A SYNTHESIS ORIENTED OMNISCIENT MANUAL EDITOR FOR FPGA CIRCUIT DESIGN

by

Tomasz Sebastian Czajkowski

A thesis submitted in conformity with the requirements
For the degree of Master of Applied Science,
Graduate Department of
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto

A Synthesis Oriented Omniscient Manual Editor for FPGA Circuit Design

Tomasz Sebastian Czajkowski
Master of Applied Science, 2003
Graduate Department of
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto

Abstract

Logic circuit designers for Field-Programmable Gate Arrays (FPGAs) put increasing demands on Computer Aided Design (CAD) tools to provide higher logic circuit speeds than are possible using a traditional CAD flow. The problem with the traditional CAD flow lies in that the logic synthesis makes assumptions about how its logic optimizations will affect the speed of the logic circuit post-routing. These assumptions are not always realized by the place and route tool, which leads to a degradation in logic circuit speed. If the post-routing effect of a logic optimization was known, then better logic circuit optimization decisions could be made. This work refers to such knowledge as *omniscience* and explores its applications in the domain of manual physical synthesis.

This work develops a manual editor, named Augur, which uses omniscience in the context of physical synthesis. The user is allowed to select physical synthesis transformations to improve the speed of the logic circuit. After each modification the user is *instantly* informed about the effect of the specified transformation on the speed of the logic circuit. Because of the manual nature of the editor the size of logic circuits is limited to 1000 logic elements.

The manual editor was tested on a suite of 10 logic circuits, all implemented on a Xilinx Virtex-E device. The post-routing timing analysis performed with commercial tools shows that the application of omniscience improves the logic circuit maximum operating frequency by 9.9% on average, with a low area penalty. In addition, several new logic synthesis transformations are developed that arise from the architectural properties of the target device.

Acknowledgements

I would like to take this opportunity to express my thanks to my thesis supervisor, Professor Jonathan Rose, for his guidance, advice and encouragement throughout the course of my research. I wish to express my sincere gratitude for all his help. Without his knowledge and technical expertise this work could not have been realized.

I would like to thank the professors at the University of Toronto who have taught me throughout my undergraduate and graduate studies. I would like to express my gratitude to Professor Stephen Brown, Professor Zvonko Vranesic and Professor Paul Chow who through their teaching encouraged me to pursue graduate studies.

I would also like to thank Dr. Kevin Chung at Xilinx for answering my questions with patience and dedication, and William Chow for providing the software basis for this thesis.

I would like acknowledge my friends in graduate school: Anish Alex, Navid Azizi, Mehrdad Eslami, Valavan Manohararajah and Lesley Shannon for their friendship and technical advice.

My closest friends: Dagmara Biskupska, Mark Bourgeault, Borys Bradel, Jennifer Gee, Henry Jo, Agnieszka and Cezary Piekacz, Gabriel Quan, and Chris Sun. I can only hope that in the years to come I can do justice the friendship you have blessed me with.

I would like to thank my father, Dr. Grzegorz Czajkowski, for everything that he has taught me, both as a teacher and a father. My mother, Tatiana Czajkowska, has always supported me in all my endeavors. My brother, Adam Czajkowski, always reminded me that there is more to life than research.

I would also like to thank my academic thesis supervisor, the University of Toronto, Micronet R&D and Xilinx Corporation for funding this research.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Introduction to FPGA Circuit Design Process

Field-Programmable Gate Arrays (FPGAs) are programmable devices capable of implementing complex digital systems. Implementing digital systems on FPGAs allows designers to test and modify their designs without the need for the FPGA to be re-fabricated, as would be necessary with Application Specific Integrated Circuits (ASICs), saving both time and money. For lower volume products FPGAs provide an inexpensive platform to implement digital logic, while for the high volume products FPGAs can be used to prototype logic circuits before fabrication into ASICs. These benefits made them very popular and pushed them into the mainstream of digital design. As the demand for FPGAs increases, so does the demand for the Computer-Aided Design (CAD) tools, which can achieve higher operating speeds for FPGA logic circuits.

In the traditional FPGA CAD flow, digital circuits are implemented in a sequential process consisting of several stages, as shown in Figure 1-1. A logic circuit is initially represented by a Hardware Description Language (HDL), which the Logic Synthesis stage converts to a set of logic gates that are mapped into lookup tables (LUTs) during the Technology Mapping stage. The Lookup Tables are then placed and routed onto the FPGA during the Placement and Routing stage. The final performance of the circuit is obtained through Timing Analysis. The algorithms governing the optimization decisions made at each stage use cost functions that give only an approximate view of the final effect of any change to the circuit at any level prior to routing and timing analysis.

A more advanced CAD flow, known as physical synthesis, improves the cost functions in

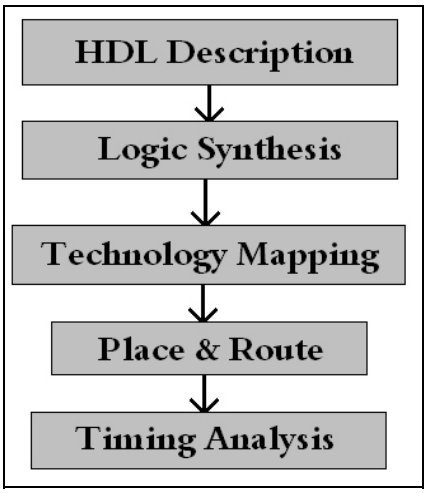

Figure 1-1: Traditional CAD Flow for FPGA circuit design

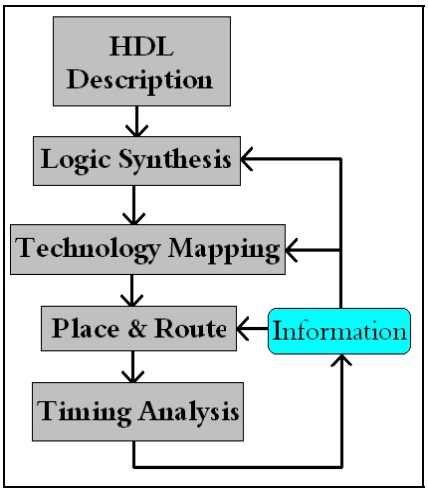

Figure 1-2: The Physical Synthesis approach

the early stages of the design process by providing information from the later stages, as shown in Figure 1-2. This is achieved through iteration of the traditional CAD flow, which provides the cost functions in the early stages with the information about the overall effect of the optimizations on the final circuit speed. With this information the early stages are able to better predict the effect of optimizations they perform.

The goal of this research is to develop a manual editor, which provides the user with the selection of logic synthesis optimizations to improve the speed of a digital circuit. The manual editor adopts a physical synthesis approach, but instead of applying it to the entire circuit, or on *macroscopic* scale, the editor focuses on implementing small and incremental, or *microscopic*, design modifications. After each modification the editor performs routing and timing analysis and provides the user with almost instant feedback about the new circuit speed.

The complete placement, routing and timing analysis performed after each microscopic modification provides the knowledge of the effect of a small change on the post-routed circuit performance. This level of knowledge will be referred to as *omniscience*. The concept of omniscience is used as a guide to evaluate incremental logic synthesis transformations.

The application of omniscience within the manual editor creates a manual CAD tool capable of providing the user with all necessary information to make superior optimization decisions. The manual editor is named **Augur**, after a religious official in ancient Rome, who interpreted "omens"

to guide the public, similarly to the way that the manual editor guides the user to improve the speed of a digital circuit.

Providing omniscience is computationally expensive, as the circuit has to be routed and timing analyzed after each microscopic modification. To make the entire process manageable for the user, this work restricts the size of the circuits Augur is used on. A number of experiments with various circuit sizes have shown that a good user interaction with Augur can be maintained on circuits under 1000 Basic Logic Elements in size.

1.3 Research Goals

There are three goals for this research:

1. To explore the effectiveness of the concept of omniscience in improving the performance of logic circuits,
2. To develop logic synthesis transformations applicable to commercial devices, and
3. To suggest logic circuit optimization strategies that can be applied by automatic CAD tools.

This research uses the concept of omniscience to improve the speed of logic circuits designed for the Xilinx Virtex-E FPGA [7].

1.4 Organization

This dissertation consists of six chapters. Chapter two reviews the necessary background information and terminology used throughout the document. After a brief summary of the user's view of the manual editor, the description of logic synthesis transformations provided by Augur is given in chapter three.

In chapter four, Augur's user interface is presented, along with all necessary information to use Augur successfully. Augur's user manual is followed by the description of the software design of this work.

Chapter five presents results from the use of Augur in improving the speed of FPGA logic circuits. The observations made by the author during this process are utilized to suggest optimization strategies, which might be automated in CAD tools. The conclusion of this thesis and the avenues for future work are presented in chapter six.

2 Background

2.1 Introduction

This chapter introduces concepts and algorithms from the field of logic synthesis required as background to the work presented in this dissertation. Section 2.2 describes prior works related to technology mapping of logic equations onto FPGAs. Section 2.3 describes physical synthesis algorithms. The material covered in these Sections is relevant to the discussion about logic synthesis transformations in chapter 3. We then describe the Xilinx Virtex-E FPGA, which is the target device of this work. A short overview of the work of Chow and Rose [6], which is the basis for this research, is given in Section 2.5.

2.2 Technology Mapping for FPGAs

Technology mapping is a step in logic synthesis in which specific logic components are chosen to implement a desired logic circuit. The mapping of logic gates is performed with the goal of either minimizing the number of logic components that are required to implement a logic circuit or to maximize the speed at which the implemented logic circuit operates. Many works exist that address both of these goals. However, this thesis focuses on improving the speed of logic circuits, and therefore the following Sections will focus on describing the progress made in the technology mapping with the goal of improving the speed of the logic circuit implementation.

This Section describes technology mapping algorithms starting with the basic approach to technology mapping, followed by improvements made to it to increase the speed of the mapped

logic circuit. The three major topics in this Section are mapping logic functions into logic components, reducing the estimated routing delay during technology mapping, and modifying the representation of logic functions to explore a wider range of technology mapping possibilities.

Before any of the above topics can be discussed a common terminology used to describe technology mapping algorithms needs to be presented. This is the purpose of the following subsection.

2.2.1 Terminology

A logic circuit can be represented as graph that consists of nodes and directed edges. The nodes in a graph that represents a logic circuit correspond to logic gates, *primary inputs* and *primary outputs*. A directed edge (a, b) in a graph is present between a pair of nodes *a* and *b* if the output of a node *a* is an input to node *b*. A primary input of a graph represents a primary input to the logic circuit or an output of a flip-flop and has no incoming edges. Similarly, the primary output of a graph corresponds to a primary output of the logic circuit or an input to a flip-flop and has no outgoing edges [4].

For the remainder of this Section a graph representing a logic circuit will be considered to be free of paths that start and end at the same node, also referred to a a graph without cycles. A graph with directed edges and no cycles is called a *Directed Acyclic Graph* (DAG) [27].

In a DAG a node is said to be *K-feasible* [22] if the number of inputs to the node is less than or equal to K. If all nodes in a graph are K-feasible then the graph is said to be a *K-feasible* graph. In this Section we assume that a graph that is to be the input to the technology mapping algorithm is K-feasible, so that any single node, or a group of nodes that implement a function of at most K inputs, can always be implemented in a lookup table that has K inputs (K-LUT). The *depth* of a K-LUT in a logic circuit is determined by the maximum number of K-LUTs on any path from any primary input to that K-LUT. Therefore, the primary input has a depth of 0, while a node whose inputs are only the primary inputs has a depth of 1.

The edges between the nodes in a DAG are directed edges and represent a *connection* between a pair of logic gates. The connection has a source, which is the node where the directed edge originates, and a destination, which is where the connection terminates. A set of connections

that have the same source are also referred to as a *net*.

In the following subsection the above terminology is used to describe several technology mapping algorithms. The description of the technology mapping algorithms begins with the basic approach to technology mapping.

2.2.2 Basic Approach

A basic approach to perform FPGA technology mapping, developed by Chen *et. al.* [21], uses a DAG representation of a logic circuit as input and produces a functionally equivalent graph, where each node is a K-LUT. The algorithms consists of two steps: labeling and mapping. In the labeling step the algorithm traverses all nodes in the circuit in the topological order and assigns a depth to each node. In the mapping step, the algorithm processes the nodes starting at the primary outputs of the circuit towards the primary inputs. Each node encountered in this traversal is assigned to a K-LUT. To decrease the number of K-LUTs on any path, the algorithm attempts to pack as many gates as possible into the LUT, such that the depth of logic that is left for mapping is minimized.

This approach does not achieve a depth optimal mapping. There is a case for which this algorithm fails to produce depth optimal solution. This case is shown in Figure 2-1. Consider a mapping into a 3-LUT. It is clear that the group of logic gates near the output have more than 3 inputs so they cannot be put in the same 3-LUT. However, further analysis of the circuit reveals that

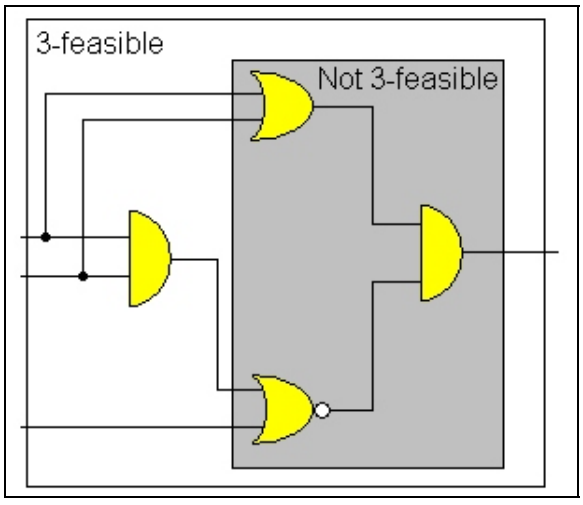

Figure 2-1: Sub-optimal solution produced with the basic approach [4]

the entire circuit only has 3 inputs, therefore only one 3-LUT is required to implement it. An algorithm that covers this case was introduced by Cong and Ding. Their Flowmap [4] algorithm, which guarantees a depth optimal mapping in polymonial time, is presented in the following subsection.

2.2.3 Depth Optimal Solution

The aforementioned problem is difficult to overcome using the basic approach, because during the mapping stage the logic gates are processed from primary output to primary input to reduce the LUT depth. To address that problem a mechanism known as the network flow [27] is used.

Consider a directed graph **G**, where a node *s* has only outgoing edges and node *t* only incoming edges. The *network flow* of such a graph determines the number of paths between *s* and *t*. If each edge is allowed to be used only once then the network flow will determine the maximum number of paths between nodes *s* and *t* that have no common edges. Each edge is assigned a capacity, which represents the maximum flow per unit time that the edge can support. In the process the network flow locates the place in the network where a minimum number of edges can be removed to disconnect all paths from *s* to *t*.

The Flowmap algorithm, developed by Cong and Ding [4], uses the network flow to overcome the problem illustrated in Figure 2-1. Similar to the basic approach, the Flowmap algorithm processes the logic network in two phases: labeling and mapping. The difference is that the network flow computation is utilized to determine the LUT depth that should be assigned to each gate in the logic network.

The input to the Flowmap algorithm is a K-feasible DAG **G**, which represents the logic network. Each node in **G** corresponds to a logic gate or a primary input, while each edge of **G** is a connection between the output of one logic gate and the input of another. To determine the label for each gate the graph **G** undergoes the following transformation to be suitable for use with the network flow algorithm:

- a node *s* driving all primary inputs is created
- a node *t* is the node to be labeled. All nodes with label $\max\{label(v) : v \in inputs(t)\}$ and

node *t* are put together to form node *t'*

- any node or edge of the graph that is not on a directed path from *s* to *t* is ignored
- each node *v*, except *s* and *t'*, is split in two. The resulting nodes v_1 and v_2 are such that:
  - all input edges are assigned to v_1
  - all output edges are assigned to v_2
  - a directed edge from v_1 to v_2 is created. This edge is called the *bridging edge* [4].

In this new graph, **G'**, consider the bridging edges as outputs of logic gates. To find a mapping that covers the node *t'* the graph **G'** must be divided, or cut, into two parts. One part contains the node *t'* and possibly a few other nodes, while the other part contains the rest of the graph **G'**. Cutting one of the bridging edges means that the gate corresponding to the source of this edge will drive a LUT that implements node *t'*. Finding a mapping that fits *t'* into a K-LUT means separating the graph into two parts by cutting only the bridging edges. One part of the graph must contain node *s* and the other node *t'*. A successful mapping results in locating a cut of size K or less in the graph **G'** by using the network flow algorithm. If the cut is not found then a new LUT must be created to implement the gate represented by node *t*.

The logic circuit implementation obtained by FlowMap is optimal under the assumption that the delay between any pair of connected LUTs is the same.

2.2.4 Improving Flowmap

The assumption of the FlowMap algorithm that the delay on each edge is identical is not always true. In fact, after placement and routing each connection in the logic circuit will have a delay determined by the length of the connection and the number of connections driven by the same source. To improve the speed of the logic circuit the delay of each connection between K-LUTs must be considered. A modified version of the FlowMap algorithm that considers variable delay between K-LUTs, called Edge-Map [22], was introduced by Yang and Wong.

There are two important differences between Edge-Map and FlowMap. First, the Edge-Map algorithm assigns a delay value to each edge. Second, the algorithm that divides a node *v* into v_1 and v_2, does not assign all output edges to v_2. Instead, only edges that have a delay less than or equal to some delay value *d* are assigned to v_2. Other edges are assigned to v_1. This change causes edges with

longer delay to be realized within a LUT, reducing their delay to zero, leaving only edges with smaller delay to connect the LUTs. The experimental results [22] have shown a 27.8% reduction in logic circuit delay, based on post-placement estimates, when compared to FlowMap, with only a small increase in logic circuit size.

2.2.5 Modifying Initial Representation

The result of the FlowMap algorithm relies heavily on the initial circuit representation. It is important to note that the FlowMap algorithm, as well as the Edge-Map algorithm, try to optimize the mapping of a logic circuit without resorting to exploring other logic gate networks that implement the same functionality. Thus, the mapping may be optimal for a specific gate network, but not necessarily optimal over all possible gate networks that can implement the same logic circuit.

One method of addressing this problem is presented by Chang *et. al.* [12]. They propose grouping logic gates that implement a function of 9-10 inputs and representing the logic function as a Binary Decision Diagram (BDD). Then a set of mapping solutions is explored, by using the Roth-Karp decomposition [23], to minimize the depth of LUTs needed to implement the logic function. The work of Chang *et. al.* [12] showed that allowing some flexibility in the initial representation of the circuit decreases the depth of LUTs needed to implement logic functions by an average of 8 per cent on a set of 20 logic circuits.

The approach presented by Lehman *et. al.* [24] explores a wider variety of mapping solutions. Instead of grouping logic and generating BDDs for each group, this work proposed performing logic transformations on the entire logic circuit. Each transformation applied to the graph produces alternate solutions that will be considered by the technology mapping algorithm. The application of their approach on a set of 14 benchmark logic circuit showed a 34% reduction in logic circuit delay when compared to SIS-1.2 [28].

2.2.6 Mapping Logic Functions into Complex Logic Structures

The mapping of logic functions into programmable logic blocks is a key step in technology mapping. In the previous subsections, the mapping algorithms considered only a mapping that can

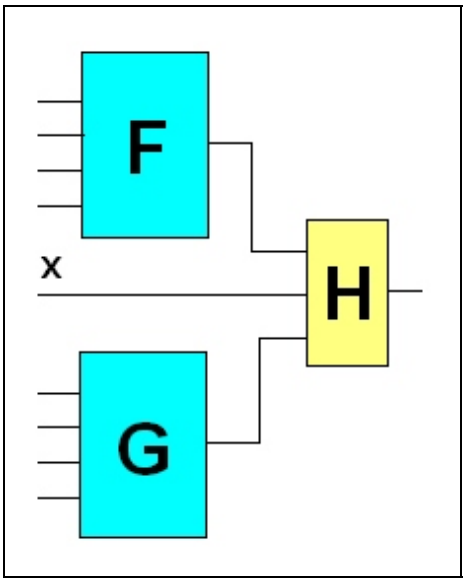

Figure 2-2: Generic Logic Block Structure for Virtex-E and XC4000 series FPGAs

be realized in a single K-LUT. Most commercial FPGAs, such as the Xilinx Virtex-E [7] and the Xilinx XC4000 devices [19], contain more complex logic structures. The generic logic structure for both of these devices is shown in Figure 2-2, where F, G and H are LUTs.

Boolean matching approaches for logic structures in Figure 2-2 have been presented by Cong and Hwang [26]. This dissertation focuses on the mapping approach for the Virtex-E type logic block, in which the LUT H implements a multiplexor controlled by signal **x**.

To map a logic function *f* into the Virtex-E type logic structure a logic function decomposition must be found such that $f(\mathbf{X}) = x \cdot \mathbf{F} + \bar{x} \cdot \mathbf{G}$, where $x \in \mathbf{X}$. The logic function *f* can be mapped into the Virtex-E type logic structure when Shannon's expansion [9] $f(\mathbf{X}) = \bar{x} \cdot f_{\bar{x}} + x \cdot f_x$ can be found for some $x \in \mathbf{X}$ and each of the cofactors ($f_{\bar{x}}$ and f_x) can be implemented in LUT G and F respectively.

In this work the approach of Cong and Hwang [26] is extended to cover more complex logic structures. The logic structures and the algorithms for mapping logic functions into them are presented in Chapter 3.

2.3 Physical Synthesis

The previous Section described technology mapping algorithms that focus on improving the delay in the logic circuit as measured by the number of logic components on all paths in the logic circuit. Some of the algorithms went one step further by attempting to model interconnect delay. However, all of these works maintained a strict separation between the logic synthesis and the placement and routing (P&R) stages. Thus, a logic synthesis optimization that appeared to be beneficial from the logic synthesis point of view, may in fact be detrimental to the speed of the logic circuit, once placement and routing is performed. Improving the interaction between the logic synthesis and P&R to improve the synthesis of the logic circuit is the domain of physical synthesis.

Physical synthesis algorithms use an iterative approach to converge on a good design implementation. Each iteration provides additional information about the final implementation of the logic circuit that affects the choice of the logic synthesis optimizations. In this process it is important to have a delay model so that the delay between logic components can be computed accurately. Furthermore, methods of leveraging logic synthesis optimization and improving the interaction between logic synthesis and placement and routing are needed.

We review works that study the means by which an accurate delay model for the connection between logic components can be obtained, and then give examples of physical synthesis algorithms. These algorithms present various methods of improving the interaction between the logic synthesis and the placement and routing (P&R) stages to improve the speed of the logic circuit.

2.3.1 Estimating Net Delay to Improve Logic Synthesis

The physical synthesis approach performs iterations of logic synthesis and placement to produce an implementation of a logic circuit. If a placement has been created after a first pass of logic synthesis, an estimate of the delay of nets that connect logic components in the logic circuit can be obtained. The delay of a net can be estimated using various different methods. The simplest method, given placement, is to calculate the distance between the source and the destination of a connection and use it as the delay estimator. On an FPGA this procedure is quite simple as the distance between logic components can be measured by the number of rows and columns that separate them, assuming that wires of fixed length are used. This method can be modified to take

into account wires of various lengths to better estimate the delay of a logic connection.

A recent work by Lin *et. al.* [2] uses the location of the source and the target LUTs, in which a pair of gates reside, to estimate the delay between a pair of connected gates. Their mapping algorithm uses this information to decide how to modify the assignment of logic gates to the LUTs, such that the performance of the circuit is improved. The algorithm moves a gate from one LUT to another, sometimes necessitating the creation of an additional LUT due to LUT's input size constraint. Every time a LUT is modified it is assigned a preferred location to guide the subsequent placement iteration.

The work of Lu *et. al.* [10] suggests taking the type of gate and the size of the gate fanout into account when estimating the delay of a net. Their algorithm utilizes these models in an attempt to decrease the estimated critical path delay through logic synthesis and placement perturbations. The results show a 17% reduction in post-placement logic circuit delay when these models are used to estimate net delay in comparison to SIS-1.2 [28].

Each of the aforementioned works does have some inaccuracies in the delay model. In the present work, as discussed in Chapter 3, commercial FPGA devices and tools are used to estimate the net delay. The Xilinx FPGA Editor has sufficient knowledge of Xilinx Virtex-E devices to accurately model every delay element in the FPGA.

2.3.2 Improving Interaction between the Logic Synthesis and the P&R

In the physical synthesis CAD flow the logic synthesis stage precedes the placement and routing (P&R) stage. This is because the P&R stage requires logic components to be created before they can be placed and routed. However, to perform good logic synthesis optimizations a logic synthesis tool needs accurate net delay estimates, which are only available after placement and routing. Similarly, the P&R stage must be aware of the intention of a logic optimization so that it does not undermine the efforts of the logic synthesizer. Therefore, a close interaction between the logic synthesis and the placement and routing (P&R) stages is essential for improving the speed of logic circuits.

There are currently three types of approaches that address this issue. The first approach applies synthesis and placement in an iterative process. The second method is for the synthesizer

to specify to the placer where to place the synthesized logic components. This enables the placer to better understand the decisions made by the synthesizer, and possibly accommodate them. The third option is to permit the placer to evaluate several alternate logic mappings so that their placement can be considered.

The example of the iterative approach is given by Lin *et. al.* [2]. During each iteration the mapping algorithm takes some of the gates from one LUT and places them in another, basing its decisions on net delays between the gates. The new mapping is then placed again, using last placement as a guide. The iterative application of synthesis and placement that strives to minimize the number of logic changes in each consecutive iteration is shown to yield 12.3% speed improvement, based on the post-routing delay using VPR [29] on a set of 10 logic circuits.

The work of Singh and Brown [3] proposes that the placer should be provided with an incentive to situate logic components in a specific location on the device. Their approach starts with a regular CAD flow to obtain a synthesized and placed logic circuit implementation. Then layout-driven optimization techniques are used to reduce the delay on the critical paths. Each new logic element, which is created in the process, is assigned a location that the placer aims to obtain for it while minimizing the disruption to the entire logic circuit. This approach conveys the context in which the synthesizer made its decision, thus allowing the placer to respond to it accordingly. The results, based on a set of 10 logic circuits, presented in [3] show that performing logic re-synthesis on small subsets of logic and assigning "preferred locations" [3] to newly synthesized components improves convergence.

The previous two examples maintained the separation between the logic synthesis and the placement stages. The approach proposed by Lou *et. al.* [1] breaks this boundary by having the synthesis stage provide several mapping solutions for a subcircuit it considers to be good. The placer then chooses the mapping solution to improve the speed of the logic circuit, since speed is easier to estimate during placement and routing stages. This approach was tested on a set of 13 logic circuits, using a 0.35µm ASIC library. The results have shown an average of 29% reduction in delay and 5% increase in area.

The method applied in this thesis borrows from each of the three approaches. The iterative process is used to let the user improve the circuit in an incremental fashion. The user is provided

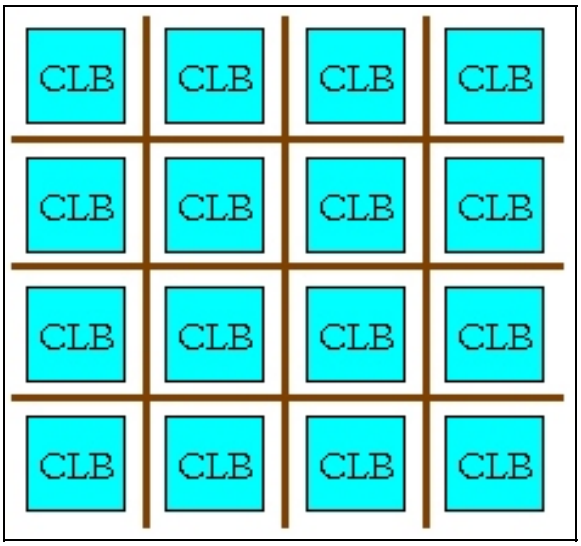

Figure 2-3: An Island-Style FPGA

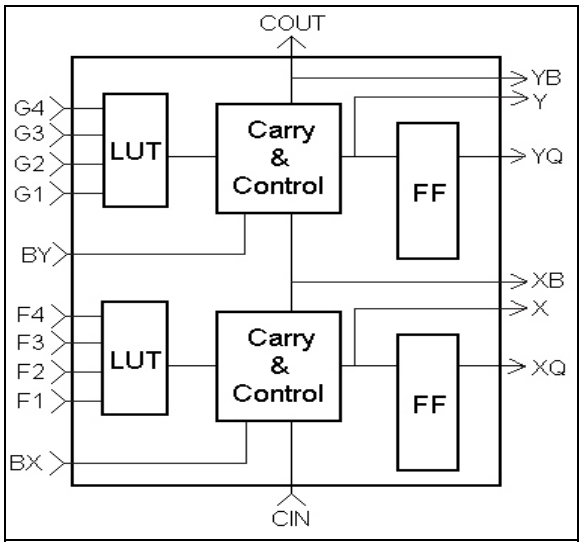

Figure 2-4: Simplified view of a Virtex-E slice

with detailed timing analysis and visual cues, such as rubber bands and visual representation of circuit components on the device, to suggest the place the component should be situated. Finally, the user can explore more than one logic optimization alternative while performing placement.

2.4 Xilinx Virtex-E device and Xilinx CAD tools

The algorithms described in previous sections focused on FPGA architectures that consist of K-LUTs and flip-flops. The focus of this thesis is to examine the logic synthesis transformations on a commercial FPGA. This Section describes logic structures specific to the Xilinx Virtex-E architecture, which are central to the logic synthesis transformations proposed in this work. The set of Xilinx tools, which are used to implement the logic synthesis transformations on a commercial device and to provide omniscience, are described as well.

2.4.1 Xilinx Virtex-E Device Family

The FPGA devices in the Xilinx Virtex-E family [7] are island-style FPGAs, as shown in Figure 2-3. The following subsections describe the Configurable Logic Block (CLB) of the Virtex-E. The routing architecture is also described, as some of the optimization strategies hinge on its features.

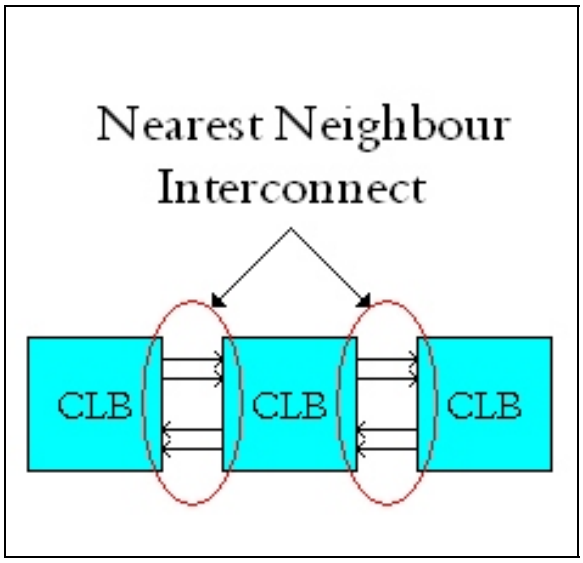

Figure 2-5: The Nearest Neighbour Interconnect on the Virtex-E device

The Configurable Logic Block

The Xilinx Virtex-E CLB is the building block of the Virtex-E FPGA. The CLB is divided into two slices, which contain two flip-flops, two LUTs, as well as carry and control logic, as shown in Figure 2-4. The flip-flops in each slice must have common control signals: Clock, Set, Clear and Enable. The Set and Clear control signals are either synchronous or asynchronous. Each LUT in a slice can implement a 4-input logic function. More complex functions can be implemented by using LUTs in conjunction with the carry and control logic. Depending on the configuration of the carry and control logic, different functions are implemented by slices or the CLBs. The detailed diagram of the Xilinx Virtex-E slice is shown in Appendix A.

The Routing Architecture

The routing architecture consists of four types of wires, which provide connections between CLBs. Single length wires facilitate the connection between neighbouring CLBs. Connections between more distant CLBs are realized using either length six wires, connecting CLBs six rows/columns apart, or long wires that span the width and height of the device. The crucial type of wire that is used by the optimization strategies is the Nearest Neighbour (NN) Interconnect [8].

The NN Interconnect is a dedicated set of wires that connects horizontally adjacent CLBs.

This interconnect has a very low delay and can significantly improve performance of the circuit when used properly. This resource is limited to two pairs of unidirectional wires between each pair of CLBs, as shown in Figure 2-5.

2.4.2 Xilinx FPGA Editor

The Xilinx FPGA Editor is a tool that allows the user to modify a placed and routed logic circuit. It visually represents all the components of the chip and provides a set of functions that enable design modification. In addition to logic circuit modification the FPGA Editor performs timing analysis.

To modify a logic circuit in FPGA Editor, the user must specify a sequence of very specific changes to the logic circuit. For example, to move a LUT from location (x_1, y_1) to location (x_2, y_2), the following set of commands must be entered:

1) Select slice at (x_1, y_1)
2) Remember the logic function implemented by the LUT and the input signals driving the LUT
3) Remove inputs from the LUT
4) Remove the output of the LUT, but remember which components it was driving. If the LUT provided a data signal to the local flip-flop, modify the flip-flop to receive external to the slice data input.
5) Select slice at (x_2, y_2)
6) Input LUT equation into the slice
7) Connect input and output signals to the LUT. If one of the outputs drives the local flip-flop, then remove the connection from the output net and reconfigure the slice to implement this connection internally

It is obvious that making a significant number of changes in this manner is cumbersome. To give the user the ability to make such modification by providing only an abstract request, such as move LUT from (x_1, y_1) to (x_2, y_2), Chow and Rose [6] developed a CAD tool called EVE that automates most of these abstract tasks.

2.5 EVE - An Omniscient Placer and Packer

A prior work by Chow and Rose [6] introduced a manual editor, which targeted the Virtex-E architecture. The **EV**ent Horizon **E**ditor (EVE) [6] enabled the user to modify the placement and packing incrementally. After every modification EVE conducted timing analysis to inform the user about the new circuit performance, thereby providing omniscience.

EVE relies on Xilinx FPGA Editor to extract the delay information from the design, since Xilinx FPGA implementation tools encompass more detailed information about Xilinx devices. The timing information in this environment is therefore accurate, so the performance gains obtained using the manual editor are realistic. Furthermore, accurate timing information is necessary to provide omniscience.

To modify a logic circuit with EVE the user must first synthesize, place and route a logic circuit using commercial tools. The implemented logic circuit is then read by EVE and represented visually as a set of connected logic components on an FPGA grid. The FPGA grid is represented as a two dimensional array of rectangular cells, where each cell of the grid corresponds to a single Virtex-E CLB.

To improve the logic circuit the user selects logic components and moves them into a free location on the FPGA. The user can select a single component, or a group of components, to move from one place to another. After each placement and packing modification EVE makes the specified modification in the FPGA Editor, which holds the exact copy of the logic circuit. After the FPGA Editor implements the requested changes it performs timing analysis. The results of timing analysis are transmitted to EVE so that the user is informed about the new speed of the logic circuit.

The EVE software was applied to a set of 8 benchmark logic circuits to modify their placement and packing. On this set of benchmarks EVE achieved a 12.7% improvement in logic circuit speed.

In addition to the placement and packing modifications, EVE also assists the user with pipelining of the logic circuit. EVE allows the user to specify where a flip-flop should be inserted into the logic circuit and then determines if the specified location is valid. If the location for the new flip-flop is valid, EVE inserts a flip-flop at the user specified location and at any other location that requires a flip-flop to be inserted to preserve logic circuit functionality. The pipelining feature of

EVE was tested [25] on two logic circuits, improving their operating frequency by 3.53% and 42.24%.

2.6 Summary

This chapter presented prior work concerning physical synthesis. There are two key points suggested by these works. The first point is that a close interaction between the logic synthesis and placement and routing stages is necessary to improve the speed of the logic circuit. The second point is that accurate timing information is crucial to make better logic synthesis optimizations.

In this thesis omniscience is the means by which a close interaction between logic synthesis and placement and routing is achieved. The timing information that is used to provide omniscience comes from commercial tools, which have accurate timing information for the device they target. The work of Chow and Rose [6] is the basis for the manual editor developed in this thesis. The manual editor is enhanced to provide the user with the ability to perform various logic synthesis transformations in the context of omniscience.

3 Augur Context and Logic Synthesis Transformations

3.1 Introduction

The goal of this research is to improve the speed of the implementation of logic circuits on FPGAs by providing the user with the most correct, and complete, feedback possible on the consequences of manually specified logic transformations. If logic synthesis transformations were made with the full knowledge of their final effect after routing then the final result would likely be better.

The methodology adopted in this work is to first implement a logic circuit using commercial logic synthesis, placement and routing tools and then let the user improve the speed of the logic circuit through manually specified logic transformations. To make informed decisions the user is provided with information about how each microscopic logic circuit modification affects the speed of the circuit. This information includes the maximum circuit operating frequency, delay distribution of all paths in the circuit, the function implemented by each logic component and the placement of logic components. We term the ability of a CAD tool to provide the above information after every microscopic modification as omniscience.

In this research we develop a manual editor, called Augur, which uses logic synthesis optimizations in the context of omniscience to improve the speed of a digital circuit. The focus of Augur is on microscopic logic transformations, such that the user can observe the effect of a single logic transformation on the speed of the circuit. After each manual modification resulting in the change in the netlist and placement of the logic circuit, the editor performs routing and timing

analysis and provides the user with instant feedback about the new circuit speed, our so-called omniscience.

3.2 Implementing Logic Transformations with Augur

Before the logic synthesis transformations that are provided by Augur to the user are described, the context in which they are used is presented. First, a brief description of the manual editor is provided, focusing on how the user performs circuit modifications. Then the logic transformations available in Augur are described in detail.

Augur presents the user with a picture of the logic circuit as shown in Figure 3-1. There are three types of components shown: LUTs, carry logic and flip-flops. The LUTs are symbolized by cyan rectangles with an icon, depicting a multiplexor, in the middle. The carry logic is represented

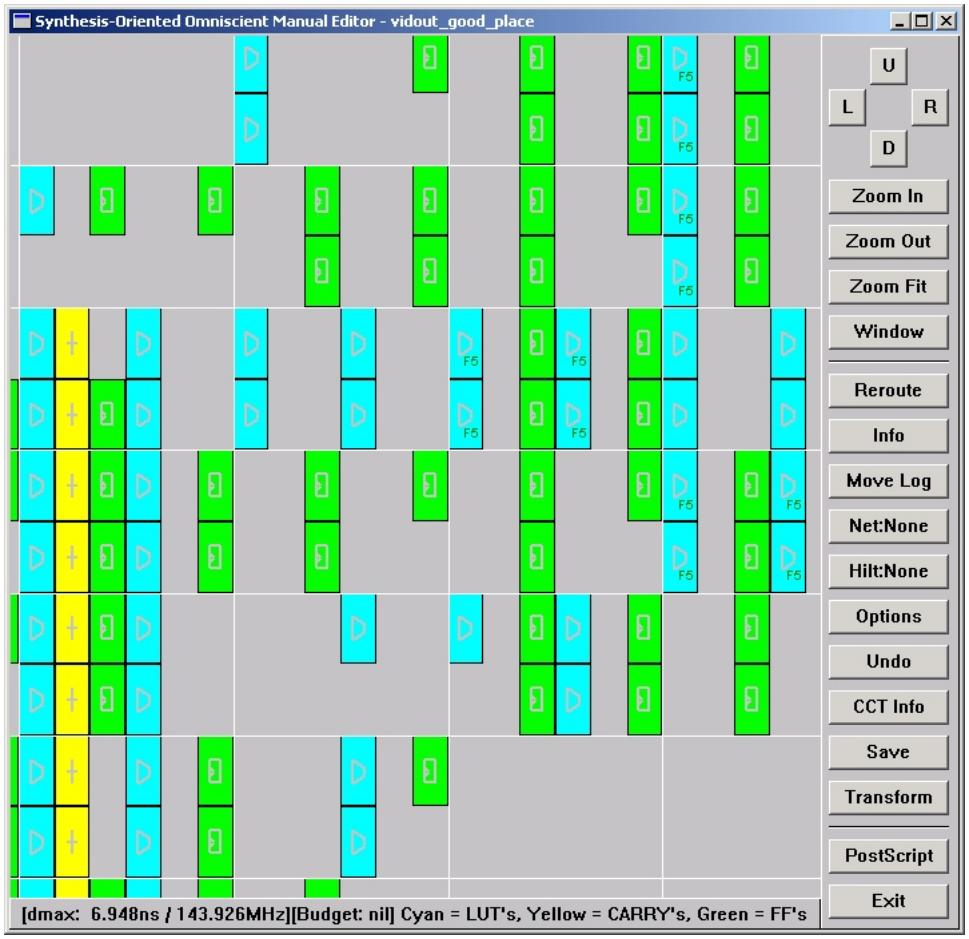

Figure 3-1: Logic circuit representation in Augur

in yellow with a plus sign (+) icon in the middle. The flip-flop is green with a flip-flop icon in the middle. The user can select one or more of these icons to make logic circuit modification. The two types of logic circuit modifications available are placement/packing and logic synthesis modifications.

To make a placement modification the user selects a set of components that can then be moved, retaining their relative positions, to a new location. Augur does not allow components to be moved into illegal positions, such as moving a LUT into a carry chain slot. A successful logic circuit modification causes Augur to perform the routing and timing analysis and return to the user with the new logic circuit maximum operating frequency.

To perform a logic synthesis transformation the user selects a set of logic components and attempts to employ one of a set of available logic transformations, by pressing the ***Transform***

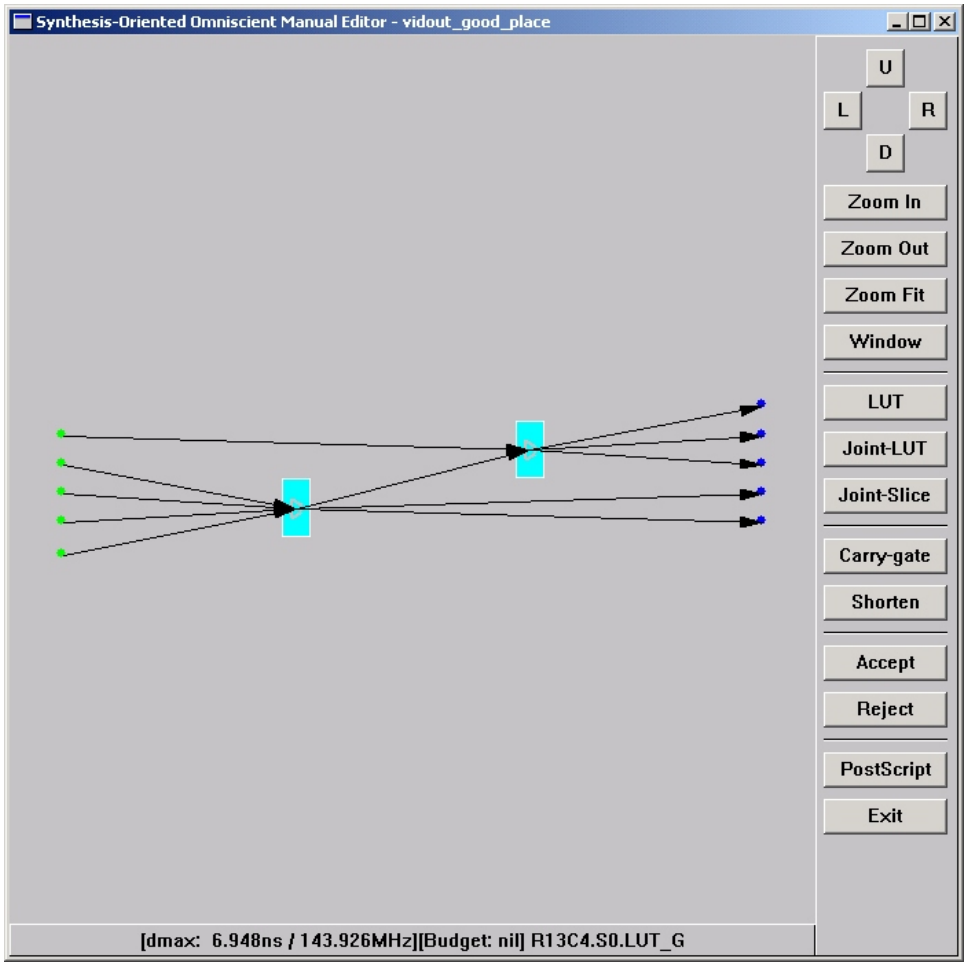

Figure 3-2: The SubCircuit view

button. This causes Augur to present the user with an alternate representation of just the selected logic, as shown in Figure 3-2.

This view is termed the *SubCircuit View*, and it shows the logic subcircuit selected by the user in isolation from the rest of the design. The inputs to the logic subcircuit are shown on the left side of the screen, the subcircuit itself in the middle, while the outputs of the subcircuit are located on the right side of the screen. The subcircuit itself is organized on the screen such that it presents a topological view of the subcircuit, looking from left to right. The user can select the components in this view and perform logic synthesis transformations on them, causing the creation of a different netlist of components (still with LUT, carry and flip-flop cells).

At any point during the resynthesis process, the user may attempt to place currently synthesized subcircuit - this is an important issue as once the netlist has been changed by a logic

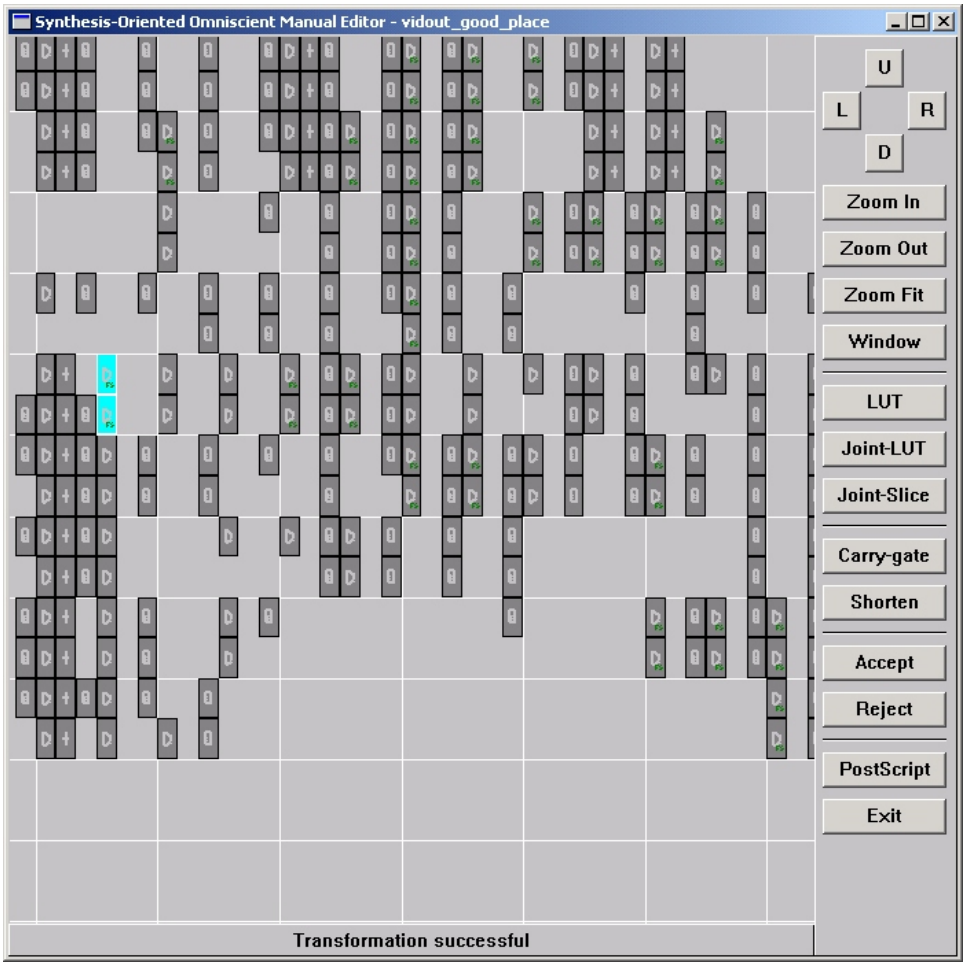

Figure 3-3: The Placement view used during resynthesis

synthesis transformation the old placement is no longer valid. By selecting a component in the view in Figure 3-2 and pressing the *Tab* key, the selected component is shown on the FPGA grid, allowing the user to select a suitable placement for it. The placement view is shown in Figure 3-3. Once the user places all logic components of the subcircuit, the user can press the ***Accept*** button for the changes to take effect. This will cause Augur to perform the routing and timing analysis and provide the user with the new maximum operating frequency.

Augur provides a total of five types of logic transformations: remapping, logic duplication, merging, carry chain shortening, and register control signal extraction, as described in the following Sections.

3.3 Remapping

The remapping operation attempts to transform a selected set of logic components into a functionally equivalent set that fits into a slice or a CLB of the Virtex-E FPGA. The following subsections describe two remapping algorithms: Carry Chain Mapping and Multiplexer mapping.

3.3.1 Carry Chain Remapping

The Virtex-E slice contains logic capable of directly implementing an arithmetic carry chain [7]. This same logic can be configured to implement an AND or an OR gate, such that non-arithmetic functions can utilize the carry chain to improve the speed of the logic circuit.

The carry chain logic in the Virtex-E slice is a set of multiplexors (coloured in yellow in Figure 3-4) that connect the two LUTs together, as shown in Figure 3-4. This structure, referred to as the carry chain structure, can be manipulated to implement an AND or an OR by assigning constant values to inputs CY0F, CY0G and CIN. To convert the carry chain into an AND gate the inputs CY0F and CY0G are set to 0, while CIN is set to 1. This transformation implements a function F•G, where F and G are the functions of LUTs F and G in the slice. An OR gate is implemented when the inputs CY0F and CY0G are set to 1, while CIN is set to 0. This implements a function F+G.

The advantage of using the carry chain structure is that a pair of serially connected LUTs can sometimes be converted into a parallel pair of LUTs connected through a fast AND or OR gate

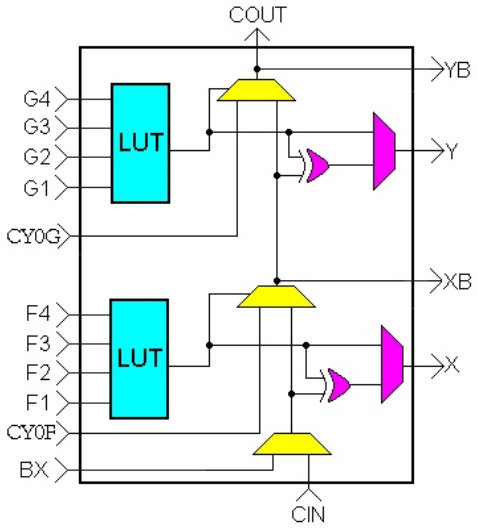

Figure 3-4: Virtex-E Carry Chain Structure

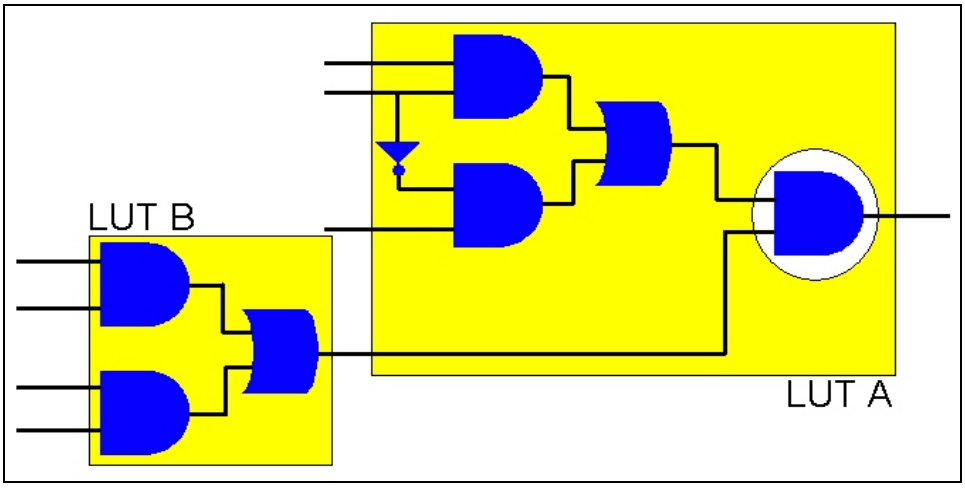

Figure 3-5: Sample circuit to be mapped into a carry chain configuration

implemented in the carry logic. Since the carry multiplexor is very fast, the transformation leads to an overall reduction in delay if the original pair of LUTs is on the critical path.

Consider the example pair of LUTs (A and B) illustrated in Figure 3-5, which shows the logic function of each LUT as a schematic inside each LUT box. The highlighted AND gate at the right of LUT A can be implemented in the Carry Chain of Figure 3-4 because one of its inputs comes directly from LUT B. Figure 3-6 illustrates the isolation of this AND gate and Figure 3-7 shows the final implementation of the function using the carry chain to implement an AND gate.

To determine if a pair of serially connected LUTs can be mapped into a slice, the AND gate

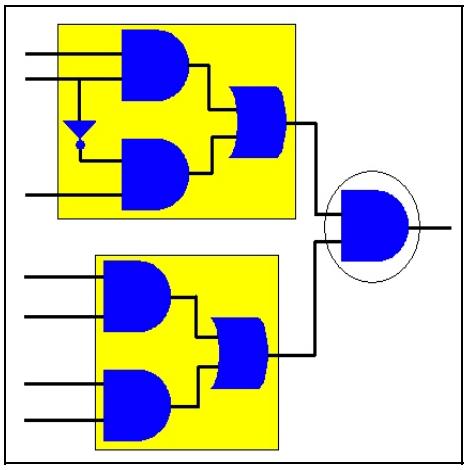

Figure 3-6: AND gate extracted from the forward LUT

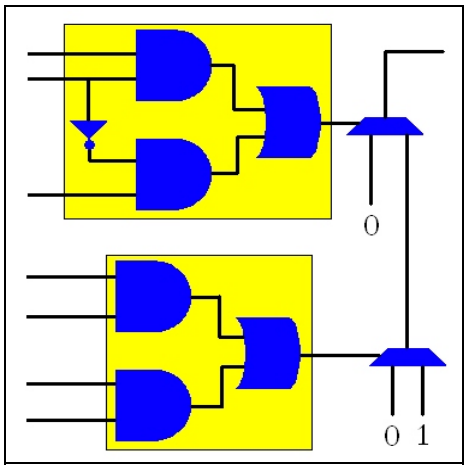

Figure 3-7: Final implementation in carry chain configuration

(or OR gate) must be found, as shown in Figure 3-6. The search process is performed using Shannon's expansion of LUT A's function with respect to LUT B. Let A be the function of LUT A, and B be the signal generated by LUT B. The inputs to A are x_1, x_2, x_3 and B. From Shannon's theorem [9] the following equation is obtained:

$$A = A(x_1, x_2, x_3, 0) \bullet \overline{B} + A(x_1, x_2, x_3, 1) \bullet B \quad (1)$$

If the function $A(x_1, x_2, x_3, 1)$ evaluates to a constant 0, then the equation (1) reduces to:

$$A = A(x_1, x_2, x_3, 0) \bullet \overline{B} \quad (2)$$

which produces the desired AND gate that can be mapped into the carry chain of the slice.

An OR gate can be detected by following a similar procedure. When the function $A(x_1, x_2, x_3, 1)$ in equation (1) evaluates to 1 then the following simplified equation is obtained:

$$A = A(x_1, x_2, x_3, 0) + B \quad (3)$$

which produces the OR gate to be implemented in the carry chain of the slice.

The algorithm in Figure 3-8 determines if a pair of serially connected LUTs can be transformed in this manner. Its input is a user-selected pair of serially connected LUTs, and the output is either the mapping of those LUTs into a slice, or a declaration that the mapping is not possible.

Input: A pair of series-connected LUTs A and B, with LUT B driving LUT A.
Output: On success, a mapped slice with the same functionality of A & B, but mapped into carry logic

Let A be the function of LUT A, and B be the function of LUT B.
f_0 = Shannon expansion of A with respect to B=0
f_1 = Shannon expansion of A with respect to B=1

if either f_0 or f_1 is a constant (i.e. is 0 or 1) then
    it is possible to map A and B into a slice:
    let h be the cofactor (f_0 or f_1) that is not a constant
    implement h in the top LUT of the slice, inverting if necessary
        as per section 2.1
    implement B in the bottom LUT
    create a LUT that inverts the output of LUT B if B uses output pin X
    return function_implemented_in_slice
else
    return fail.

Figure 3-8: Algorithm for mapping an AND or an OR gate into the carry chain

In addition, it is possible to map a pair of serially connected LUTs in which the LUT B has fanout greater than 1. Notice that the carry chain configuration in Figure 3-4 allows the bottom LUT to produce an output signal through pin X. To properly generate this secondary output it may be necessary to add a single LUT that inverts the output of pin X, since the function of LUT B may need to be inverted to implement an AND gate (or an OR gate) in the carry chain.

3.3.2 Multiplexor Mapping

The Virtex-E slice also contains fast multiplexors that combine the output of two or more LUTs to implement complex logic functions. In this Section a basic approach for mapping single output logic functions into multiplexor-based logic structures is presented, followed by the enhanced algorithm that maps multi-output logic functions into multiplexor-based logic structures.

The Basic Approach

The basic approach to multiplexor mapping allows a single output logic function to be implemented in a multiplexor based structure using Shannon's decomposition. The approach for

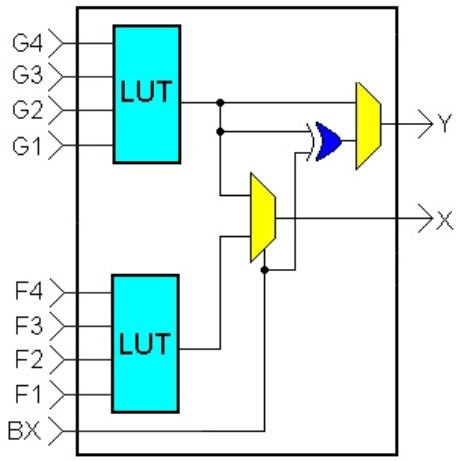

Figure 3-9: The Joint-LUT structure

Input: A logic function $f(x_1,...,x_n)$
Output: A mapped Virtex-E Slice that implements f in a Joint-LUT structure

for all variables x_i in f do
    f_0 = cofactor of Shannon's Expansion of f with respect to x_i=0
    f_1 = cofactor of Shannon's Expansion of f with respect to x_i=1
    if (f_0 fits in a LUT and f_1 fits in a LUT) then
        implement f_0 in top LUT of the Joint-LUT
        implement f_1 in bottom LUT of the Joint-LUT
        exit
end do

Figure 3-10: Basic Joint-LUT mapping algorithm

mapping logic function into a structure with a single multiplexor that joins the output of two LUTs has been presented before by Cong and Hwang [26]. This section presents the approach which allows a mapping into such a structure and extends it to a more complex logic structure that contains three multiplexors.

The first structure that makes use of the fast multiplexors is the Joint-LUT structure, shown in Figure 3-9. This structure can implement a logic function containing up to 9 inputs. To implement a 9-input logic function, the function has to be decomposed to determine if the mapping into the Joint-LUT is possible. The decomposition process determines if it is possible to break up the logic function into a multiplexer driven by two 4-input LUTs. We use Shannon's decomposition theorem [9] to perform the decomposition of the logic function. The basic idea is to perform Shannon's

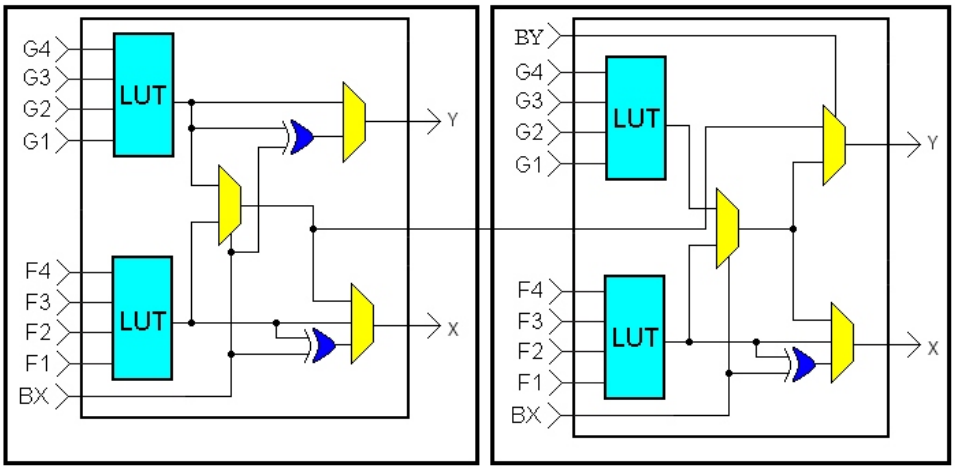

Figure 3-11: The Joint-Slice structure

expansion with respect to every variable of the function [26]. A valid mapping of the 9-input function into the Joint-LUT structure is found when both cofactors of the function in the Shannon's expansion are functions with at most 4 inputs. The algorithm is summarized in Figure 3-10.

The advantage of using the Joint-LUT structure is that it exploits parallelism, which can reduce the delay of signals passing through it. For example, when a pair of serially connected LUTs is mapped into this structure, the function of both LUTs in the Joint-LUT structure is evaluated simultaneously. Thus, the delay through the Joint-LUT structure is the delay of one LUT plus the delay of the dedicated multiplexer. A pair of serially connected LUTs has a longer delay, which is equal to the delay through the two LUTs plus the routing delay between the LUTs.

It is possible to implement even more complex functions by merging two Joint-LUT structures with a multiplexer available in the Virtex-E CLB, as illustrated in Figure 3-11. This will be called the *Joint-Slice* structure. The Joint-Slice can implement some logic functions with up to 19 total inputs. The mapping algorithm is very similar to the Joint-LUT mapping algorithm above, except that after Shannon's decomposition of the logic function f the algorithm looks for a successful mapping of each cofactor, f_0 and f_1, into a Joint-LUT structure instead of a LUT.

Multiplexor Mapping with Multiple Outputs

The basic method of mapping logic functions into multiplexor-based logic structures allows logic functions to be implemented in a Joint-LUT or a Joint-Slice structure. The work of Cong and

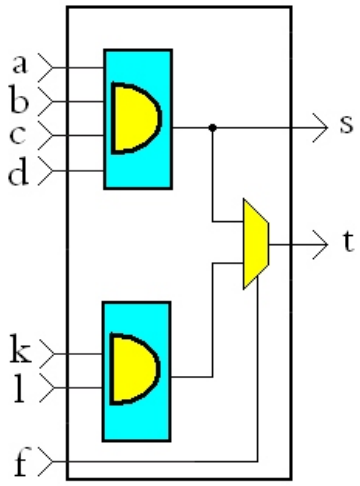

Figure 3-12: Example of mapping a 2 output logic function into the Joint-LUT structure

Hwang covered the case for the Joint-LUT, which we extended to the Joint-Slice structure. A more important contribution of our multiplexor mapping algorithm is its ability to implement logic functions with multiple outputs in the Joint-LUT and the Joint-Slice structures, as shown in Figure 3-12. The following discusses how the basic algorithm is modified to provide this ability.

Figure 3-9 shows that the Joint-LUT structure pin X produces the multiplexor output for the structure. In addition to this output the structure has a *secondary* output that is generated by pin Y. Notice that pin Y corresponds to the function of the top LUT in this structure. Because the logic function implemented produced through the multiplexor, and therefore output through pin X, depends on the function of the top LUT, the logic function produced by pin X will be termed the *primary* output function.

To map a logic function with two outputs into the Joint-LUT structure we must first determine the primary and the secondary output function. This can be done by checking which function is a subfunction of the other. Then the algorithm proceeds to decompose the primary output function using Shannon's decomposition, but a successful mapping is considered to be found if and only if one of the cofactors of the expansion implements the secondary output function. The algorithm is summarized in Figure 3-13.

Note that this algorithm will be unable to map structures like those illustrated in Figure 3-14, even though this is a correct mapping. This is because during Shannon's expansion one variable is

Input: a function $f(x_1,...,x_n)$ with up to two outputs
Output: a Joint-LUT structure that implements $f(x_1,...,x_n)$

Let t be the primary output function
Let s be the other function
For all variables x_i in function t
    t_{xi} = Shannon's Expansion of t with respect $x_i = 1$
    $t_{\bar{x}i}$ = Shannon's Expansion of t with respect $x_i = 0$
    if (t_{xi} fits in a single LUT) and ($t_{\bar{x}i}$ fits in a single LUT) then
        if ($|F| = 2$ and ($t_{xi} = s$ or $t_{\bar{x}i} = s$)) or ($|F| = 1$) then
            map t_{xi} and $t_{\bar{x}i}$ into LUTs and apply signal x_i to the selector input of the multiplexor
end for

Figure 3-13: Mapping a multi-output logic function into the Joint-LUT structure

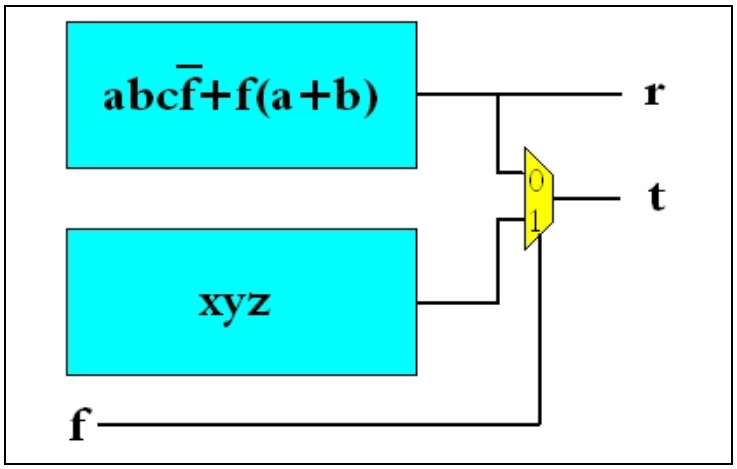

Figure 3-14: Mapping solution not found by the algorithm

removed from the equation of each cofactor and it is assigned to drive the multiplexer selector input. However, the case shown in Figure 3-14 requires the variable *f* to drive both the selector input of the multiplexer and the top LUT.

3.4 Duplication

The second transformation available in Augur is duplication, which creates a copy of a selected component. By duplicating a component on the critical path, one can increase the freedom

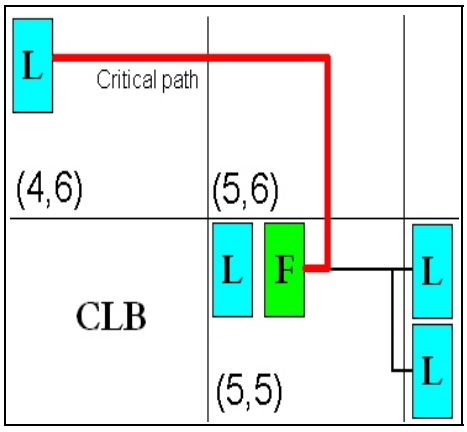

Figure 3-15: Circuit before duplication

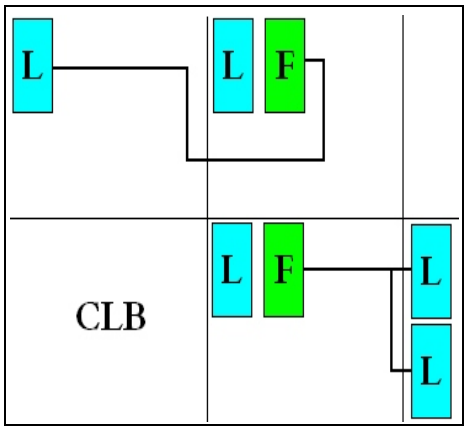

Figure 3-16: Circuit after duplication

to position and route critical connections [4][13]. Logic duplication is a particularly useful feature that enables the use of fast Nearest-Neighbour (NN) interconnect for critical connections.

For example, consider the circuit in Figure 3-15. The critical path starts at a flip-flop at location (5,5) and goes through the LUT at location (4,6). Generally, a good strategy is to put critical connects on NN interconnect to speed them up, but in the current placement this strategy cannot be used, because the first LUT on the critical path is not in the adjacent CLB. Moving the flip-flop from (5,5) to (5,6) permits this connection to use the NN interconnects, but removes the NN connections from the LUTs at (7,5). By duplicating the flip-flop and LUT at (5, 5) and placing them in the CLB at location (5,6), as shown in Figure 3-16, NN connections can be used for all outputs of the flip-flop.

3.5 Merging

It is sometimes beneficial to reverse duplication that has occurred in previous synthesis, which is allowed in a transformation called merging. For example, after the placement and routing it becomes clear that the distribution of connections between two duplicated components is causing the performance to suffer. This is because duplication during logic synthesis cannot predict the final placement of the circuit.

Merging can be used to redistribute connections between duplicate components. To do this the user selects a pair of identical logic components and uses the merging transformation to merge

them. Once the components are merged, the user applies the logic duplication transformation to recreate the two logic components, but with a different connection distribution.

3.6 Carry Chain Shortening

Carry chains have long been part of FPGA architectures [19][20] because they provide a high-speed path for long bit addition and other arithmetic operations. Their use often reduces the delay along the critical path, or removes the arithmetic operation from the critical path entirely. They come with some drawbacks, however, that reduce their positive impact: most importantly, carry structures force the logic blocks that use them to be a fixed vertical or horizontal structure. This lack of flexibility is the flip side of the greater speed of connectivity.

In addition, the blind use of a full carry chain may prevent other beneficial optimizations. For example, consider the circuit in Figure 3-17, which illustrates a 4-bit carry chain that feeds one more 2-input LUT and then a flip-flop. Assume that the most significant bit of the carry chain, including the LUT, implements a 3-input function. The most significant bit calculation, including carry, and the final 2-input LUT function can all be implemented in a single 4-input LUT as illustrated in Figure 3-18. This operation is called *carry chain shortening*, as it removes the carry primitive from the top of the chain. Most synthesis tools are not permitted to optimize carry primitives away, and so this opportunity is typically unexplored. It may also be possible to shorten the carry chain at the least-significant bit end, if for some reason that bit is part of a critical path.

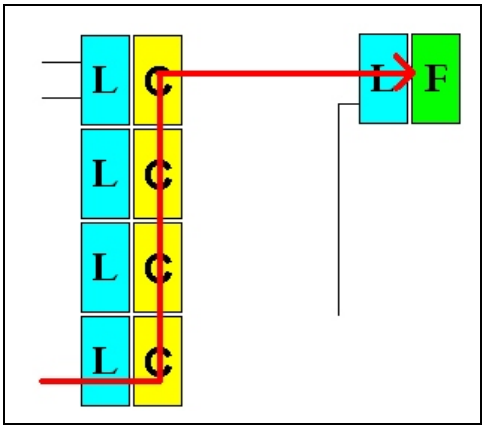

Figure 3-17: Critical path before the application of carry chain shortening

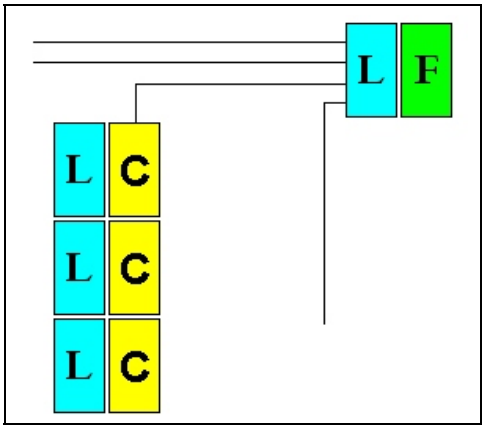

Figure 3-18: Circuit after carry chain shortening

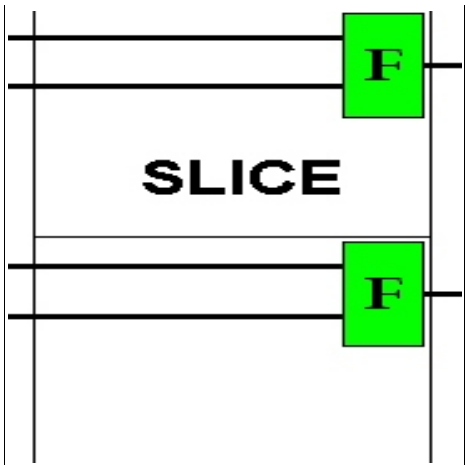

Figure 3-19: Two registers with incompatible control signals

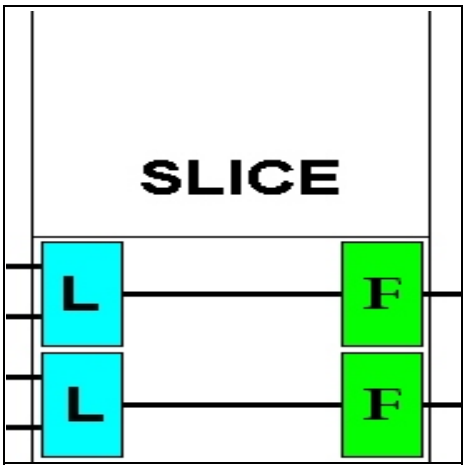

Figure 3-20: Control signal functionality implemented in LUTs

The procedure to test if this optimization is possible is quite straightforward.

3.7 Flip-flop Control Signal Extraction

Recent versions of synthesis tools [14] have used flip-flop control signals to implement greater logic functionality in a single slice. For example, attaching a logic signal to a flip-flop's synchronous clear input has the effect of ANDing that signal with the flip-flop's D input. While this kind of optimization can be beneficial with respect to logic depth, it can also have a negative side-effect: a flip-flop synthesized this way cannot be packed into a slice with another flip-flop that does not use exactly the same clear signal, as shown in Figure 3-19. This poses a restriction on the packer and may degrade the final performance.

The alternative, which is implemented as a logic synthesis transformation, is to implement the synchronous clear function in a separate LUT, as shown in Figure 3-20. This certainly costs extra logic, but it can potentially improve speed because it increases the packing flexibility and therefore local connectivity and access to NN interconnects.

3.8 Summary

This chapter presented the logic synthesis transformation available in Augur. These transformations are used in the context of omniscience to improve the speed of the logic circuit. Each logic transformation is designed specifically for the Xilinx Virtex-E device, taking advantage

of the design of the Virtex-E slice to implement complex logic functions with low delay.

The logic synthesis transformations described here are provided by the manual editor, which allows the user to apply these logic transformations in the context of omniscience. The user experience with the manual editor and the explanation of the method the user can apply are the topics of the next Chapter.

4 The User Experience with the Manual Editor

4.1 Introduction

The previous chapter introduced a set of logic synthesis transformations that Augur performs. These transformations take advantage of the CLB design of the Virtex-E and focus on improving the speed of the logic circuit. This chapter focuses on how Augur provides the user with omniscience and how the logic synthesis transformations are used in the context of omniscience.

Providing omniscience requires a CAD tool to perform routing and timing analysis after every logic synthesis or placement transformation. In addition, a CAD tool must provide all information about the implementation of a logic circuit. In this work omniscience is provided through a manual editor called Augur, which is capable of implementing various physical and logical transformations. Augur provides the user with detailed information about the design implementation at every step of the improvement process.

The input to Augur is a placed and routed logic circuit. The user uses placement, packing and logic synthesis transformations to improve the operating frequency of the logic circuit. The output of Augur is a new placed and routed logic circuit. The logic circuit produced by Augur can be used by commercial tools. The speed of the logic circuit will be as reported by Augur, as the results obtained by Augur are verified by commercial tools at every step of the optimization process.

4.2 Getting Started

To create an input file for Augur the user must first synthesize the logic circuit into an EDIF

file format and then use Xilinx Place and Route tool (called **PAR**) to generate placement and routing for it. The synthesis of the logic circuit must follow three rules:

1. There can only be one clock signal, because Augur does not perform timing analysis of logic circuits with more than one clock.
2. The Synthesis Tool must be set to not create I/O pins, because Augur is intended for use with small modules. It is expected that the placed and routed module will be instantiated in a higher level logic circuit. The assignment of pins should happen at a higher level, because not all input or outputs of the logic circuit will be connected to I/O pins.
3. The primary output nets must have the prefix "END_", to allow Augur to distinguish them for other non-I/O nets.

The logic circuit synthesized this way into an EDF file can then be used to generate placement and routing for the logic circuit using the PAR program. The PAR program will generate an NCD file, which contains the synthesis, placement and routing information about the logic circuit.

The user starts the manual editor with the following command entered in the MSDOS prompt:

editor <filename.ncd>

where the *<filename.ncd>* is the complete filename for the logic circuit that is to be improved.

The manual editor will launch the Xilinx FPGA Editor in the background and display Augur's graphical user interface. The following Sections present how the user utilizes the graphical user interface to interpret the data provided and to improve the speed of the logic circuit.

4.3 Augur's Graphical User Interface

Augur's graphical user interface consists of three distinct views: the Placer/Packer view, the SubCircuit view and the SubCircuit Placer view. When the user first starts Augur, the placer and packer view is presented. The placer and packer view allows the user to modify and see the placement and packing of a logic circuit to improve its speed [6]. In this view the user selects a set of components and moves them to a different location on the FPGA. The SubCircuit view adds the ability to resynthesize the logic circuit, while the SubCircuit Placer view allows the user to place the new netlist created by logic synthesis transforms.

To demonstrate how the user utilizes all of these views, and the functions provided with them, the following subsections will show an example of how each view is used. In each subsection a specific view will be described, showing the information that can be gathered from it. This information assists the user in making an informed optimization decision.

4.3.1 The Placer and Packer View

The Placer and Packer view presents an abstract view of the circuit using three types of cells: LUT, Carry, and Flip-Flop. The LUT cell is a cyan rectangle with a multiplexor icon in the middle. The Carry cell is a yellow rectangle with a plus sign in the middle, while the Flip-flop cell is a green rectangle with a Flip-Flop icon in the center. An example view is shown in Figure 4-1.

The components highlighted in red, and the highlighted arrows between them, show the critical path of the logic circuit. After analyzing each component on the critical path the user may modify the logic circuit by moving a component on the critical path to a different location so that

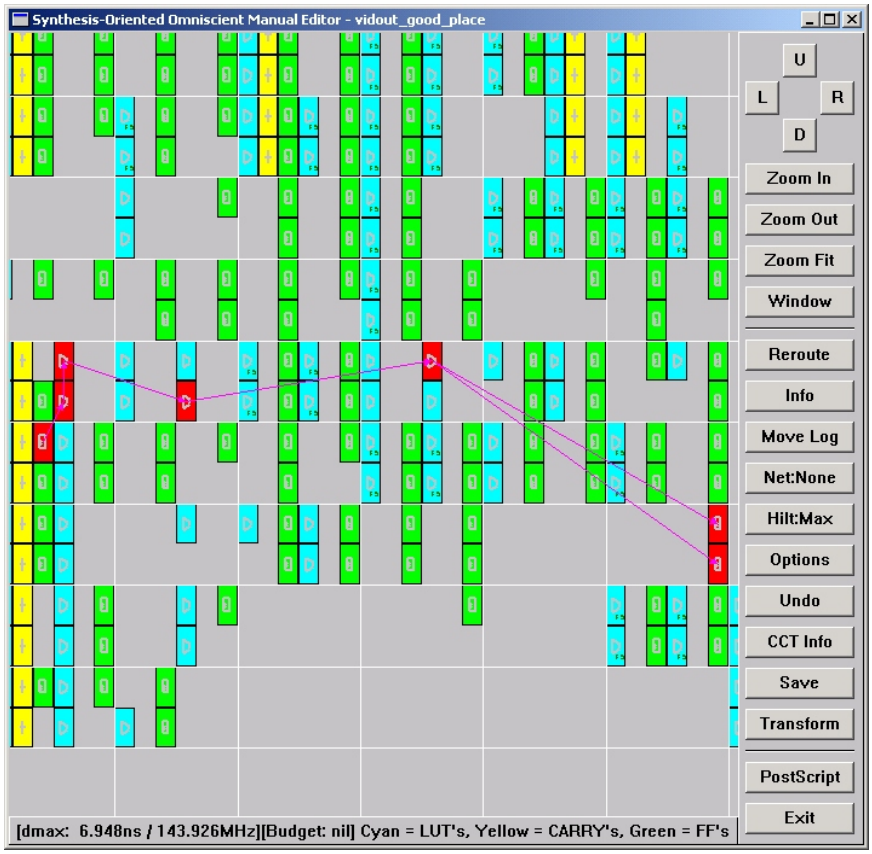

Figure 4-1: The Placer and Packer View

the delay on the critical path is reduced. When the user moves a component that is on the critical path, the manual editor will immediately implement the requested operation, if it is valid, and perform timing analysis to provide the user with the update logic circuit speed.

The most effective method of achieving the delay reduction through placement and packing changes is to create placement that allows connections between logic components to utilize the Nearest-Neighbour (NN) interconnect [8]. To do this the user must place logic components horizontally across the CLB boundaries. In the view these boundaries are marked by vertical white lines. To better see the CLB boundaries, and the logic circuit components, the user can scroll or zoom the Placer View using the view controls.

The View Controls

The view controls, shown in Figure 4-2, consist of buttons which allow the Placer and Packer view to shift up, down, left and right, as well as perform a zoom in and zoom out operation. The four buttons in the top right corner of the window cause the view to shift **U**p, **D**own, **L**eft, and **R**ight. To perform a zoom operation the user can use the **Zoom In**/**Zoom Out** pair of buttons, the **Window** button or the **Zoom Fit** button. The **Zoom In**/**Zoom Out** buttons cause the view to zoom in (or out) towards the center of the view. The **Window** button enables the user to use the mouse to select the area of the design to be enlarged. The same effect is achieved by pressing the right mouse button in the view area, selecting an area to zoom into and then pressing the left mouse button. Finally, the **Zoom Fit** button causes Augur to display the entire logic circuit in the view area. This option is very useful as it gets the user out of the zoom, making it easy shift focus to a different location in the design quickly.

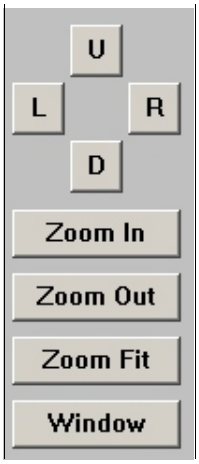

Figure 4-2: View Controls

At this point in the optimization process the user has identified the critical path and zoomed in the view to better observe why the highlighted path has the longest delay and determine if anything can be done to remedy the situation. To proceed further, more information needs to be gathered so that the next step of the optimization can be determined. There are three ways in which the user can obtain information: visual inspection, delay profiling and information dialog box.

Visual Inspection

The user can visually inspect the connectivity of logic components as well as the delay information for a subset of longest paths in the logic circuit. The **Net** button enables the user to observe all, some or no connections between logic components. Initially, the Placer View does not show any connection between components. This is indicated by the state of the Net button (None).

The user can toggle the Net button to be in two other states: Selected and All. When the Net button is in "Selected" state, only the connections for selected components are shown, while in the "All" state every connection in the logic circuit is shown. The "Select" option is the most useful as it shows the user only the connections for the components of interest. The "All" option is useful to determine possible areas of high connectivity.

By looking at the fanout of a single component, the user can determine if it may be beneficial to use logic duplication to reduce the fanout of the logic component in question. A component with a large number of arrows that fanout from it is usually a good candidate for duplication. Furthermore, by locating the logic components that drive the selected component, the user can locate components that may be identical. Identical components will have the same type (LUT, Carry, or Flip-Flop) and will be configured the same way, which would make them possible candidates for merging.

Delay Profiling

In addition to visual inspection, Augur provides the user with the ability to determine the location of paths whose delay is in a user-specified range. Each path starts at a flip-flop or a primary input and ends at a flip-flop or a primary output. Knowing where these paths are located helps in selecting an appropriate logic optimization approach. There are two tools provided by Augur to help the user determine the delay and location of paths in the logic circuit: the delay profile and the delay budget [6].

The delay profile is a histogram of paths and their delays. Each path is associated with a bin, where a bin has an upper and a lower bound on the delay of paths that belong to it. The delay profile for the example in Figure 4-1 is shown in Figure 4-3. The bins are arranged so that bin 1 contains all paths which operate within a speed of 1 MHz of the critical path, while every consecutive bin

```
Geometric bin delay profile :
Bin 9: 3.308ns-4.553ns (302.337MHz-219.613MHz), count = 1239
Bin 8: 4.553ns-5.384ns (219.613MHz-185.734MHz), count = 855
Bin 7: 5.384ns-5.938ns (185.734MHz-168.413MHz), count = 877
Bin 6: 5.938ns-6.307ns (168.413MHz-158.556MHz), count = 523
Bin 5: 6.307ns-6.553ns (158.556MHz-152.601MHz), count = 249
Bin 4: 6.553ns-6.717ns (152.601MHz-148.874MHz), count = 78
Bin 3: 6.717ns-6.826ns (148.874MHz-146.489MHz), count = 56
Bin 2: 6.826ns-6.899ns (146.489MHz-144.940MHz), count = 21
Bin 1: 6.899ns-6.948ns (144.940MHz-143.926MHz), count = 9
```

Figure 4-3: Delay profile showing nine delay bins

speed range grows by 50%. This arrangement of bins allows the user to determine the overall number of paths that are close to critical and then use the delay budget to locate these paths in the logic circuit.

The delay budget is the means by which Augur displays paths that do not meet the timing constraints. The Options window, shown in Figure 4-4, allows the user to specify the delay budget at any time. To specify the delay budget for the logic circuit, the user must press the **Options** button and then enter the logic circuit delay in the delay budget edit box [6].

Once the delay budget is set all paths in the logic circuit with the delay longer than specified by the budget are highlighted. The user can use the **Hilt** button to toggle the display to show only the critical path, the paths that do not meet the timing budget or no paths.

The Hilt button has three states: None, Max and Slow. The *None* state disables highlighting of paths in the logic circuit, while the *Max* state displays only the longest path in the logic circuit. The *Slow* state is used to displays all the paths that have the delay longer than the specified budget.

The budget can be set to any delay value. However, using the delay profile to pick a good delay budget aids the user in improving the speed of the logic circuit. For example, the logic circuit in Figure 4-1 has the delay profile shown in Figure 4-3. By setting the delay budget to show all the paths in bin 1, by setting the delay budget to 6.899ns, highlights the nine critical paths. Figure 4-5 shows the critical paths using the highlighting ability of Augur.

One of these paths traverses the carry chain, near the left edge of the figure, while the remaining eight paths start at the same flip-flop, go through three common LUTs and then fanout

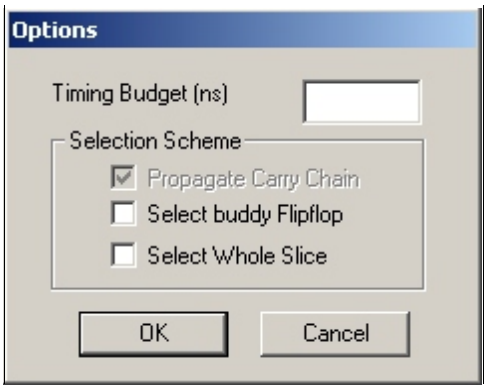

Figure 4-4: The Options window

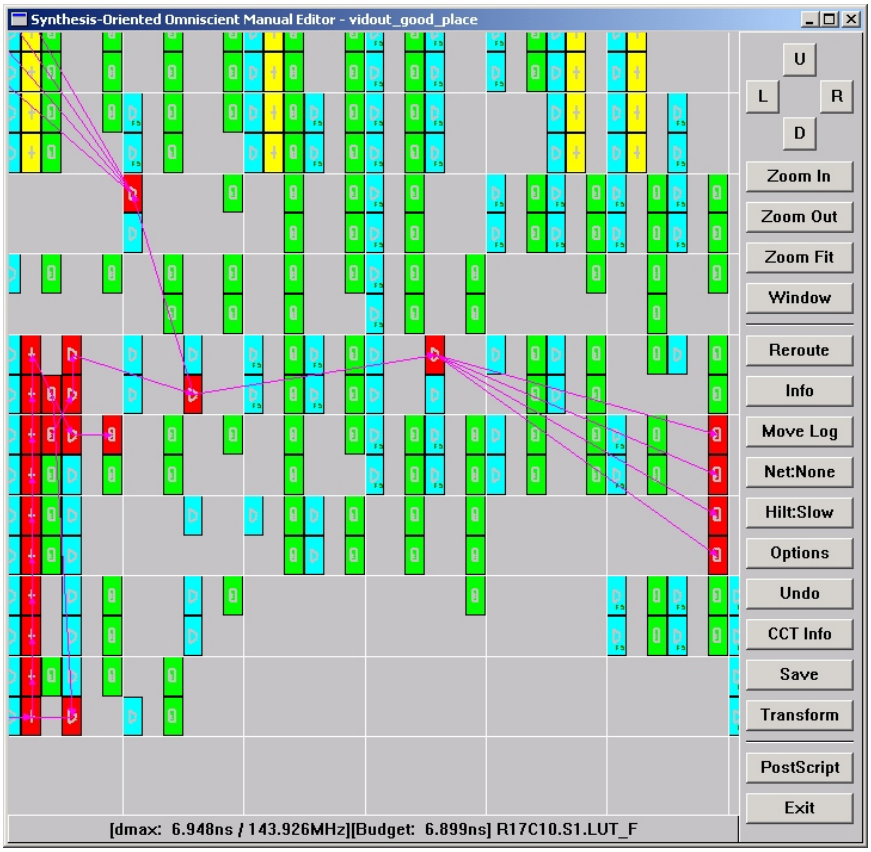

Figure 4-5: Critical paths with delay greater than specified budget

into two different directions. These paths overlap the longest path in the logic circuit, shown in Figure 4-1. Therefore, improving the performance of the logic components that are shared between the eight paths would improve the performance of eight critical paths.

One way to attempt improving the speed of these critical paths is to move the originating flip-flop one row up, so that the Nearest Neighbour interconnect can be utilized. However, the flip-

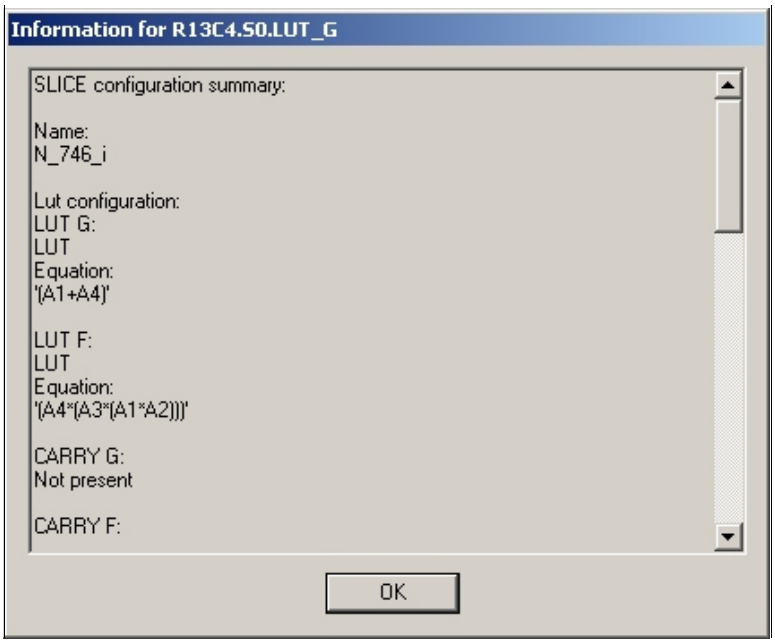

Figure 4-6: Information box

flop is driven by the carry chain structure and moving it from its current location significantly increases the delay of paths passing through the carry chain. Moving the LUTs one row lower and to the right is also not a good idea as some of the critical paths will have their delay increased. Therefore, the only alternative is to search for a logic transformation that will decrease the delay on these eight paths. To find out if some of these logic components can be resynthesized such that the delay through them is decreased the user needs the information about the logic function these components implement. This is obtained using the Logic Component Information box.

Logic Component Information Box

The information box displays all information about the implementation of logic components. The dialog box contains a scrollable list, which lists the logic function implemented by LUTs, the configuration of carry chain components and the settings associated with flip-flops. In addition, each input connection between the logic components is given with the delay associated with it.

The logic function implemented by the first two LUTs on the critical path of Figure 4-1 is given in the information box in Figure 4-6. The first LUT on the critical path is the LUT implemented in the bottom half of the slice, also known as LUT F. The second LUT is implemented

in the top LUT of the same slice, also known as LUT G. Figure 4-6 states that the LUT G implements an OR function, while the arrow connecting these logic components in the Placer View indicates that one of the inputs to LUT G is the output of LUT F. This means that the entire function has 5 inputs. Therefore it may be a candidate for a Joint-LUT mapping, described in Section 3.3.2. To perform a logic transformation the user must select the two LUTs and then press the **Transform** button.

Transform Button

This button allows the user to select one the logic transformations available in Augur. When pressed the menu on the right hand side of the window presents the user with the following options:

1. Back - do not perform any logic transformation
2. Duplicate - perform logic duplication
3. Merge - perform logic merging
4. Remap - initiate logic remapping, which includes carry chain shortening
5. Ctrl Ext - perform flip-flop synchronous control signal extraction

Selecting any logic transformation switches the view to the SubCircuit view, where the user is able to perform logic synthesis transformations. In this example, the user chooses the **Remap**ping transformation.

4.3.2 The SubCircuit View

The SubCircuit view is the means for the user to see the dependency between logic components and select subsets of these components to perform logic synthesis transformations. Once the logic transformations are completed the user must switch to the SubCircuit Placer view to place the logic components. Once this process is complete the user presses the **Accept** button to implement the specified changes in the logic circuit and obtain the new logic circuit speed. Alternatively, the user can press the **Reject** button to abort the resynthesis procedure.

The SubCircuit view is organized such that the logic sub-circuit selected in the Placer and Packer view appears in the topological order from the left to the right side of the screen. The logic synthesis transformations that can be selected in this view are shown on the right hand side of the

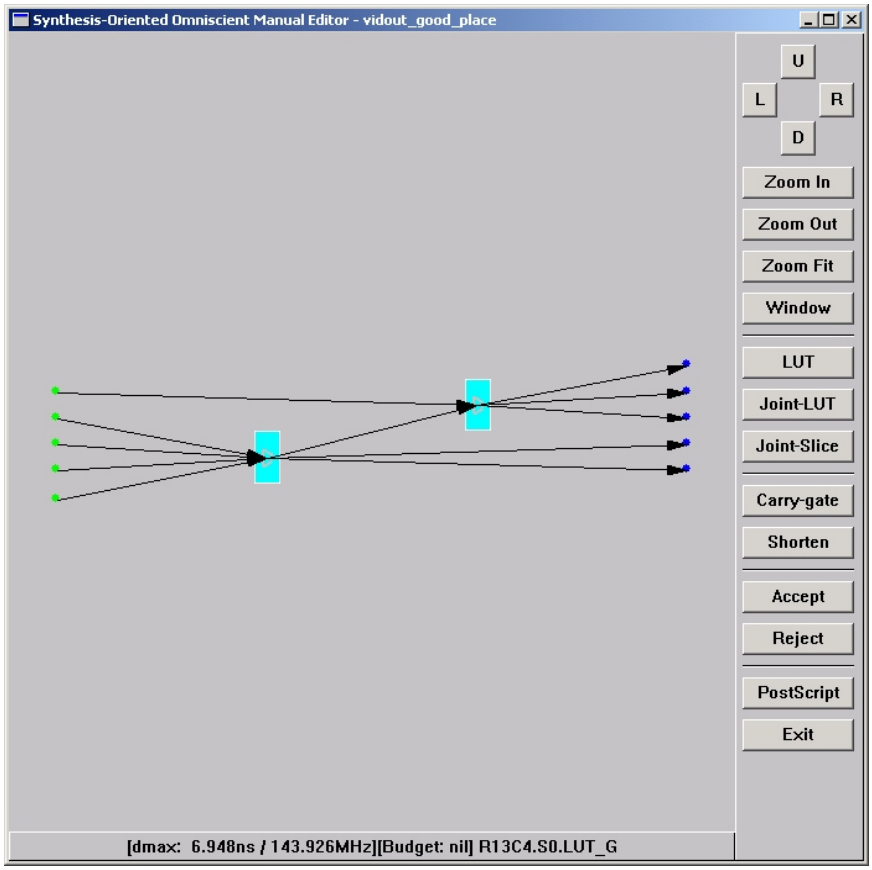

Figure 4-7: The SubCircuit view

screen in the form of buttons. When the user selects a set of components and presses one of the buttons responsible for a logic synthesis transformation, Augur will attempt to implement the selected logic components using the selected logic synthesis transformation.

The SubCircuit view for the logic components selected in Figure 4-5 is shown in Figure 4-7. To perform a logic synthesis transformation that implements these components in the Joint-LUT structure the user must select both components and press the **Joint-LUT** button. A successful transformation will generate the view shown in Figure 4-8.

In this view the dark gray box, which contains the two LUTs, represents a placement dependency between the two LUTs. These two LUTs have been implemented such that a Joint-LUT structure is formed. Therefore, when these LUTs are placed they must be in the same slice and in the same relative position. Augur will always keep track of the relative position of these components and will not allow them to be placed separately. To place the Joint-LUT structure created by the

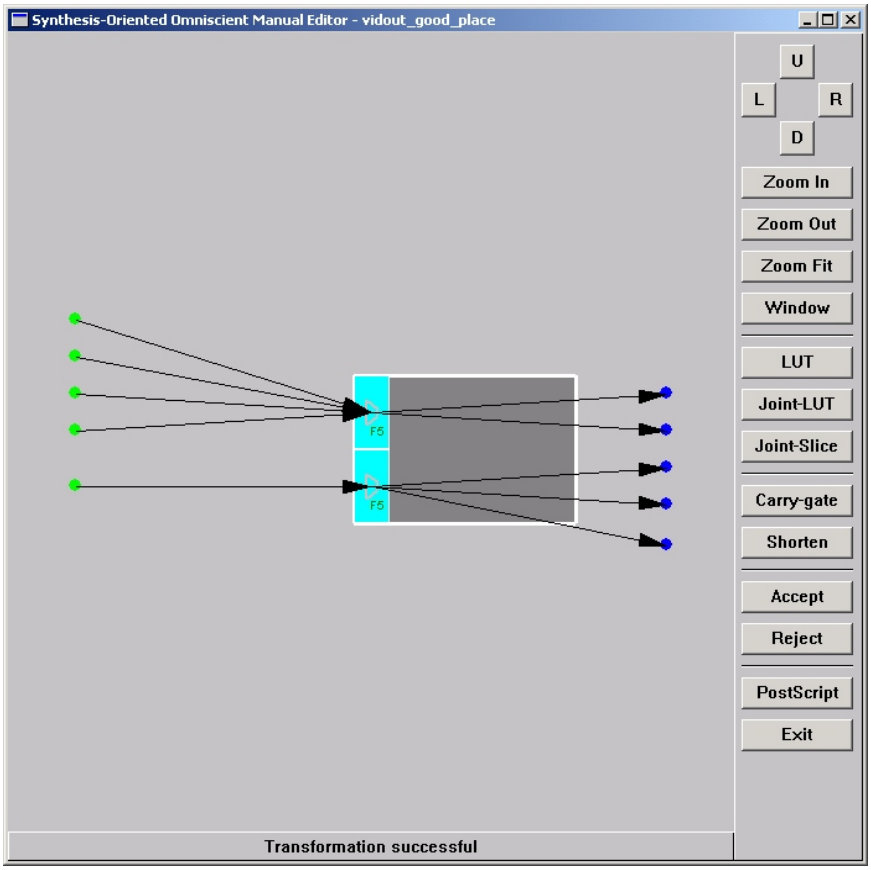

Figure 4-8: The SubCircuit view after a logic transformation

logic synthesis transformation the user must select the structure and press the *TAB* key. This action switches the view to the SubCircuit Placer view, where the user is able to look at the logic circuit and place the newly synthesized components in a suitable location.

4.3.3 The SubCircuit Placer View

The SubCircuit Placer view provides the means for the user to place the re-synthesized logic components into the logic circuit. The SubCircuit Placer view does not allow components that are not part of the subcircuit to be moved. Once the placement for every logic component is specified, the user may press the **Accept** button in order for the change to take effect, or press the **Reject** button to abort the logic transformation entirely. Figure 4-9 presents the SubCircuit Placer view with a placed Joint-LUT for the example in Figure 4-8.

The SubCircuit Placer view is similar to the Placer and Packer view, except that once a logic

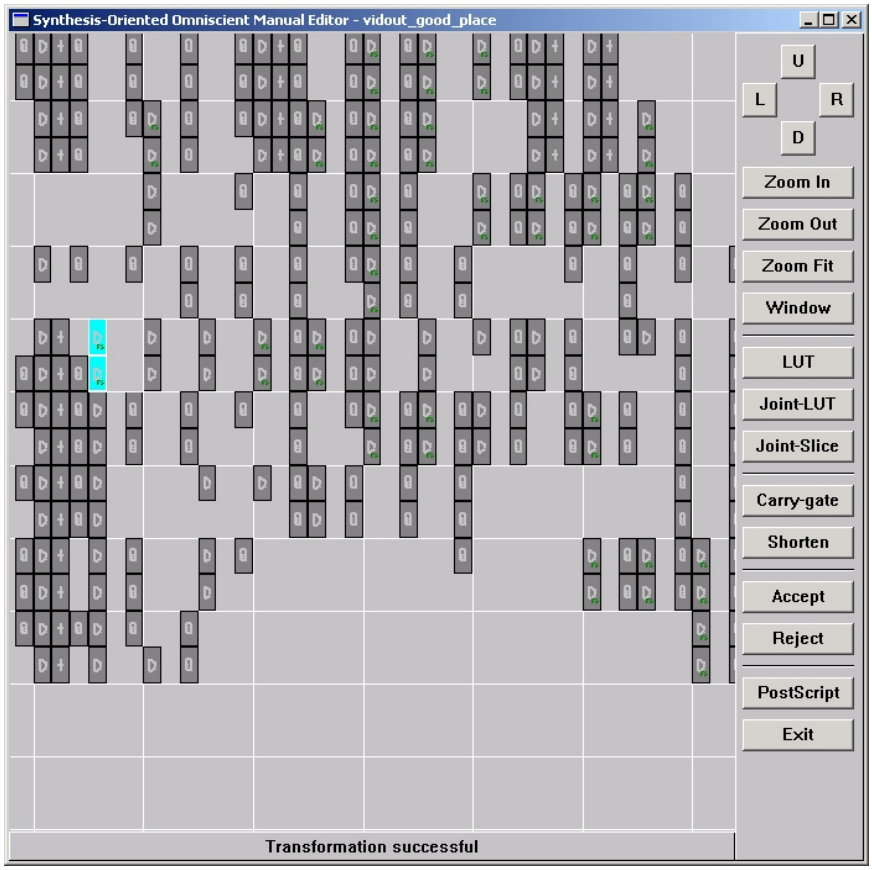

Figure 4-9: SubCircuit Placer view

component is placed the manual editor will not implement the desired change until the user presses the **Accept** button. As a consequence, the logic components that are not a part of the subcircuit are grayed-out to signify that they cannot be moved.

To help the user place the logic component rubber bands are shown whenever a logic component is being moved in the SubCircuit Placer view. The rubber bands are semi-transparent arrows that show the sources and targets of connections associated with a particular logic component. The rubber bands only show connection to placed logic components. Therefore, if a logic component is being placed that has a connection to another logic component that is not yet placed, then the rubber band that corresponds to the connection between these logic components will not be shown.

The user can also remove logic components from the SubCircuit Placer view. To perform this action the user must select the desired component and make it move. While the SubCircuit

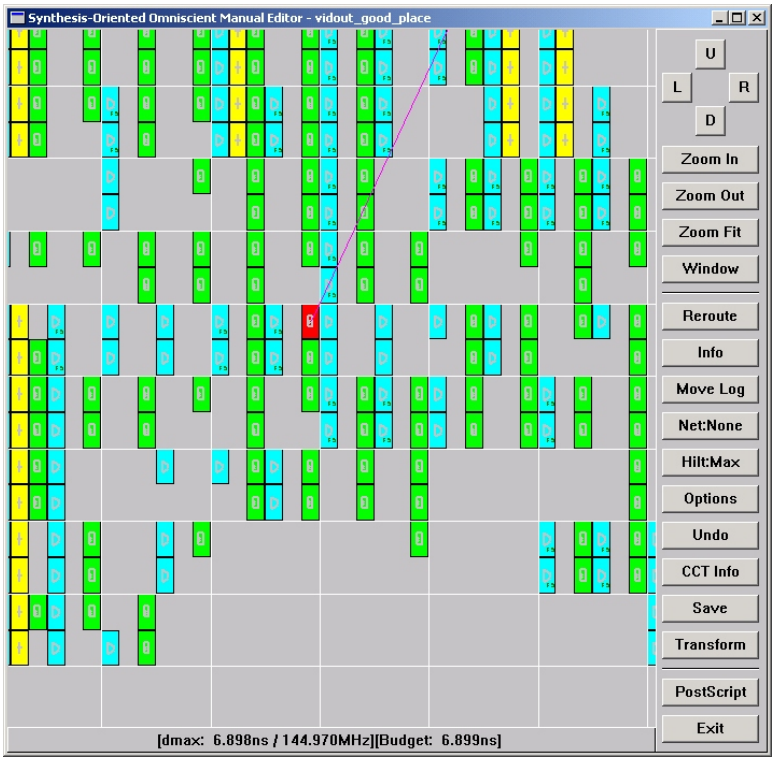

Figure 4-10: The Placer and Packer view after accepting the logic synthesis transformation

Placer view is moving the selected logic component and the user presses the TAB key the user will be returned to the SubCircuit view and the selected component be removed from the SubCircuit Placer view.

In Figure 4-9 the placement for the resynthesized logic has been selected. Since all of the logic components have been placed the user can now press the **Accept** button to see the result of implementing the logic synthesis transformation. The result of this transformation is presented in Figure 4-10. This action takes the user back to the Placer and Packer view.

Once back in the Placer and Packer view the user can observe the effect of the logic synthesis transformation on the speed of the logic circuit. From this point the process of logic circuit modification becomes iterative. The user can again focus on a logic subcircuit, gather information about it and select a logic circuit modification that best suits the given situation. Once the user has completed all modifications, or the desired logic circuit speed is achieved, the file commands can be used to save the logic circuit.

4.4 File Commands

While working with the manual editor, the user can keep intermediate results by using the **Save** button. The save button allows the user to create a copy of the logic circuit and save it in a NCD type file, which can be processed by Xilinx tools. This operation will not change the logic circuit source file unless the name of the file the logic circuit is to be saved to is identical to the initial file name.

It is helpful to manually save the logic circuit on occasion, as it allows the user to go back to the logic circuit implementation that the user considered to be good. Another option which allows the user to do a similar thing is the **Undo**. By pressing the Undo button the manual editor will immediately undo the last logic circuit modification, restoring logic synthesis, placement and routing of the logic circuit. The Undo command is capable of undoing all logic circuit modifications that occurred since the start of the program.

4.5 Leaving Augur

To quit Augur press the **Exit** button at any time. The manual editor will close the user interface, close the FPGA Editor that was opened at the start of the program and return control to command prompt. If the logic circuit was modified since the last save operation, the program will ask if the user wishes the logic circuit to be saved before quitting.

4.6 Software Organization

The previous Sections of this chapter presented the user experience with Augur. In this Section the software design of Augur is described briefly.

Augur consists of four modules: the Graphical User Interface (GUI), the manual placer and packer, the synthesis module, and the FPGA Editor Interface, as shown in Figure 4-11. The GUI provides the means for Augur to communicate with the user and present data necessary to improve the speed of the logic circuit. The commands are relayed from the GUI to the manual placer and packer, which enables the user to modify the placement and packing of a logic circuit. In a case where the user request a logic synthesis change, the manual placer and packer forwards that request

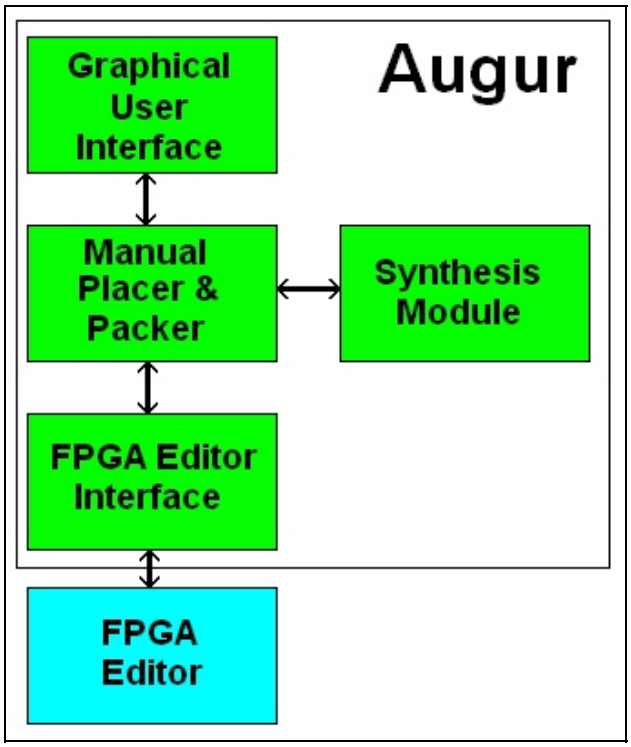

Figure 4-11: Augur software overview

to the synthesis module.

Upon receiving a logic synthesis change request the synthesis module accesses the data structures, which represent the logic circuit to obtain a complete picture of the subcircuit that is to be modified. The subcircuit is then modified according to user specifications. The synthesis module then asks the user to perform placement of new logic components by asking the manual placer and packer to provide access to the proper GUI view, as described in Section 4.3. Once the modification is accepted by the user the synthesis module transfers the logic circuit modification back to the manual placer and packer.

To transfer the logic circuit modification incrementally the synthesis module uses the data structure in Figure 4-12. The data structure in Figure 4-12 represents a list of actions that need to be performed on the logic circuit to implement the logic circuit modification. The *type* field specifies what kind of modification is requested, such as add or delete a net. The *transformation_data* field provides detailed information about the action that needs to be performed. For the case of adding a net, the source pin location and target pin locations are included in the

```
typedef struct s_action_list {
        action_type             type;
        t_action_data *transformation_data;
        struct s_action_list *previous, *next;
} t_action_list;
```

Figure 4-12: Action list data structure

transformation_data field. Finally, the *previous* and *next* fields specify a link to the last action that was performed and the next action to be performed to implement the logic modification. Although it is sufficient to provide only the link to the next action to be performed to complete the logic modification, the link to the previous action allows the same set of actions to be reversed to undo the logic transformation in case the modification could not be implemented.

Once a placement, packing or synthesis modification is complete the manual placer and packer module uses the FPGA Editor Interface to communicate the changes to the FPGA Editor. The FPGA Editor is a Xilinx software, which runs in the background. The FPGA Editor allows Augur to implement placement, packing or synthesis modification on a real FPGA device and perform timing analysis. The result of timing analysis is returned to the manual placer and packer module, through the FPGA Editor Interface, so that it can be displayed to the user using the GUI.

4.7 Summary

This chapter has presented the user interface of Augur and a method of using the features of Augur to improve the speed of the logic circuit. The two aspects of omniscience, which are the gathering of information from the design and instant feedback after design modification, were presented in the context of interaction with the user.

In this chapter the process of improving the logic circuit is also shown. The user begins the process by analyzing the data in the Placer and Packer view to decide how to improve the logic circuit. The user can use placement or packing modifications to improve the circuit or apply a logic synthesis transformation. The SubCircuit view and the SubCircuit Placer view facilitate the implementation of logic synthesis transformations. After each modification the logic circuit undergoes timing analysis to provide the user with the effect of the logic circuit modification on the

speed of the logic circuit.

The approach presented in this chapter has been applied to a suite of benchmark logic circuits. The improvements resulting from the use of Augur as well as the strategies used during the logic circuit improvement process are presented in the following chapter.

5 Experimental Results

5.1 Introduction

The goal of this work was to create an omniscient manual editor (Augur), which uses logic synthesis transformation as the means to improve the speed of logic circuits. In Chapter 3 the logic synthesis transformations that are available in the manual editor were described, while Chapter 4 presented the method by which the user employs Augur. This chapter presents the results obtained using the editor. In addition, several strategies that can be automated and form the basis for algorithms in automatic CAD tools are presented.

To evaluate the performance improvement obtained using Augur, a fair basis of comparison is needed. The reference point chosen in this work is a suite of 10 benchmark logic circuits, which are synthesized, placed and routed using latest commercial CAD tools. Each of the logic circuits in the benchmark suite is briefly described in section 5.2. To ensure that the comparison of results obtained by the use of Augur to those obtained by automatic CAD tools is fair, a rigorous method of generating the benchmark suite was devised. This method is described in section 5.3.

To evaluate the effectiveness of this new approach, the results for each benchmark logic circuit were obtained in two phases. The first phase was to apply the approach of Chow and Rose [6], using only placement and packing modifications to improve the speed of the logic circuit. The results of this approach are discussed in section 5.4. The second phase was to employ logic synthesis transformations. The results obtained using these logic synthesis transformations are described in section 5.5.

Logic Circuit Name	Size		Operating Frequency (MHz)
	LUTs + Carry	FFs	
Batcher	252	436	298.6
Miim	162	119	155.0
Vision	310	243	197.4
Banyan	176	335	359.3
Trap	186	486	381.0
Boundcontroller	472	466	131.5
Linearmap	460	72	108.0
Vidout	447	220	134.4
Raygencont	211	118	162.1
Mult	29	21	122.2

Table 5-1: Statistics for benchmark logic circuits

Section 5.6 describes several optimization strategies that emerged during the course of improving logic circuits using Augur, that can be automated.

5.2 Benchmark Circuits

The benchmark logic circuits come from designs made at the University of Toronto and IP cores available through the internet. The summary of the logic circuit statistics is presented in Table 5-1 along with the baseline performance for each circuit obtained using commercial synthesis and P&R tools. The logic circuits in the benchmark suite are:

1. **Batcher** - An ATM packet-sorting circuit that serially compares bits of two packets. It is a part of the StarBurst ATM chip [16].
2. **Miim** - The MII Management module of an Ethernet IP core obtained from [5].
3. **Vision** - An FIR filter circuit used in [11]. It is a highly pipelined circuit primarily consisting of shifters and adders.
4. **Banyan** - From the StarBurst ATM [16], a packet router that delivers packets to ports specified in the packet address field.
5. **Trap** - From the StarBurst ATM [16], a duplicate packet detector.
6. **Boundcontroller** - A controller for the hardware ray tracing rendering system [17].
7. **Linearmap** - A 2D to 1D coordinate mapping circuit. Used to calculate the offset in

1. Set target frequency to any value, synthesize, place and route the design. The resulting operating frequency will be used as initial setting for the remainder of this procedure
2. Set target frequency to initial setting
3. Synthesize using Synplify 7.1 Pro [14]
4. Place and route tool (par.exe) [15] provided with Xilinx ISE 5.1 Service pack 3 tools. The placement and routing is performed 100 times, each time with a different seed.
5. Record best result
6. Repeat 3-5 for target frequency -10%, -5% +5% and +10% with respect to current
7. If a better solution was obtained in 6 then repeat 2-6 using the frequency setting that produced a better result as initial setting.

Figure 5-1: Procedure to obtain baseline performance for a benchmark circuit

memory to look up textures based on given (x, y) coordinates. Also a part of the hardware ray-tracing engine [17].

8. **Vidout** - module used to display a rendered image using the VGA interface on the Transmogrifier-3 [18], and also part of the hardware ray-tracing engine [17].
9. **Raygencont** - a circuit that generates all rays to be traced for a given view. Part of the hardware ray-tracing engine [17].
10. **Mult** - a 4x4 bit multiplier.

The next section describes the process used to generate the baseline results for each of the benchmark logic circuits.

5.3 Baseline Comparison

To achieve the best possible baseline logic circuits a very rigorous procedure using the best-in-class tools was created. Figure 5-1 summarizes the procedure, the key of which is that each circuit is placed and routed 100 times for at least five different target frequency settings. During each place and route run the Xilinx Place and Route tool uses a different seed.

This method is not practical for large circuits, but the limit imposed on the logic circuit size allowed the baseline generation to be completed in a reasonable amount of time. Compared to the method used by Chow and Rose [6] to generate the baseline logic circuits, this method results in logic circuit performance that is on average 2.4% better.

Logic Circuit Name	Baseline Frequency (MHz)	Frequency after Placement and Packing (MHz)	% Improvement
Batcher	298.6	314.0	5.1
Miim	155.0	155.2	0.1
Vision	197.4	197.8	0.2
Banyan	359.3	367.8	2.4
Trap	381.0	398.6	4.6
Boundcontroller	131.5	137.9	4.8
Linearmap	108.0	109.3	1.3
Vidout	134.4	140.0	4.1
Raygencont	162.1	173.2	6.8
Mult	122.2	122.3	0.1
Average			**3.0**

Table 5-2: Results using only placement and packing modifications

5.4 Placement and Packing Results

To separate out the additional advantage of the synthesis transformations described in Chapter 3, the baseline logic circuits were first improved using only packing and placement modifications, using the approach described by Chow and Rose [6]. We tried to improve the placement and packing of every logic circuit for about 3 hours. Table 5-2 provides a summary of these results. The results show an average of 3.0% performance improvement across the 10 circuits. This is significantly less than the 12.7% achieved by Chow and Rose [6]. The following six reasons are largely responsible for the difference in performance improvement obtained using only placement and packing modifications:

1. Newer placement and routing tools are used (Xilinx ISE 5.1 service pack 3 vs. 3.3 SP7)
2. Better Synthesis tools are used (Synplicity Synplify 7.1 Pro vs. Synplify 6.2 Pro)
3. The method of generating baseline logic circuits is better than in [6]
4. The benchmark suite in this work has 25% more logic circuits
5. Only 4 of the 10 logic circuits in this benchmark suite are the same as in [6]
6. A human factor has to be taken into account as well, since the placement and packing of a logic circuit was not a predominant factor in this work

Logic Circuit Name	Logic Circuit after placement and packing modifications only			Logic Circuit after re-synthesis			Improvement (%)	
	LUT + Carry	FFs	Operating Frequency (MHz)	LUT + Carry	FFs	Operating Frequency (MHz)	With respect to baseline	Due to logic synthesis transformations
Batcher	252	436	314.0	252	447	374.8	25.5	19.4
Miim	162	119	155.2	164	119	155.6	0.4	0.2
Vision	310	243	197.8	320	243	210.2	6.5	6.3
Banyan	176	335	367.8	176	335	367.8	2.4	0.0
Trap	186	486	398.6	187	501	418.4	9.8	5.0
Boundcontroller	473	466	137.9	480	469	149.5	13.7	8.5
Linearmap	460	72	109.3	460	76	125.7	16.4	14.9
Vidout	447	220	140.0	454	221	155.6	15.8	11.2
Raygencont	211	118	173.2	211	118	173.2	6.8	0.0
Mult	29	21	122.3	28	25	124.3	1.7	1.6
Average							**9.9**	**6.7**

Table 5-3: Speed improvement results using the new logic synthesis transformations

The new placement and routing tools were successful in performing a number of placement and packing modifications that the ISE 3.3 SP7 was not. In addition to that the Synplify 7.1 Pro version utilized the multiplexers in slices more often than version 6.2, essentially making use of the Joint-LUT and Joint-Slice structures. While this can reduce the delay it also places some restrictions on placement perhaps contributing to diminished improvements using just placement and packing modifications. Finally, the baseline logic circuits were created using a more rigorous procedure than in [6], which made it significantly harder to improve logic circuits using placement and packing modifications only.

5.5 Results Including the New Logic Synthesis Transformations

This section presents the speed improvement results we obtained using Augur when manual packing, placement and logic synthesis transformations are employed. We made an attempt to improve each logic circuit until no further improvement was deemed possible. The conditions for

Logic Circuit Name	Logic Circuit after placement and packing modifications only			Logic Circuit after re-synthesis			Area Increase (%)	
	LUT	Carry	FFs	LUT	Carry	FFs	LUT + Carry	FFs
Batcher	252	0	436	252	0	447	0	2.5
Miim	149	13	119	151	13	119	1.2	0
Vision	195	115	243	197	123	243	3.2	0
Banyan	176	0	335	176	0	335	0	0
Trap	186	0	486	187	0	501	0.5	3.1
Boundcontroller	383	90	466	390	90	469	1.7	0.6
Linearmap	270	190	72	270	190	76	0	5.6
Vidout	317	130	220	325	129	221	1.6	0.5
Raygencont	166	45	118	166	45	118	0	0
Mult	29	0	21	28	0	25	-3.5	19.0
Average							**0.5**	**3.1**

Table 5-4: Area change due to the new logic synthesis transformations

stopping logic circuit improvement are described as a part of the stopping criterion in Section 5.6.4. The speed improvement presented in Table 5-3 was determined by calculating the per cent difference between the speed of the logic circuit in Table 5-3, column 7, and the baseline logic circuit speed in Table 5-1, column 4.

The results obtained for the 10 benchmark logic circuits show that using Augur improves the speed of logic circuit of up to 25.5% and an average of 9.9%. The speed improvement did not cause a severe area penalty. In the worst case, a total of 16 LUTs, carry elements and flip-flops were added to the logic circuit **Trap**. On average the number of LUTs and carry elements has increased by 0.5%, while the number of flip-flops has increased by 3.1%. The speed and area results are summarized in Tables 5-3 and 5-4 respectively.

The speed improvement results are divided into two categories. The first category is the logic speed improvement obtained using placement, packing and logic synthesis transformations, which provides the logic circuit speed improvement when compared to the baseline logic circuit speed. The second category is the logic circuit speed improvement that was possible due to the introduction of logic synthesis transformations. The contribution of logic synthesis transformations was

determined by calculating the per cent difference between the frequency in column 7 and column 4.

The following are the key steps involved in improving each logic circuit:

1. **Batcher** - the majority of the flip-flops in this designed were synthesized to use distinct control signals. These control signals were used to minimize the combinational logic part of the circuit. However, the resulting packing allowed only one register to occupy a slice, which spread the circuit over a larger area than necessary. The application of Flip-Flop control signal extraction (described in Section 3.7) allowed previously incompatible flip-flops to share a slice, resulting in an overall 25.5% improvement over the baseline logic circuit speed.
2. **Miim** - duplication (described in Section 3.4) was used to improve performance of some of the paths in the circuit. However, the complexity of the design, as well as the congestion in the critical region, allowed for only minor (0.6%) improvements.
3. **Vision** - this logic circuit suffered from improper synthesis of logic that controlled Flip-Flop enable signals. A detailed analysis of these signals showed that each of the enable signals was functionally identical, while the logic function for these signals was a 7-input AND gate. The logic synthesis tools implemented this logic function as a pair of serially connected LUTs and duplicated the forward LUT to reduce fanout. Logic merging (described in Section 3.5) was first used to create a single logic function to implement the enable signal. Then both LUTs were duplicated (described in section 3.4) to reduce the fanout and distribute the connection properly. Further improvement was obtained by implementing these two LUTs in a carry chain, using Carry Chain Remapping transformation (described in Section 3.3.1). The resulting logic circuit speed exceeded the baseline speed by 6.5%.
4. **Banyan** - each path in this logic circuit contains at most one LUT. This was possible because of the use of flip-flop control signals to reduce the logic depth of the logic circuit, when synthesized by Synplify 7.1 Pro. We were only able to improve the performance of the logic circuit by modifying the placement and packing. The resulting speed improvement was 2.4% over baseline.
5. **Trap** - in this logic circuit a few registers used flip-flop control signals to decrease logic

depth. However, it was critical to circuit performance that these flip-flops had the freedom to share a slice with other flip-flops. We used control signals extraction transformation (described in Section 3.7) to achieve placement flexibility for these flip-flops. Once these flip-flops had the flexibility to share a slice with other flip-flops, we were able to modify the placement and packing of the logic circuit effectively. In combination with logic duplication (described in Section 3.4) the logic circuit speed was increased by 9.8%.

6. **Boundcontroller** - the design contained a number of Joint-LUT configurations. A closer examination revealed that remapping certain pairs of them into Joint-Slice structures with multiple outputs was possible. After these pairs of Joint-LUTs were remapped into Joint-Slice type structures the placement of the logic components was rearranged to promote usage of NN interconnect. These modifications resulted in the improvement in the logic circuit speed by 13.7% compared to the baseline logic circuit speed.
7. **Linearmap** - the design contains mostly carry chain logic. The problem was that a few registers were driving multiple carry chains and could not use NN interconnect for all connections. Duplicating (described in Section 3.4) some of those registers and re-synthesizing non-carry chain logic into Joint-LUT and Joint-Slice structures (described in Section 3.3.2) improved the logic circuit speed by 16.4%.
8. **Vidout** - the logic circuit contained a carry chain that was unnecessarily long. The output of the top carry cell was not driving the local register, which made it a candidate for the carry chain shortening transformation (described in Section 3.6). Further analysis revealed that the functionality implemented by the top segment of the carry chain and the logic function implemented by the LUT that the carry chain was driving could be implemented in a single LUT. This modification allowed for further logic optimization resulting in the improvement in the logic circuit speed by 15.8% compared to the baseline logic circuit speed.
9. **Raygencont** - the critical path of this logic circuit traverses LUTs that could be re-synthesized into wide AND gate carry chain structures (described in Section 3.3.1). However, that causes the near critical paths to become critical with longer delay. Without logic synthesis transformations we were able to modify the placement and packing of the

logic circuit to improve the speed by 6.8%.

10. **Mult** - the original placement and synthesis was good, however performing logic duplication and remapping improved the logic circuit speed by 1.7%.

5.6 Optimization Strategies

The previous Sections of this chapter presented the specific steps used to obtain logic circuit improvement results and the results themselves. Although the results in this thesis are obtained manually, one of the long-term goals of this research is to discover new optimization strategies that could be fully automated. In this Section several such strategies are proposed, which could form the basis of future algorithms.

5.6.1 Promoting Nearest-Neighbour Interconnect

The first strategy focuses on the nearest-neighbour routing (NN) architecture of the Virtex-E. The best optimization opportunities are those that enable many critical connections to use NN interconnect. The essence of this strategy is to move logic elements so that they can take advantage of available direct connections between logic blocks, as well as liberate the NN interconnect that is occupied by non-critical logic. The latter can be done by logic transformations such as remapping and merging.

Remapping can be used to liberate an NN interconnect by transforming the implementation of serially connected pair of LUTs to a Joint-LUT or Joint-SLICE structure. This replaces the NN connection between LUTs by an internal-to-the slice connection, freeing the NN for use by other critical connections.

Here is an outline of the optimization strategy:

1. Create an ordered list **L** of critical and near-critical path, ordered from the longest delay to shortest
2. For each path in the list:
    a. Let P_i be the selected path
    b. For each pair of serially connected logic components on P_i determine the pairs that can be remapped into Joint-LUT or Joint-Slice structures (described in Section 3.3.2)

and put them in set **S**

c. If set **S** is not empty then attempt to resynthesize P_i so that the NN interconnect is not required, otherwise

d. If set **S** is empty and every connection on P_i that can use NN interconnect is using it then *abandon this strategy*, otherwise

e. If set **S** is empty and some connections on Pi are not utilizing NN interconnect then for each connection that could use a NN interconnect on P_i

   i. Locate the path P_j, which uses that the NN interconnect P_i needs

   ii. Check if a components on P_j can be moved or remapped to free the NN interconnect, while ensuring that the delay of P_j is less than the delay of P_i

   iii. If step (ii) is unsuccessful then *abandon this strategy*

If successful, this should allow the critical logic to acquire the liberated NN connection.

5.6.2 Liberating Free Space for Critical Logic

The focus of the second strategy is to free the logic components in a slice or CLB that can be utilized by critical logic. The essence of this strategy is to examine logic components on the critical path and search for a suitable placement for them. When the desired placement is found to conflict with non-critical logic then the non-critical logic must be moved out of its current location, without the loss in logic circuit speed.

The logic transformations liberate space occupied by non-critical logic by:

- Speeding up non-critical logic, allowing it to move from its current location, without a performance penalty
- Duplicating components with high fanout, allowing them to be placed in different locations of the device, while maintaining the circuit performance

The following steps can be used to liberate space to improve performance:

1. For each area containing a critical, or close-to-critical, paths:

   a. Locate the non-critical logic

   b. Move or duplicate (as per Section 3.4) the non-critical logic if possible without incurring speed penalty, otherwise

c. Find all paths that pass though the non-critical logic and put them in the ordered list **L**. The list is ordered from longest delay to shortest delay.

d. For each path in **L** do:

i. Let P_i be the path

ii. For each pair of serially connected logic components on P_i determine the pairs that can be remapped into Joint-LUT or Joint-Slice structures (as per Section 3.3.2) and put them in set **S**

iii. Apply remapping to component pairs in **S** to liberate the space, while maintaining, or lowering, the delay of P_i

2. Move the critical logic into the liberated space

5.6.3 Increasing Packing flexibility of Flip-Flops

A recent development in logic synthesis CAD tools is the use of flip-flop control signals to reduce the amount of logic in the logic circuit. Although this approach has the benefit of reducing number of gates on the path to such flip-flop, the use of flip-flop control signals in this manner restricts the flip-flop placement flexibility, as discussed in Section 3.7. The examples of the misuse of flip-flop control signals are found in logic circuits such as batcher, vision, and trap.

The focus of this logic optimization strategy is ability to trade logic for flip-flop placement flexibility. This strategy can be implemented using the following steps:

- Move all non-critical flip-flops that have unique control signals out of congested areas of the circuit, provided this doesn't hurt performance.
- For each critical flip-flop A, find another critical flip-flop B that would benefit from sharing a slice with A in the congested area of the circuit. Extract control signals (i.e. put the control signals into a LUT as described in Section 3.7) for both flip-flops.
- Put both A and B in the same slice and repeat for other critical flip-flops
- When all critical flip-flops have been processed, move the non-critical flip-flops back into the congested areas, extracting their control signals only if they need to share a slice with a non-compatible flip-flop

```
Bin 10: 1.715ns-3.314ns, count = 16
Bin 9: 3.314ns-4.379ns, count = 323
Bin 8: 4.379ns-5.090ns, count = 386
Bin 7: 5.090ns-5.563ns, count = 444
Bin 6: 5.563ns-5.879ns, count = 238
Bin 5: 5.879ns-6.089ns, count = 210
Bin 4: 6.089ns-6.230ns, count = 92
Bin 3: 6.230ns-6.323ns, count = 42
Bin 2: 6.323ns-6.385ns, count = 10
Bin 1: 6.385ns-6.427ns, count = 5
```

Figure 5-2: Delay profile for the miim circuit

5.6.4 Stopping Criterion

A key aspect of any automated iterative algorithm is to determine when to stop the iteration. As the primary goal in this work is to improve the maximum clock frequency, it is important to inform the user when a further logic circuit optimization may be unlikely to produce significant logic circuit speed improvement. In this work the stopping criterion is based on the distribution of path delays.

Augur summarizes the path delays using a geometric delay profile, as described in section 4.3.1. The delay profile is used to determine the number of critical, and near-critical, paths as well as their location in the logic circuit. Clearly it is easier to improve the speed of a logic circuit that has just a few critical paths that do not share components or connection, rather than a logic circuit with many critical paths that share logic connection or components. In this work the optimization of the logic circuit was stopped when the following two criterion were met:

1. The two slowest bins contained 15 or more paths and
2. The paths in the two slowest bins were:
    a) Situated in close physical proximity to each other, and
    b) A logic transformation that improved the delay on all these paths could not be found

The first criterion says that there is little point in continuing if there are too many paths that must be improved to gain speed. The second notices that close-to-critical paths pose a problem if improving one of them has a strong likelihood of increasing the delay of the other close-to-critical paths by virtue of their close physical proximity.

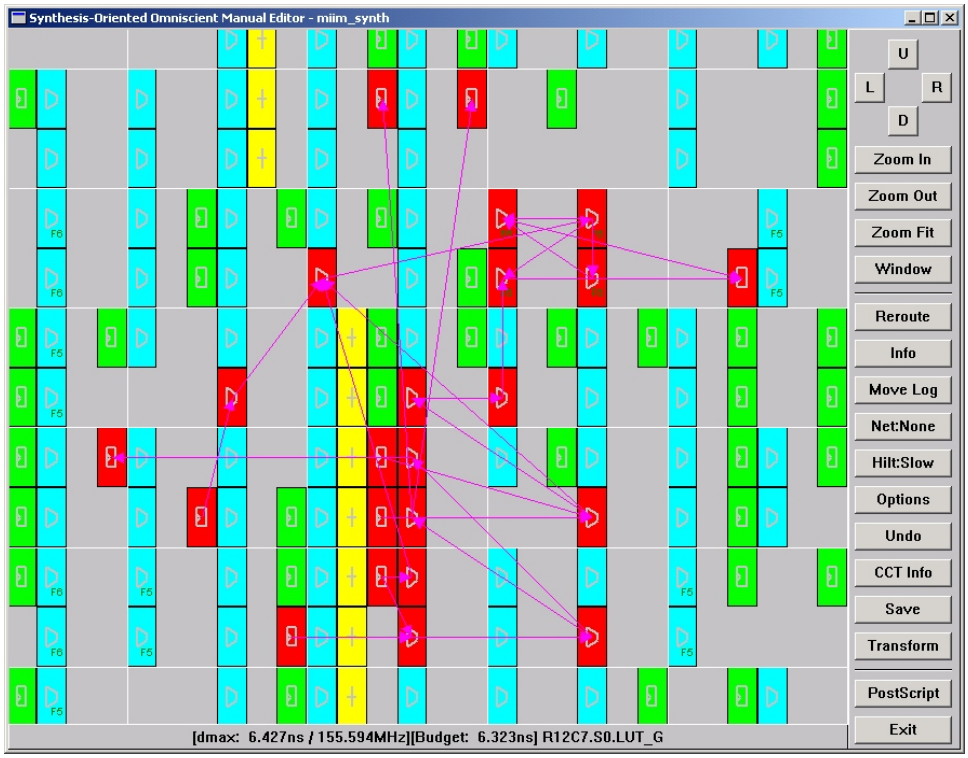

Figure 5-3: The 15 slowest paths in the miim circuit

An example of the application of this strategy is shown in Figures 5-2 and 5-3. The logic circuit miim has the 15 slowest paths as indicated by the geometric delay profile shown in Figure 5-2. The 15 slowest paths in this logic circuit are highlighted in red in Figure 5-3. These paths are in close proximity to one another. No further optimization of the circuit was performed, as none of our strategies could further improve the logic circuit in reasonable amount of time.

5.7 Summary

This chapter presented the results obtained using the manual editor Augur on a set of benchmark logic circuits. The set of 10 logic circuit was used as a benchmark. The logic circuits in the benchmark suite were synthesized, placed and routed using a rigorous method, which yielded a high logic circuit speed and therefore a fair point of reference.

Using only placement and packing modifications the average logic circuit speed was improved by 3.0 per cent. The introduction of logic synthesis transformations improved those results by another 6.7 per cent, resulting in an average speed improvement with respect to the baseline of

9.9 per cent. These results show that the application of omniscience in the context of logic synthesis yields significant logic circuit speed improvement.

In addition to the speed results, this chapter presented a set of logic circuit optimization strategies. These strategies are based on the observations made during the logic circuit improvement process that yielded the above results. The optimization strategies can be used as a base for automated algorithms in commercial CAD tools.

The results presented in this chapter show that the development of Augur was a success. However, there are still a number of things that can be improved. This topic and avenues for future research that bases on the approach presented in this thesis are the topics of the next chapter.

6 Conclusion

6.1 Thesis Summary

The goal of this research was to use the concept of omniscience in the context of physical synthesis to obtain higher speed for FPGA circuit implementation. We showed how the concept of omniscience was used to address the limitations in circuit performance through specific technology mapping transformations. Augur was developed so it could implement these transformations, allowing the user(s) to gain an insight into the way how circuits can be implemented better, and faster, on Field-Programmable Gate Arrays.

This work has three major contributions. The first contribution is the manual editor Augur, which uses omniscience in the context of physical synthesis. The second contribution is the set of logic synthesis transformations targeting the Xilinx Virtex-E devices. The final contribution is the set of optimization strategies that can be applied in an automatic FPGA CAD tool. The concept of omniscience is used throughout this thesis to evaluate the optimization choices, while the logic synthesis transformations provide the means to improve the logic circuit performance.

The results obtained in this thesis shown that the application of omniscience to logic circuit optimization can yield an average of 9.9% improvement over a suite of 10 benchmark circuits. The speed improvement obtained for each individual logic circuit varies from 0.1 to 25.5 per cent, while suffering only minor area penalty. In addition, the use of Augur has lead to the development of innovative logic synthesis transformations, such as the synchronous flip-flop control signal extraction and carry chain shortening.

6.2 Future Work

While Augur is a good manual CAD tool, there are still a number of issues that need to be addressed. Currently, the user is not informed of an estimated delay of the logic subcircuit while performing a logic transformation. This information could better assist the user in performing a logic optimization. Furthermore, the remapping transformations only result in the creation of a single LUT, Joint-LUT or a Joint-SLICE. An alternate approach would be to search for a mapping solution which consists of multiple logic structures.

An interesting topic of future research is the development of a logic block exploration software based on the approach presented in this dissertation. The following discussion describes how this approach could be applied in the context of architecture exploration.

There are several issues concerning the logic block design that need to be considered by the logic block design exploration software. They are:

1. The number of outputs available to the CLB
2. The kind of hardwired logic to include and its relation to the LUTs
3. Output sharing

This work has shown that multiple outputs for a complex logic configuration can be beneficial. In traditional CAD tools the problem of multiple-output functions is resolved by logic duplication. Thus, a multi-output function becomes a set of single output logic functions, which implement the same functionality. The mapping of each of these functions is performed independently, thus a possible sharing of resources, which arises in the context of complex logic structures, may be omitted. An alternate approach presented in this thesis shows that there is a significant benefit from taking into consideration multi-output complex logic structures, such as the Joint-LUT and the Joint-Slice.

A related topic to multiple output CLBs is the hardwired logic. In this work the hardwired logic that is considered are the multiplexers in the Virtex-E slice. This thesis showed how to map into logic structures of this type. When exploring FPGA architectures, it would be beneficial to implement a black box instead of a multiplexer. Then, a study of which basic function yields the best results on average could result in a different choice for a logic component.

Finally, there is a question of how the outputs of a CLB should be shared. Providing each

output of a CLB with a distinct pin would be very flexible. However, the area and delay penalty may be too great if all output pins are not frequently used.

References

[1] J. Lou, W. Chen and M. Pedram, "Concurrent Logic Restructuring and Placement for Timing Closure," Proc. of the 1999 ACM/IEEE Int. Conf. on CAD, November 1999.

[2] J. Lin, A. Jagannathan and J. Cong, "Placement-Driven Technology Mapping for LUT-Based FPGAs," ACM/SIGDA Int. Symp. on FPGAs, February 2003, Monterey, California, USA.

[3] D. P. Singh and S. D. Brown, "Incremental Placement for Layout-Driven Optimizations on FPGAs," Proc. of the 2002 ACM/IEEE Int. Conf. on CAD, November 2002.

[4] J. Cong and Y. Ding, "FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," IEEE Trans. on CAD of Integrated Circuits and Systems, vol. 13, issue 1, 1994, pp. 1-12.

[5] Online resource: http://toolbox.xilinx.com/docsan/xilinx4/data/docs/lib/dsgnelff47.html #226728, Last accessed: November 22nd, 2003.

[6] W. Chow and J. Rose, "EVE: A CAD Tool for Manual Placement and Pipelining Assistance of FPGA Circuits," Proc. of ACM/SIGDA Int. Symp. on FPGAs, February 2002, Monterey, California, USA.

[7] The Virtex-E device is manufactured by Xilinx Corporation. Online resource: http://direct.xilinx.com/bvdocs/publications/ds022.pdf, accessed on July 7, 2003.

[8] A. Roopchansingh and J. Rose, "Nearest Neighbour Interconnect in Deep Submicron FPGAs", Proc. of IEEE CICC, May 2002, pp. 59-62.

[9] S. Devadas, A. Ghosh and K. Keutzer, **Logic Synthesis**, McGraw-Hill Inc., 1994, ISBN 0-07-016500-9

[10] A. Lu, H. Eisenmann, G. Stenz, and F. M. Johannes, "Combining Technology Mapping with Post-Placement Resynthesis for Performance Optimization," Proc. of Int. Conf. on Computer Design: VLSI in Computers and Processors, October 1998, pp. 616-621.

[11] R. McCready and J. Rose, "Real-Time Face Detection on Configurable Hardware System," Tenth International Workshop on Field Programmable Logic, 2000, pp.157-162.

[12] S. Chang, M. Marek-Sadowska, T. Hwang, "Technology Mapping for TLU FPGA's based on Decomposition of Binary Decision Diagrams," IEEE Trans. On CAD of Integrated Circuits and Systems, Vol. 5, No. 10, pp. , October 1996.

[13] K. Schabas and S. D. Brown, “Logic Synthesis and mapping: Using logic duplication to improve performance in FPGAs,” Proc. of ACM/SIGDA Int. Symp. on FPGAs, February 2003, Monterey, California, USA

[14] Synplify 7.1 Pro is a product developed by Synplicity Incorporated. Online product link is: http://www.synplicity.com/products/synplifypro/index.html

[15] Xilinx ISE 5.1, Service Pack 3, is a product of Xilinx Corporation. Online product link is: http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=ISE+Foundation

[16] P. Bade, W. Chow, P. Kundarewich, N. Saniei, and A. Wong, “StarBurst ATM Chip project at the University of Toronto,” October 2000.

[17] J. Fender and J. Rose, “A High Speed Ray Tracing Engine Built on a Field programmable System,” submitted to the ACM/SIGDA International Conference on FPGAs, 2004.

[18] D. Galloway, M. Van Ierssel, J. Rose, P. Chow, “Transmogrifier-3 project,” http://www.eecg.toronto.edu/~tm3, 2000.

[19] XC4000 is an FPGA Device manufactured by Xilinx Corporation. Online resource: http://www.xilinx.com/bvdocs/publications/4000.pdf

[20] S. Hauck, M. Hosler, and T.Fry, “High-Performance Carry Chains for FPGAs,” Proc. of the ACM/SIGDA Int. Symp. on FPGAs, Monterey, California, February 1998, pp. 223-233.

[21] K. Chen, J. Cong, Y. Ding, A. Kahng and P. Trajmar, “DAG-Map: Graph Based FPGA Technology Mapping For Delay Optimization,” IEEE Design and Test, September 1992, pp.7-20.

[22] H. Yang and D. F. Wong, “Edge-Map: Optimal Performance Driven Technology Mapping for Iterative LUT Based FPGA Designs,” 1994, pp. 150-155.

[23] J. P. Roth and R. M. Karp, “Minimization over boolean graphs,” IBM Journal, April 1962, pp 227-238.

[24] E. Lehman, Y. Watanabe, J. Grodstein, H. Harkness, “Logic Decomposition During Technology Mapping,”IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol 16, No. 8, August 1997, pp. 813-833.

[25] W. Chow, “EVE: A CAD Tool Providing Placement and Pipelining Assistance for High-Speed FPGA Circuit Designs,” Master of Applied Science Thesis, University of Toronto,

2001.

[26] J. Cong and Y. Hwang, "Boolean Matching for LUT-Based Logic Blocks With Applications to Architecture Evaluation and Technology Mapping," IEEE Trans. on CAD of Integrated Circuits and Systems, Vol. 20, No. 9, September 2001, pp. 1077-1090.

[27] Douglas B. West, **Introduction to Graph Theory**, Second Edition, Prentice-Hall Inc., 2001, ISBN 0-13-014400-2.

[28] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli, "Sequential circuit design using synthesis and optimization," IEEE International Conference on Computer Design, 1992, pp. 328-333.

[29] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," Seventh International Workshop on Field Programmable Logic, September 1997, pp. 213-222.

A Appendix

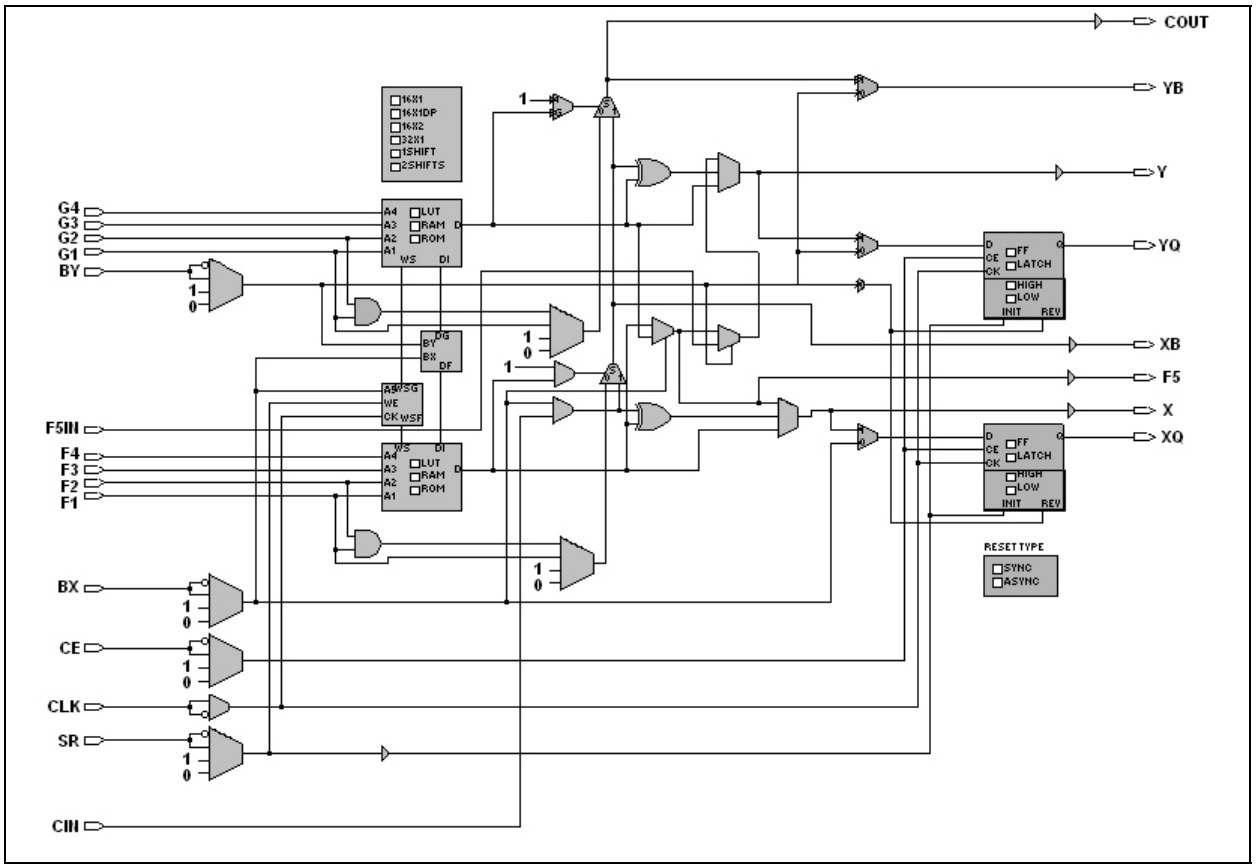

Figure A-1: Detailed Xilinx Virtex-E Slice Schematic