing the average area required to implement a set of benchmark circuits using each of the architectures.

5. We showed that the area penalty introduced by using a "soft" core rather than a "hard" core of similar specifications is approximately 6.4x.

6. We studied the sensitivity of our experimental results to various architectural parameters and CAD tool settings.

7. Finally, we examined the possibility of additionally improving the area performance of the Gradual Architecture using three approaches. We quantified the improvement or degradation in area for each approach.
REFERENCES


