

**EVE: A CAD Tool Providing
Placement and Pipelining Assistance for
High-Speed FPGA Circuit Designs**

by

William Chow

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

© Copyright by William Chow 2001

EVE: A CAD Tool Providing Placement and Pipelining Assistance for High-Speed FPGA Circuit Designs

William Chow

Master of Applied Science, 2001

Edward S. Rogers Sr. Department of Electrical and Computer Engineering,
University of Toronto

Abstract

As FPGAs push ever deeper into mainstream digital design, there is an increasing desire for high-performance circuits. This thesis describes a manual editor called EVE that can assist a designer to perform manual packing, placement and pipelining of commercial FPGA circuits to achieve a meaningful increase in performance. This effort is inspired by Von Herzen's paper [VonH97] [VonH97a], which proposed the notion of an "Event Horizon" – a high-speed circuit design approach in which complete knowledge of the timing effect of every synthesis change is used. It is very laborious to implement circuits using this approach; therefore we try to augment manual design tools in order to make this Event Horizon methodology easier to perform. This thesis describes a first step in that direction, which focuses on placement, packing and pipelining. EVE provides an interactive environment that immediately reroutes and timing analyzes after each user circuit modification, giving an exact value for critical path delay. It can also suggest good placement positions and provide flip-flop insertion assist during pipelining. Experimental results show that EVE can achieve up to 18.8% and on average 12.7% speed increase on a set of eight Xilinx Virtex-E circuits of 250 or fewer LUTs.

Acknowledgements

I would like to take this opportunity to thank my thesis supervisor, Professor Jonathan Rose, for his guidance, advice, and encouragement throughout the course of my studies.

I am also indebted to Dr. Kevin Chung at Xilinx for his timely and helpful advice on the use of Xilinx backend tools.

I would also like to thank all of my friends who have made graduate school enjoyable. I especially thank Bryan Chan, Andy Ye, Vincent Gaudet, Warren Gross, Paul Kundarewich, Ajay Roopchansingh, Derek So, Ted Fill, Kostas Pagiamtzis, and Elias Ahmed, for their friendship, encouragement, and technical advice.

I would like to thank Natural Sciences and Engineering Research Council of Canada, MICRONET and Xilinx Corporation for providing me with research funding.

I wish to express my sincerest gratitude to my parents who have always supported my studies.

Lastly, I would like to thank Amy for her endurance, understanding and love throughout the years.

Contents

CHAPTER 1	INTRODUCTION.....	1
1.1	MOTIVATION.....	1
1.2	OBJECTIVES	3
1.3	THESIS ORGANIZATION.....	4
CHAPTER 2	BACKGROUND	5
2.1	XILINX VIRTEX-E ARCHITECTURE.....	6
2.1.1	<i>Virtex-E CLB architecture</i>	6
2.1.2	<i>Virtex-E Routing Architecture</i>	7
2.2	EVENT HORIZON	9
2.3	PREVIOUS RESEARCH AND DEVELOPMENT ON MANUAL DESIGN TOOLS	11
2.3.1	<i>Electric</i>	12
2.3.2	<i>Magic</i>	14
2.4	XILINX BACKEND TOOLS.....	15
2.4.1	<i>Xilinx Hardware Design Flow</i>	15
2.4.2	<i>Xilinx Floorplanner</i>	17
2.4.3	<i>Xilinx FPGA Editor</i>	19
CHAPTER 3	EVE	23
3.1	DESIGN OBJECTIVES	23
3.2	TIMING EXACT MICROSCOPIC PLACEMENT (TEMP) MODE	25
3.2.1	<i>Timing Horizon</i>	25
3.2.2	<i>Features of TEMP mode</i>	27
3.3	PIPELINING MODE.....	31
3.3.1	<i>Graphical Circuit Representation</i>	33
3.3.2	<i>Flip-flop Insertability</i>	35
3.3.3	<i>Loop Elimination</i>	35
3.3.4	<i>Flip-flop Insertion</i>	36
3.3.5	<i>Flip-flop Motion</i>	37
3.3.6	<i>Flip-flop Tracing</i>	38

3.3.7	<i>Inserted Flip-flop Synthesis and Placement</i>	38
3.4	LIMITATIONS.....	39
CHAPTER 4	IMPLEMENTATION	41
4.0	SOFTWARE ARCHITECTURE OF EVE	42
4.1	GUI	43
4.1.1	<i>EasyGL for Windows</i>	43
4.1.2	<i>Generating Graph Layout for Pipelining Mode</i>	43
4.2	BACKEND INTEGRATION	44
4.2.1	<i>Insufficiency of existing tools</i>	44
4.2.2	<i>Solution: Interfacing with FPGA Editor</i>	45
4.2.3	<i>EVE: knowing circuit timing at all times</i>	46
4.3	DELAY EXTRACTION AND CACHING	47
4.3.1	<i>Delay Extraction</i>	48
4.3.2	<i>Extracting Logic Delays</i>	48
4.3.3	<i>Extracting Routing Delays</i>	49
4.3.4	<i>Delay Database Compression</i>	51
4.3.5	<i>Generating Compression Scheme</i>	55
4.3.6	<i>Routing Delay Retrieval</i>	56
4.4	PARTIAL INCREMENTAL TIMING ANALYSIS	58
4.4.1	<i>Constructing Timing Graph</i>	58
4.4.2	<i>Timing Analysis</i>	59
4.5	HORIZON CALCULATION	60
4.6	PIPELINING.....	61
4.6.1	<i>Loop Detection</i>	63
4.6.2	<i>Flip-flop Insertion</i>	65
4.6.3	<i>Flip-flop Motion</i>	67
4.6.4	<i>Placement of Inserted Flip-flops</i>	68
4.6.5	<i>Period Estimation</i>	69
4.6.6	<i>Flip-flop Initial State Calculation</i>	71
4.7	DESIGN RULE CHECK (DRC).....	72
4.8	VALIDATION	72
CHAPTER 5	EXPERIMENTAL RESULTS.....	75
5.1	BENCHMARK CIRCUIT DESCRIPTIONS	76
5.1.1	<i>Vision</i>	76
5.1.2	<i>Batcher</i>	77
5.1.3	<i>Banyan</i>	78
5.1.4	<i>Trap</i>	79
5.1.5	<i>Miim</i>	80
5.1.6	<i>Div</i>	80
5.1.7	<i>Dotproduct</i>	81
5.1.8	<i>Crossproduct</i>	81
5.2	BASELINE CIRCUITS GENERATION	82
5.3	RESULTS: USING TEMP MODE ONLY	84
5.4	RESULTS: USING BOTH TEMP AND PIPELINING MODES	90

CHAPTER 6	CONCLUSION AND FUTURE WORK	93
6.1	CONCLUSION.....	93
6.2	FUTURE WORK	94
REFERENCES.....		97
APPENDIX A	SCREEN CAPTURES OF EXPERIMENTAL CIRCUITS	103

List of Tables

TABLE 4-1: PSEUDOCODE FOR RETRIEVING ROUTING DELAY	57
TABLE 4-2: PSEUDOCODE FOR FINDING LOOPS.....	65
TABLE 4-3: INSERTING FLIP-FLOPS.....	66
TABLE 5-1: OPTIONS USED FOR BASELINE CIRCUITS GENERATION	84
TABLE 5-2: RESULTS FOR USING THE TEMP MODE.....	85
TABLE 5-3: RESULTS FOR USING BOTH TEMP AND PIPELINING MODES	91

List of Figures

FIGURE 1-1: TRADITIONAL PUSH-BUTTON CAD FLOW VS. EVENT HORIZON METHODOLOGY	2
FIGURE 2-1: A VIRTEX-E CLB (TAKEN FROM[XILI2001B])	7
FIGURE 2-2: ROUTING RESOURCES OF A VIRTEX-E CLB (TAKEN FROM [XILI2001B])	8
FIGURE 2-3: EVENT HORIZON	9
FIGURE 2-4: EXTENDING THE EVENT HORIZON USING PIPELINING	11
FIGURE 2-5: XILINX HDL BASED HARDWARE DESIGN FLOW	16
FIGURE 2-6: SCREEN CAPTURE OF THE XILINX FLOORPLANNER	18
FIGURE 2-7: SCREEN CAPTURE OF THE XILINX FPGA EDITOR	20
FIGURE 3-1: TIMING HORIZON	26
FIGURE 3-2: SCREEN CAPTURE OF THE TEMP MODE SHOWING A “HORIZON”	29
FIGURE 3-3: SCREEN CAPTURE OF THE PIPELINING MODE	34
FIGURE 3-4: LOOP COLLAPSING	36
FIGURE 3-5: A TIMING GRAPH USED IN THE PIPELINING MODE	37
FIGURE 4-1: ARCHITECTURE OF EVE	42
FIGURE 4-2: INTERACTION BETWEEN EVE AND THE BACKEND	47
FIGURE 4-3: CONSTRUCTING DELAY MATCHING TABLE	49
FIGURE 4-4: # OF PIN-TO-PIN CONNECTIONS VS. MANHATTAN DISTANCE	51
FIGURE 4-5: ROUTING DELAY PROFILE FOR GROUP G	52
FIGURE 4-6: ROUTING DELAY GROUP G	54
FIGURE 4-7: DELAY MATCHING	59
FIGURE 4-8: FORWARD AND BACKWARD RETIMING	61
FIGURE 4-9: TRANSFORMING FLIP-FLOPS WITH SR AND CE	62
FIGURE 4-10: FINDING EDGE IN LOOP	64
FIGURE 4-11: PERIOD ESTIMATION	71
FIGURE 5-1: STRUCTURE OF THE VISION CIRCUIT (TAKEN FROM [MR2000])	77
FIGURE 5-2: STRUCTURE OF A BATCHER ELEMENT	78
FIGURE 5-3: STRUCTURE OF A BANYAN ELEMENT	79
FIGURE 5-4: STRUCTURE OF A TRAP ELEMENT	80
FIGURE 5-5: STRUCTURE OF THE DOTPRODUCT CIRCUIT	81

Chapter 1

Introduction

1.1 Motivation

Most FPGA circuits are designed using a traditional “push-button” CAD flow, which involves design entry, logic synthesis, architectural mapping, floorplanning, placement and routing, followed by timing analysis. When a high circuit speed that pushes the limits of the silicon’s capability is desired, this approach often fails to achieve the required performance. Designers will typically repeatedly floorplan, place and route the circuit until the design goal is met. This iterative process is very time consuming because the resulting design speed is not known until after timing analysis is performed, and the result may seem to be decoupled from the changes applied. There is a clear need for a different high-speed circuit design methodology.

In [VonH97] [VonH97a], Von Herzen described the design of a signal processing circuit in FPGA running at 250MHz in 1997. He demonstrated a high-speed circuit design methodology using the notion of an “Event Horizon”. The “Event Horizon” refers to the boundary that a circuit element can be placed within in order to satisfy a timing budget. This methodology demands that the designer create each microscopic piece of the circuit with the timing budget in mind. During this process, the complete routing delays are included in the time accounting. Von Herzen used low-level manual design tools to select routing resources carefully, and to avoid the placement of logic elements outside of the “Event Horizon”. However, it is very laborious to implement circuits using the low-level manual design tools. We therefore became interested to augment such tools to facilitate circuit design employing the Event Horizon methodology. Our editor is called EVE, which stands for EVent horizon Editor.

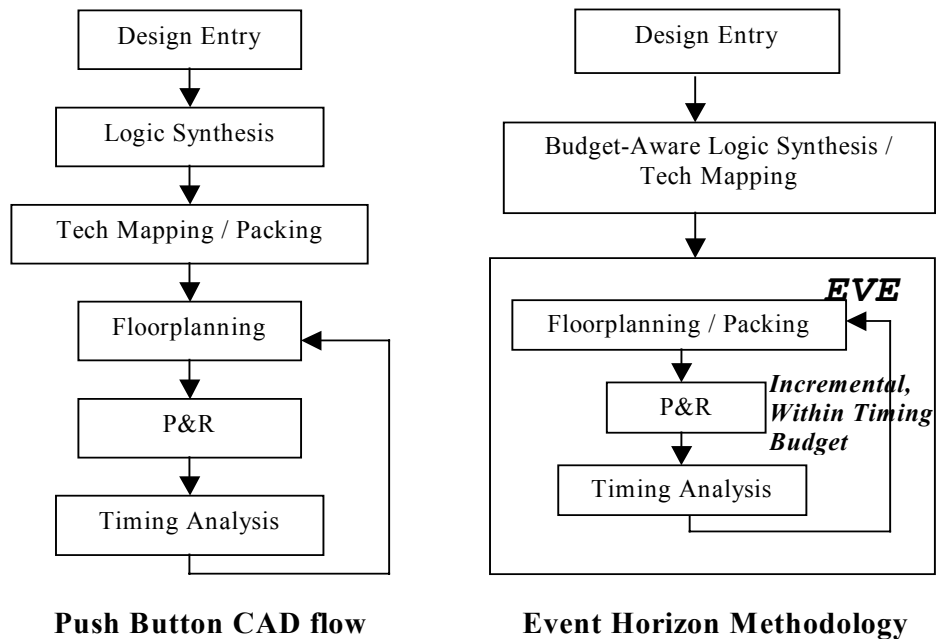


Figure 1-1: Traditional Push-Button CAD flow vs. Event Horizon Methodology

Figure 1-1 illustrates the difference between the Event Horizon methodology and the traditional push-button design flow. In the push-button design flow, the whole circuit is optimized iteratively in three separate passes including floorplanning, placement, and routing. Each pass focuses on solving its own problem, so it has limited information on how the decisions made in the current pass will affect the results in later passes downstream. As a result, timing closure is poor and many iterations are needed to obtain a design that satisfies the timing goal. The Event Horizon methodology, in contrast, focuses on constructing one small part of the circuit at a time, and by performing packing, placement and routing simultaneously for small individual pieces of the circuit, timing closure is improved. EVE is designed to facilitate high-speed circuit design following this Event Horizon methodology. We will now discuss the objectives of this work.

1.2 Objectives

This thesis has two objectives. Firstly, we would like to construct a manual editor that augments the current low-level floorplanning and circuit editing tools, by following the Event Horizon design methodology. Secondly, we would like to gain more insights to better placement and routing techniques by extensively using the tool to augment the speed of real designs.

1.3 Thesis Organization

The remainder of this thesis is structured as follows: In Chapter 2, we discuss background relevant to this work. We will first provide an overview of the Xilinx Virtex-E [Xili2001b] architecture, and then describe the concept of “Event Horizon” in more detail. Finally, we will look at some pioneering work on manual design tools research, and describe the various Xilinx backend tools. Chapter 3 describes the features of EVE, and illustrates how the user can use EVE to design high-speed circuits. Chapter 4 describes the implementation of these features. In Chapter 5, experimental results of the use of the tool are presented. Finally, we provide concluding remarks and discuss about possible future work in Chapter 6.

Chapter 2

Background

In this chapter, we give a review of relevant background material, and previous related research. First, we provide an overview of the Xilinx Virtex-E [Xili2001b] FPGA architecture in Section 2.1, which is the device targeted in the research. Then we will describe the Event Horizon concept as proposed by Von Herzen [VonH97] [VonH97a] in Section 2.2. We will also describe previous work on manual design tools, including the Electric [Rubi83] and Magic [OHM84] editors, in Section 2.3. Finally, we will describe in Section 2.4 the Xilinx hardware design CAD flow, and existing Xilinx manual editing tools such as the Xilinx Floorplanner [Xili2001] and FPGA Editor [Xili2001a].

2.1 Xilinx Virtex-E Architecture

2.1.1 Virtex-E CLB architecture

The Xilinx Virtex-E [Xili2001b] FPGA is built in a $0.18\mu\text{m}$ CMOS technology. It is an island-style [BRM99] FPGA architecture in which routing resources surround Configurable Logic Blocks (CLBs). Each CLB has two slices and each slice has two 4-input look up tables (4-LUTs) and two flip-flops (FFs). The two 4-LUTs in each slice can be combined to form a 5-LUT and two such 5-LUTs in the same CLB can be combined to form a 6-LUT. There are fast carry chains that run vertically upwards in each slice. Each slice also contains dedicated AND gates and XOR gates for fast addition and multiplication. A 4-LUT can also be configured as RAM, ROM, or a 1-bit Shift Register LUT (SRL) of variable depth ($1-16$).

The fast built-in arithmetic and carry logic, as well as the flexibility of each CLB makes it an extremely powerful building block for configurable logic. Complex logic structures can be densely packed, and will only occupy a minimal number of CLBs. The Virtex-E CLBs together with the concept of Relational Placed Modules (RPMs), which define relative placement of each circuit element in a design module, enable the development of high-performance, area-optimized IP core libraries. Figure 2-1, taken from [Xili2001b], shows a Virtex-E CLB.

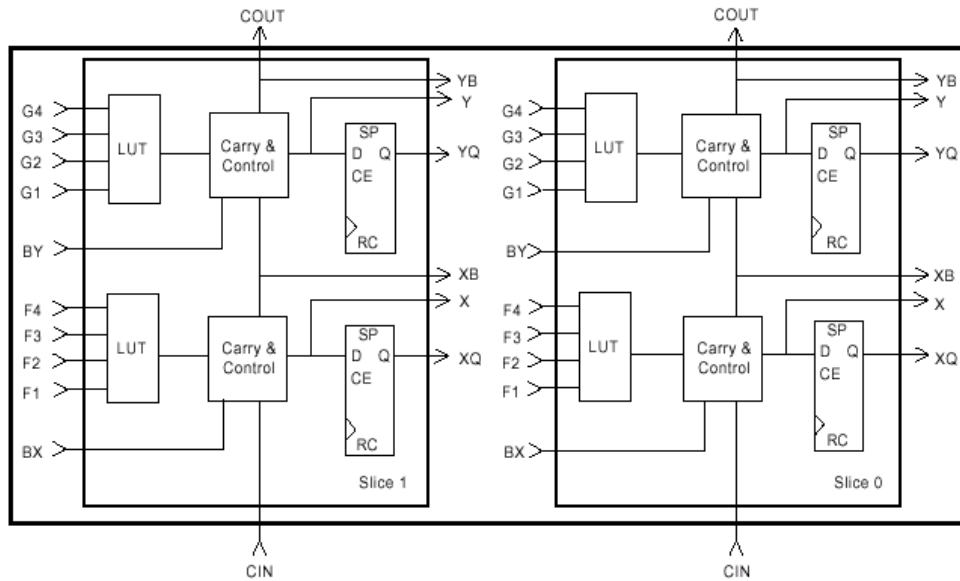


Figure 2-1: A Virtex-E CLB (taken from[Xili2001b])

2.1.2 Virtex-E Routing Architecture

Each Virtex-E CLB has nearest neighbour inputs connecting to each of the left and right neighbouring CLBs and has access to global signals. These nearest neighbour connections, unlike other regular routing resources, connect directly with the neighbouring CLB, without passing through any switch boxes. They are important for building high-speed circuits because of their low routing delays. The CLB has ample routing resources in the General Routing Matrix (GRM) consisting of singles (lines of length one), hexes (lines of length six), and long lines (spanning across the width and the height of the chip) connecting in all four directions of the FPGA array.

The Virtex-E routing architecture is highly symmetrical. Routing delays can be estimated easily based on the distance between the end-points of a route in the FPGA.

Figure 2-2 illustrates the Virtex-E routing resources.

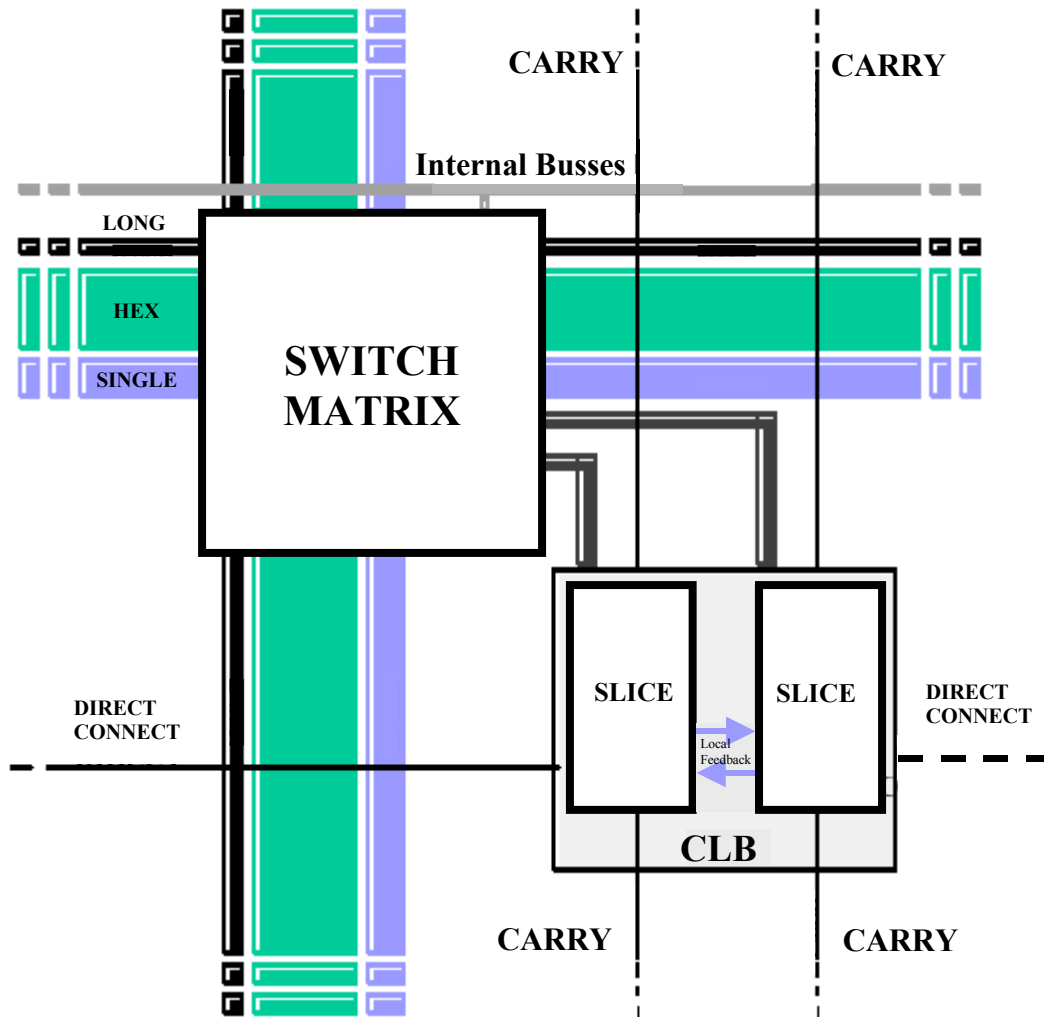


Figure 2-2: Routing Resources of a Virtex-E CLB (taken from [Xili2001b])

2.2 Event Horizon

In [VonH97] [VonH97a], Von Herzen described the design of a signal processing circuit running at 250MHz in an FPGA using $0.6\mu\text{m}$ technology in 1997! This was a speed far beyond the otherwise typical use of that required of the silicon. He did this by using low-level manual design tools to configure each logic slice of the circuit and to select the routing resources manually. In order to achieve a high-speed goal, he had to be very carefully in choosing where to place each circuit element physically on the chip, because routing delays made up a significant portion of the critical delay of the circuit. He proposed the concept of an “Event Horizon”, which could be used to quickly estimate where circuit elements could be placed without violating a very tight timing budget.

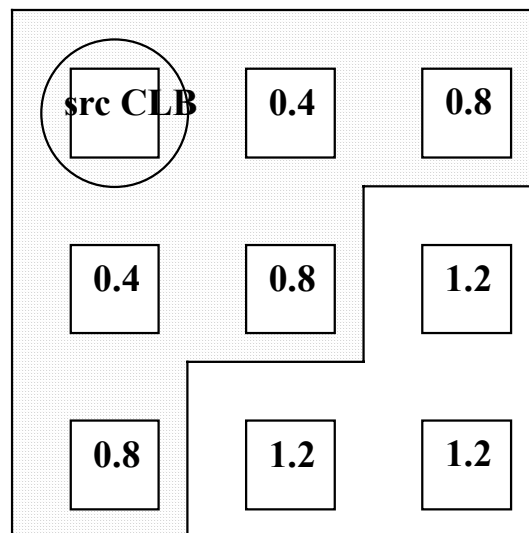


Figure 2-3: Event Horizon

The “Event Horizon” concept is illustrated in Figure 2-3. Assume that a circuit needs to run at 250MHz , so the critical path time budget is 4.0ns . A flip-flop (FF) in the

source configurable logic block (CLB) (marked with a circle in the figure) drives a LUT-to-FF combination in another CLB. To determine how far away the target CLB can be placed, we first need to obtain some timing characteristics about the chip such as the maximum clock skew, LUT delay, routing delays, as well as the clock-to-output delay and FF setup time. Assume that the maximum clock skew is $0.1ns$, the clock-to-output delay is $1.3ns$, and that LUT delay + FF setup time through is $1.5ns$, routing can then take at most $4.0 - 0.1 - 1.3 - 1.5 = 1.1ns$. Assume that the FPGA has a uniform routing architecture that requires $0.4ns$ to travel the distance of one CLB (in Manhattan distance) in all directions. For the example given in Figure 2-3, the target CLB can only be placed at locations that can support a routing delay of at most $0.8ns$, as indicated by the shaded box in Figure 2-3. Von Herzen called such a boundary box the Event Horizon of the source CLB in the context of a circuit required to run at 250MHz. The “Event Horizon” is then defined as the boundary within which the target CLB can lie, such that the routing delay to reach the target CLB from the source CLB is small enough to satisfy the timing budget.

With a timing goal in mind, Von Herzen then calculated the Event Horizon for each circuit element, and tried to place the connecting elements in its Event Horizon. In cases when this was not possible (for example, all locations in the Event Horizon were already occupied), extra flip-flops could be introduced to pipeline the circuit, permitting the target CLB to move outside of the current Event Horizon, as illustrated in Figure 2-4.

By first calculating the Event Horizon of each circuit element at a given target speed, Von Herzen incrementally built each part of the circuit, knowing that the timing budget would be met throughout the design process.

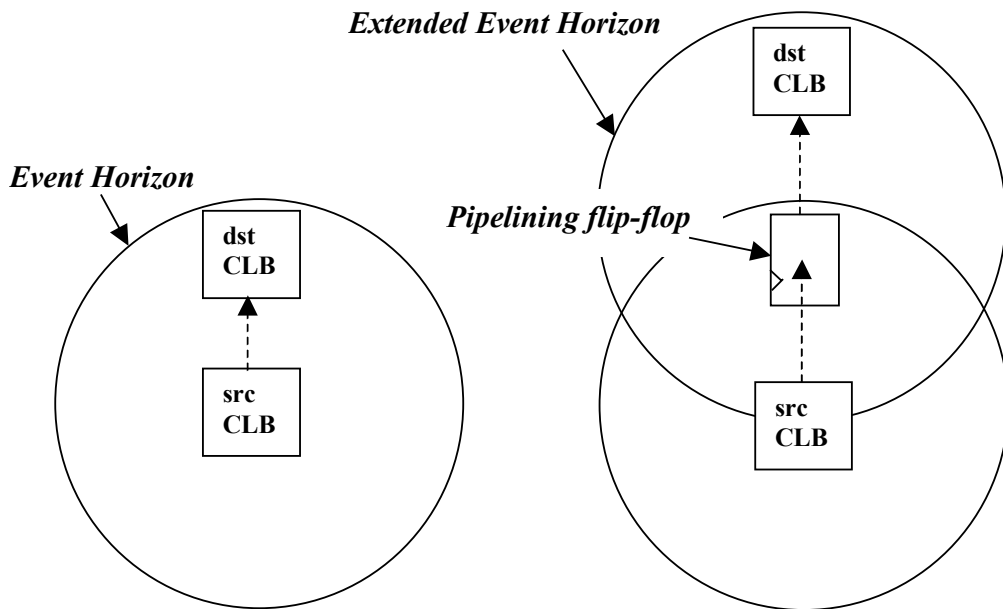


Figure 2-4: Extending the Event Horizon using Pipelining

2.3 Previous Research and Development on Manual Design Tools

In this Section, we will discuss some previous research and development on manual design tools. We will look at two representative tools: Electric [Rubi83] and Magic [OHM84]. Both tools provide powerful graphical interfaces to represent circuits

intuitively, and have various useful features to simplify circuit design tasks. They can also integrate with multiple backend tools, augmenting the power of the whole circuit design environment. Both Electric and Magic demonstrated the importance of intuitive graphical interface designs and the ability to integrate other tools. They provide valuable insights into the design of EVE.

2.3.1 Electric

Electric [Rubi83] is a major milestone in manual design tools research. It is an electrical circuit design environment developed in 1982 that combines a user-friendly GUI interface, the ability to support multiple circuit design technologies, a hierarchical circuit database that stores circuit elements in a top-down circuit layout, and a set of powerful analysis tools such as a design rule checker (DRC), and circuit simulators. At the time when Electric was first developed, a large amount of electric design tools already existed, each with its own user interface and ways to represent circuits. Electric was built to integrate multiple design tools together, providing a uniform and hierarchical graphical circuit design system.

The original version of Electric supported multiple technologies including nMOS, CMOS, Bipolar, and printed circuit board designs. Each technology has its associated component library and design rules. The component libraries in turn describe the graphic attributes and simulation behaviour of each component. The design environment maintains the hierarchy and connectivity of multiple components of mixed technologies.

The overall circuit can be easily integrated, and can be analyzed and simulated using external analysis tools.

The circuit database stores the circuit using a network of nodes and arcs. Nodes represent the circuit elements and arcs represent the connecting wires. Each component and wire has geometric attributes to enforce correct layout. Due to the component-based and hierarchical nature of the database, modification to individual components will be reflected throughout the whole circuit across hierarchical boundaries.

The Electric editor is very flexible. The user can develop new layout rules that will be enforced during the course of the design. Additional analysis tools can also be adopted into the environment by writing interfacing scripts. Electric supports simultaneous background processing of multiple backend tools.

To date, Electric has been extended to support custom IC layout, schematic drawing, hardware description language specifications, and electro-mechanical hybrid layout. It can perform various CAD operations such as design rule checking, electrical rule checking, simulation, routing, HDL compilation, and network consistency checking (LVS).

2.3.2 Magic

Magic [OHM84] is an interactive layout editor for full custom VLSI designs developed in 1984. It resembles Electric closely in terms of its flexibility to support different chip process technologies, its ability to interface with external simulation and analysis tools, and its hierarchical circuit design capabilities. Magic also has its own routing tool and a hierarchical circuit extractor.

Magic accepts a detailed description of an IC processing technology known as a technology file. The technology file describes the graphic and conducting attributes of each layer such as metals, contacts, vias, polysilicon, and diffusions. It also describes the design and routing rules, and how transistors are formed. With a technology file defined, Magic can examine a circuit and extract transistors of various sizes and determine connectivity of circuit elements. The built-in hierarchical circuit extractor can then produce circuits to be simulated in SPICE [Nage75], RSIM [Term83], or IRSIM [SH89]. Instead of modeling the process technology exactly, the technology file contains simplified design rules and circuit structures. Such simplifications make the circuit design process easier and allow Magic to provide powerful design aids.

Using the design rules defined in the technology file, Magic continuously checks for design rule violations and displays error markers to inform the user of design problems. Continuous design-rule checking means that the user will be able to fix problems quickly and incrementally, thus eliminating the need to run long design rule checks over the whole design. When a small change is applied to an existing layout, the

user will immediately find out if errors are introduced, without performing design checks on the whole layout.

2.4 Xilinx Backend Tools

In Section 2.4.1, we will describe the Xilinx hardware design flow, which uses a hardware description language (HDL) for design entry. We will briefly describe various Xilinx backend tools required in the flow. Then in Section 2.4.2 and Section 2.4.3, we will describe about the Xilinx Floorplanner and FPGA Editor, which our work is based on. EVE has a graphical interface similar to that of the Xilinx Floorplanner, and it integrates tightly with the Xilinx FPGA Editor for its backend support, performing the actual circuit editing, placement, and routing operations on EVE's behalf.

2.4.1 Xilinx Hardware Design Flow

The Xilinx hardware description language (HDL) design flow, illustrated in Figure 2-5, begins with design entry using an HDL language such as VHDL or Verilog. Then a logic synthesizer such as Synplify Pro [Synp2000] and FPGA Express [Syno2000] will synthesize the design into a netlist of logic gates in the EDIF [SM89] format. Timing constraints are also generated to drive the subsequent stages of the design flow. Then a Xilinx tool called NGDBuild converts the netlist into its native circuit description database format (NGD). Then the technology mapper (MAP) will pack the logic gates into Configurable Logic Blocks (CLBs) and I/O Blocks (IOBs) compatible

with the target chip technology. The resulting file (NCD) can then be placed and routed using the Xilinx Place and Route (PAR) tool. It can also be floorplanned using the Xilinx Floorplanner, or manually edited using the Xilinx FPGA Editor. TRACE is a timing analyzer that takes an NCD file as input. The Xilinx Description Language (XDL) program can convert between the Xilinx proprietary NCD format to a text based XDL circuit description file.

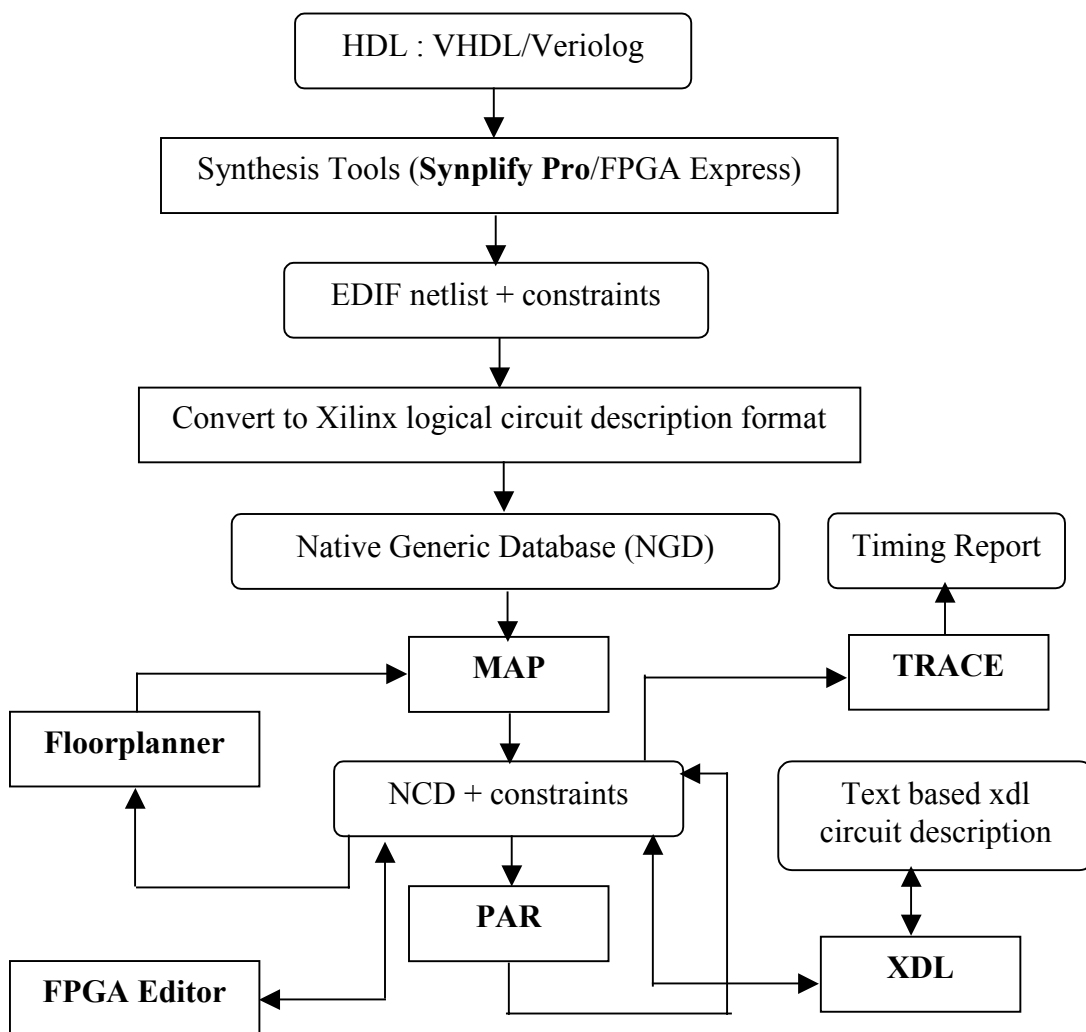


Figure 2-5: Xilinx HDL based hardware design flow

Synplify Pro is an advanced synthesis tool that provides useful features such as resource sharing, automatic pipelining and register balancing of the circuit. It has extensive knowledge of the target Xilinx chip architectures so it can produce a highly optimized netlist for the Xilinx backend tools. The Xilinx hardware design flow is timing-driven. With timing constraints set, each stage of the design flow will attempt to optimize the circuit to meet the constraints.

2.4.2 Xilinx Floorplanner

The Xilinx Floorplanner [Xili2001] is a graphical placement tool that controls where circuit elements get placed in the FPGA. It allows the user to describe coarse constraints on where portions of the circuit are positioned by the automated placement and routing software (PAR). The floorplanner has two windows, one showing the current placement information of the circuit on the FPGA, and the other showing a floorplan, which the user can modify. Each window displays the FPGA as a grid of CLBs, with each CLB showing its associated circuit elements including flip-flops, look-up tables, carry elements, and tri-state buffers. The user employs a drag-and-drop paradigm to move logical grouping of elements onto the floorplan (initial floorplan) or modify the floorplan of a previously placed and routed design by dragging circuit elements around the floorplan. The floorplanner makes use of logical information of the circuit to present all circuit elements in a hierarchical grouping. For example, circuit elements that belong to the same carry chain will be grouped together, so the user can floorplan the whole carry

chain as a group. Ungrouping and regrouping of circuit elements are also allowed. Figure 2-6 shows a screen capture of the Xilinx Floorplanner.

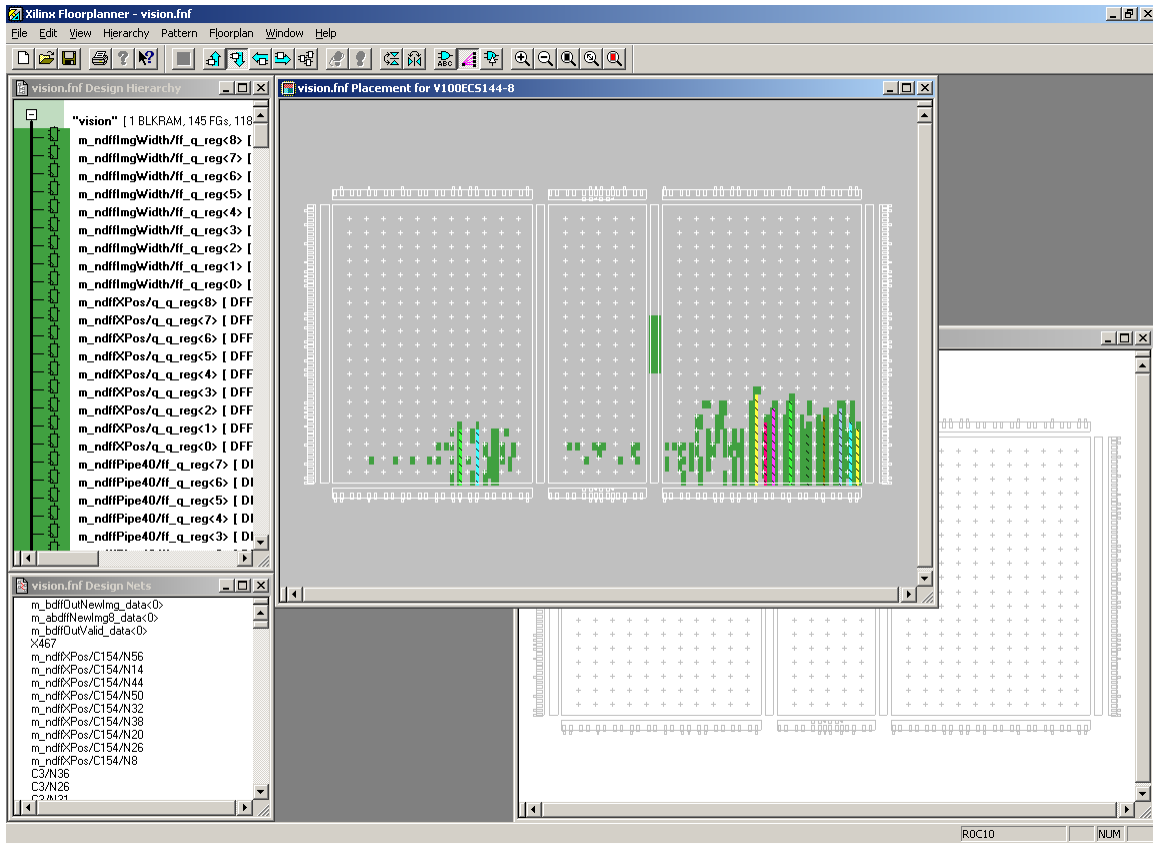


Figure 2-6: Screen Capture of the Xilinx Floorplanner

When the Xilinx Floorplanner is loaded with a placed and routed design, it can display a routing congestion map, which colours each CLB with a colour indicating the severity of routing congestion. CLBs with high routing congestion are marked in red. Upon detecting a routing congestion problem, the designer can then re-floorplan the problematic area to relieve the routing congestion. One useful feature that the Xilinx Floorplanner does not have is the ability to estimate the resulting circuit speed during floorplanning. Such instant feedback of timing information will greatly help the designer

to decide on packing/unpacking and placement operations early on in the design flow, and is one of the key feature of the present work.

2.4.3 Xilinx FPGA Editor

The Xilinx FPGA Editor [Xili2001a] is a graphical tool for manually editing FPGA circuits at the slice and routing resource configuration level. The user has complete control over each configuration bit of the circuit, including the choice of routing wires to use for each route. It displays the layout of the chip graphically as a grid of CLBs. When a design is loaded, the user will see a graphical picture showing how logic slices are configured and how each net is routed.

The FPGA Editor is able to invoke automatic placement, routing, design rule checking (DRC), and timing analysis. It also has a complete set of textual commands that the user can call to perform various editing operations. Such commands can be run from pre-written script files, or from another external program through the use of named pipes on UNIX or Windows NT platforms. To date, the ability to finely control the internal configurations and routing wire usage are also available in tools such as JBits [GLS99] and JHDL [BH98], which are Java-based programming APIs, allowing the designer to configure Xilinx FPGAs using Java programs.

The FPGA Editor is useful for completing placement and routing operations when routing cannot be completed automatically, or when the timing budget cannot be met

using an automatic approach. By examining the routing wires in the neighbourhood of a timing-critical or unroutable net, the designer can then manually reassign routing resources to complete the routing, or re-place certain logic slices to satisfy a timing budget. The FPGA Editor can also set up logic probes to expose internal states to external I/O pins of a physical chip being debugged. It also serves as a great learning tool that shows in detail each internal block of the chip, making understanding various Xilinx FPGA architectures easier.

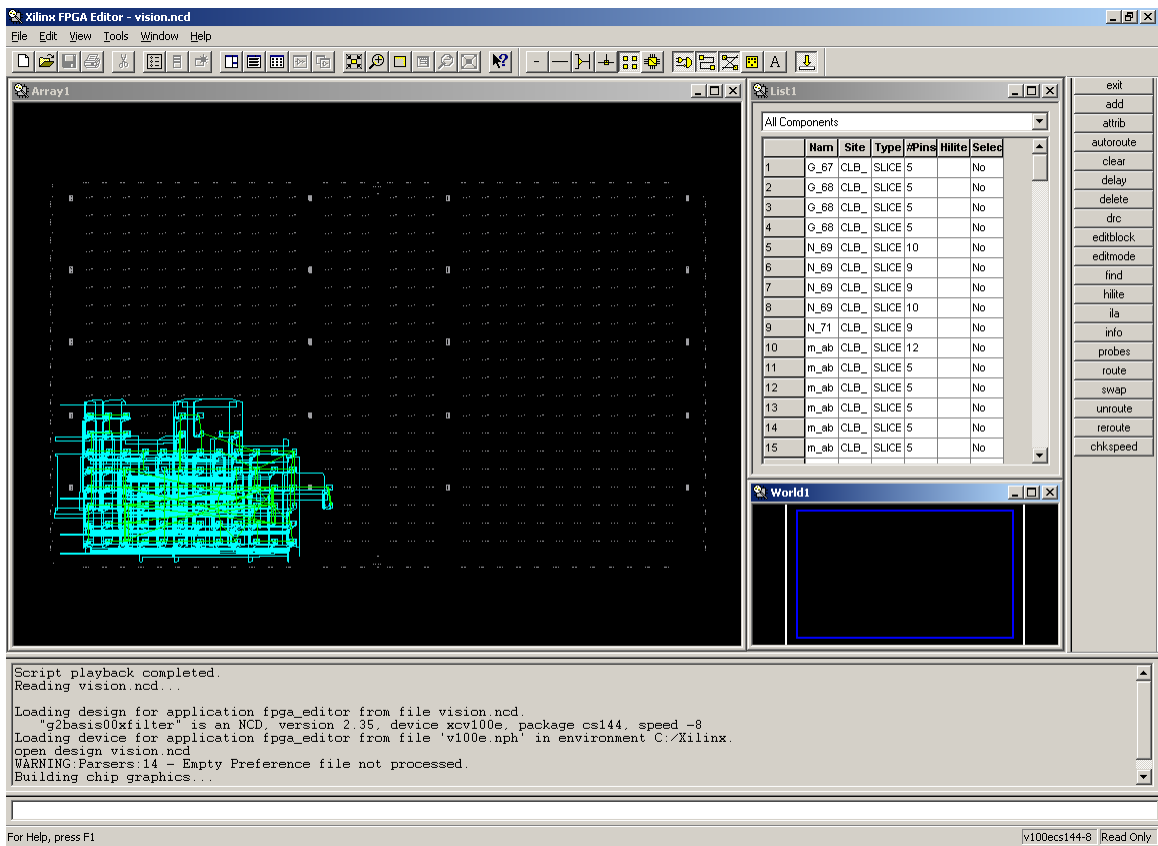


Figure 2-7: Screen Capture of the Xilinx FPGA Editor

The FPGA Editor is powerful for editing circuits at a low-level. However, it lacks the ease of use of a floorplanner, and it does not provide instant timing feedback after each circuit modification. EVE, in contrast, addresses these issues.

Chapter 3

EVE

In this chapter, we discuss our new methodology for assisting manual designs of FPGAs, and how it is implemented through the features of EVE, the Event Horizon Editor. In Section 3.1, we will discuss the design objectives of EVE. EVE has two operating modes: Timing Exact Microscopic Placement (TEMP) mode and pipelining mode. We will describe the features of the TEMP mode in Section 3.2, and the features of the pipelining mode in Section 3.3. Finally, we will discuss some limitations of EVE in Section 3.4.

3.1 Design Objectives

EVE has the following design objectives:

1. Target Real FPGA Architectures

Traditional FPGA research tools tend to work on simplified models of real FPGAs [BRM99]. These tools, for example, rarely represent carry chains correctly. Our goal in this work is to apply the Event Horizon concept and its implications to real devices so that we can deal with all of the realities that designs possess. EVE is targeted to a commercial FPGA architecture, the Xilinx Virtex-E [Xili2001b] family. It does so by integrating tightly with the backend tools provided by Xilinx [Xili2000].

2. Give Full Low-Level Control

The “Event Horizon” notion requires careful design of each microscopic piece of the circuit. EVE must permit the user to easily control placement and packing of each LUT, carry element and flip-flop, and to precisely control where flip-flops are inserted when pipelining.

3. Give Instant Performance Feedback

After each user move, EVE should immediately reroute and perform a full timing analysis to report the real circuit performance. This is usually not possible in automated algorithms, but it is acceptable for interactively editing small designs, as EVE is targeted to operate on designs with 250 or fewer LUTs.

4. Be Timing Budget Aware

EVE should be timing-budget aware. It should highlight circuit elements that violate the timing budget and quickly and accurately estimate the effect of a change to the circuit before it is applied. It should also provide a visual aid to

illustrate the “Event Horizon” itself (see Figure 3-2). The user can then decide where a circuit element can be placed on the chip without violating the timing budget.

5. Assist Pipelining

EVE should be able to assist the user to pipeline the circuit by maintaining correct functionality of the circuit throughout the pipelining process. It should also select optimal physical placement for pipelining flip-flops to minimize the critical path delay.

3.2 Timing Exact Microscopic Placement (TEMP) Mode

The Timing Exact Microscopic Placement (TEMP) mode of EVE deals with all aspects of the positioning of circuit elements. It permits microscopic placement of circuit elements while giving instant exact timing feedback. In Section 3.2.1, we first define the concept of a Timing Horizon (a modification of Von Herzen’s Event Horizon). Then in Section 3.2.2, we describe the features of the TEMP mode.

3.2.1 Timing Horizon

Figure 3-1 illustrates the concept of a “Timing Horizon”. It is based on the “Event Horizon” concept explained in Section 2.2. As a placement editor, EVE will calculate the effect on the critical path delay when a circuit element (in this case a LUT, marked with a circle in the figure) is moved to other target locations (marked in gray). In Figure 3-1, a

Timing Horizon of radius one (CLB) is displayed. For each target positions where the Timing Horizon is located, a number representing the change in critical path timing should the selected circuit element be moved there. A negative number means that the critical delay improves. When a target position is not feasible (it may be occupied or the target slice configuration is not compatible), it does not appear in the Timing Horizon. In EVE, we will call this Timing Horizon simply the “Horizon”. It is an important feature of EVE that the designer can use to evaluate the effect of moving a circuit element in the chip, to improve the overall circuit speed.

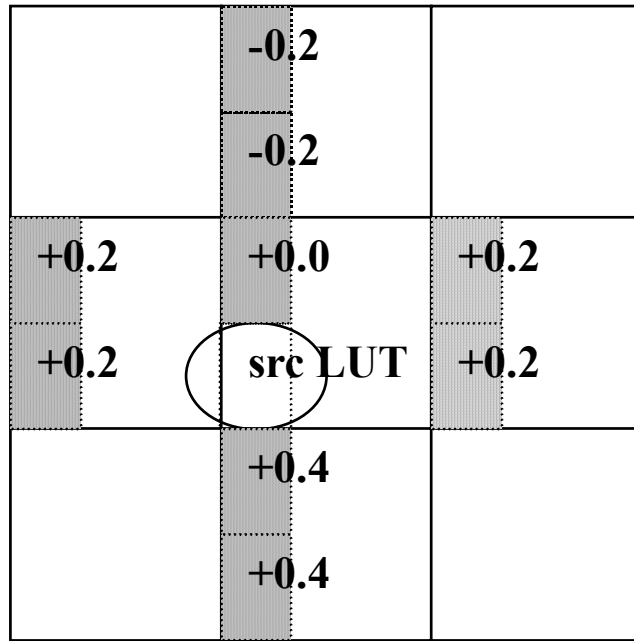


Figure 3-1: Timing Horizon

3.2.2 Features of TEMP mode

In the Timing Exact Microscopic Placement (TEMP) mode, the circuit is represented in a grid format, with each grid cell representing a CLB. Figure 3-2 shows a circuit on a Xilinx XCV100E [Xili2001b] chip. Each CLB has two slices, and each slice is divided into six components: two LUTs, two carry cells, and two FFs, as described in Section 2.1.1. In this mode, the placement of all circuit elements are shown, and logic packing and placement operations can be easily modified using a drag-and-drop paradigm.

EVE recognizes structural grouping of circuit elements such as carry chains, 5-LUTs, and 6-LUTs. When the user selects one of the circuit elements in such groupings, the whole structural group is automatically selected. The user can also select multiple objects and apply placement or routing operations to the group of selected objects.

On start up, the critical path of the circuit is highlighted. The current speed of the circuit is also calculated and displayed in the status window. The user can then perform the following operations:

1. Change Placement of Components

Select the components and drag them to the destination location. Eve does this better than the native Xilinx Floorplanner [Xili2001] because it immediately reroutes the circuit and reports the real circuit timing. EVE also immediately informs the user if the move is valid by displaying “X” markers on invalid target

positions. If the move results in an unroutable net, the effect of the move is reversed and the user is notified of the problem.

2. Packing/Unpacking of Slices

Slices are packed/unpacked based on how the components are placed. When a LUT is moved from one slice to another, the packing of the source and destination slices may be altered. This feature is also available to the native Xilinx Floorplanner, but it can only pass this packing/unpacking directive to the mapper as a slow batch task; and such packing/unpacking operation may not be applied successfully. In contrast, EVE can determine the packing feasibility instantly.

3. Change/Set Timing Budget

When the timing budget is set or changed, the design is timing-analyzed and the components and nets that violate the budget are highlighted. A typical methodology for using EVE is to slowly decrease the timing budget, and focus on a more timing critical part of the circuit, until the desired timing goal is met.

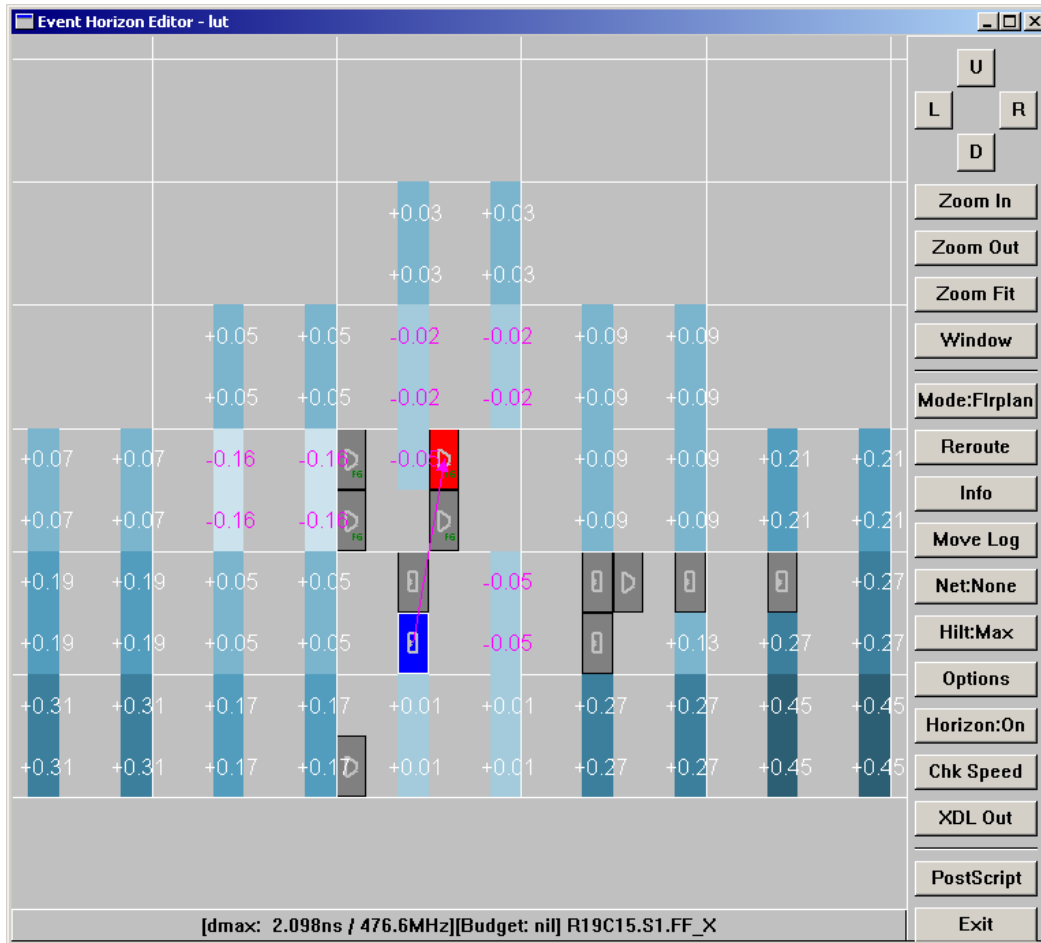


Figure 3-2: Screen capture of the TEMP mode showing a “Horizon”

4. Invoke Horizon

Here the user selects a component and press the “Horizon” button. A Horizon (based on the “Timing Horizon” notion described in Section 3.2.1) is displayed graphically as rings of colours, with a gradient indicating the “goodness” of placing the selected components at the indicated positions. A number is displayed in each valid target position, indicating the effect of the change. (Negative numbers indicate improvements in critical path delay). The user can control a

“Horizon Radius” parameter which controls in Manhattan distance within how many CLBs EVE should do the timing evaluation. Figure 3-2 shows a Horizon of radius 3 for a selected flip-flop component.

5. Nets Reroute

Timing of critical nets can be improved by ripping them out and re-routing them again. Select some components and press the “Reroute” button. All nets connected to the selected components will be ripped out and re-routed, while leaving other nets in the circuit intact.

6. Moves Playback / Design Restore

Each user move is recorded in a log file and the moves can be subsequently “played back”. Also, designs are saved after each move, so the user can restore the circuit back to any previous state.

7. Display Dynamic Delay Distribution

After each user move, EVE analyzes the timing of the circuit and outputs a delay distribution, which summarizes the number of delay paths having delays within different delay ranges. It gives the user an overall picture on the current state of the circuit.

After each move, EVE modifies the netlist as needed, incrementally re-routes nets, and performs timing analysis to report the real timing of the modified circuit. If the

horizon feature is enabled while the move is applied, a new horizon will be calculated and displayed. Using instant timing feedback and various budget-aware features, a user can produce superior performance circuits.

3.3 Pipelining Mode

Pipelining traditionally occurs during logic design, when the designer introduces pipeline stages to enable parallel execution of multiple circuitries to achieve a higher throughput. Pipelining in the Event Horizon methodology context, however, refers to the need to register logic elements when the physical placement becomes an obstacle to satisfy a high-speed design goal. Referring back to Figure 2-4, which describes this situation, the extra flip-flop inserted extends the Event Horizon to reach the target CLB, hence allowing the circuit to satisfy a timing goal by gaining clock speed at the cost of latency. We believe research in pipelining at the physical level will become more important as circuit speed pushes towards the limit of the silicon.

We need a pipelining assistant that allows the designer to fully control where pipelining flip-flops are inserted, yet helping the designer retain correct functionality of the circuit. The TEMP mode displays the physical locations of each circuit element, so it is ideal for performing packing/unpacking and placement operations. For pipelining, however, a such a circuit representation cannot present clearly to the user where flip-flops can be inserted because the graphical display will be very cluttered. It also lacks features that can facilitate pipelining. We thus propose the pipelining mode in EVE as a way to

present the circuit in a better form to assist pipelining. This is described in more detail in Section 3.3.1.

Note that not every net in the netlist can tolerate flip-flop insertion. The problem of flip-flop insertability is described more in Section 3.3.2. Also, sequential loops, which are commonly found in control circuits, cannot be pipelined because additional flip-flops can change the desired behaviour of the circuit. Section 3.3.3 discusses the need to hide loops in the circuit when doing pipelining.

When the user insert a flip-flop in the circuit, EVE will automatically determine where in the circuit to insert additional flip-flops to maintain correct functionality of the circuit. This is further discussed in Section 3.3.4 as the problem of flip-flop insertion. After flip-flop insertion, the user should have the freedom to move the pipelining flip-flops logically across other logic elements to retime [LS83] the circuit. Flip-flop motion refers to this problem of moving flip-flops across circuit elements while preserving correct functionality. It is described in detail in Section 3.3.5. To make flip-flop motion more useful, EVE can perform flip-flop tracing described in Section 3.3.6, which can help the user to identify an inserted flip-flop which is in the transitive fanin or fanout of a critical net. Finally, the actual synthesis of pipelining flip-flops into the netlist are discussed in Section 3.3.7.

3.3.1 Graphical Circuit Representation

This section describes how a circuit is graphically presented in the pipelining mode. It makes use of concepts discussed in the subsequent sections: Section 3.3.2 – Section 3.3.5. In the pipelining mode, the circuit is displayed as a directed acyclic graph (DAG). Each graph node represents an input or output pin of a logic slice, or the input and output ports of sequential elements, including flip-flops and Shift Register LUTs (SRLs). Each edge represents a logical connection between the graph nodes, which usually has an associated delay value, corresponding to an internal logic delay, or an external routing delay. Primary inputs are displayed at the top and primary outputs at the bottom of the DAG. A starting node labeled “START” connects to all primary inputs, while an ending node labeled “END” connects to all primary outputs to allow pipelining at the primary inputs and outputs (See Figure 3-3).

If there exists a sequential loop in the circuit, we need to detect and collapse it down into a single graph node, as described in Section 3.3.3. On start up, the circuit is analyzed and loops are collapsed into nodes. Each collapsed loop will have a prefix “LOOP_”. Each loop node internally contains a sub-graph of the circuit. The user can descend into the loop nodes and examine the graph nodes within.

Graph edges are coloured differently to indicate their status: critical nets are marked red while edges that are flip-flop insertable are marked green. A square appears when a flip-flop is inserted on an edge. A solid square indicates that the edge is flip-flop insertable (see Section 3.3.2), while a hollow square indicates that the edge is not flip-

flop insertable. A number appeared next to an edge indicates the number of edges connecting the nodes.

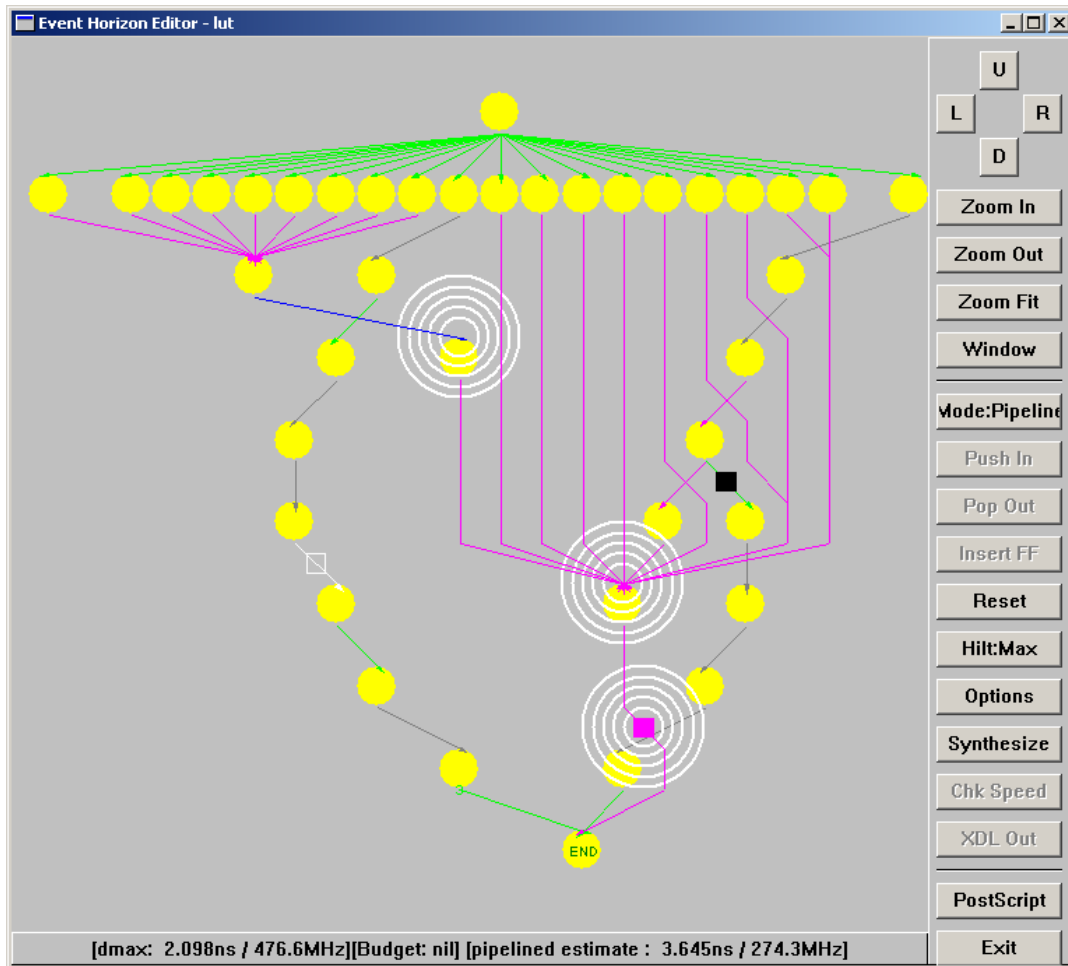


Figure 3-3: Screen capture of the pipelining mode

With this graphical representation, the user can clearly identify graph edges that are flip-flop insertable. Information that is not useful such as the sub-graphs corresponding to loops is eliminated from the graph to make it less cluttered. With a detail representation of connected graph nodes and edges, flip-flop insertion and flip-flop motion can be done intuitively as if the circuit is a combinational circuit.

3.3.2 Flip-flop Insertability

Not all edges in the DAG are flip-flop insertable (can be pipelined). The set of flip-flop non-insertable edges include:

- 1. Routing edges that join two carry cells**

This is a Xilinx Virtex-E architectural limitation.

- 2. Routing edges that join two 5-LUTs to form a 6-LUT**

This is a Xilinx Virtex-E architectural limitation.

- 3. Non-routing edges**

Routing edges correspond to routing connections in the circuit. We choose not to allow flip-flops to be inserted on non-routing edges to simplify the pipelining problem.

- 4. Edges that are in transitive fanin of asynchronous reset pins**

Edges in transitive fanin of asynchronous reset pins are eliminated from the DAG because they have no logical function except during system reset.

3.3.3 Loop Elimination

The implementation of loop detection is described in Section 4.6.1. Loops are detected and collapsed in the circuit one at a time. The process of loop collapsing is illustrated in Figure 3-4:

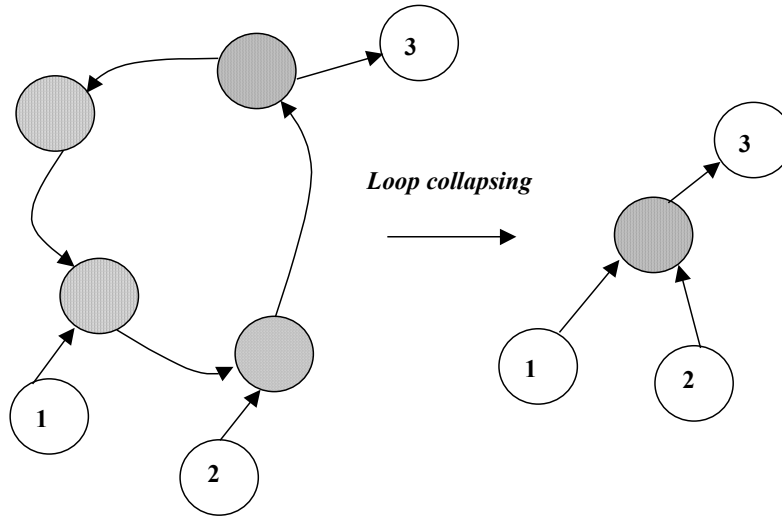


Figure 3-4: Loop Collapsing

3.3.4 Flip-flop Insertion

For both flip-flop insertion and flip-flop motion, we need to refer to the example illustrated in Figure 3-5. Flip-flop insertion, motion, tracing and synthesis (described in Section 3.3.4 – Section 3.3.7) are all user operations. We will describe what the user has to do and how EVE can help to perform the task.

During flip-flop insertion, the user has to select a flip-flop insertable edge and press the “Insert FF” button. A flip-flop is inserted in the specified location. Then EVE will insert additional flip-flops in the DAG to maintain correct circuit behaviour. For example, for the timing graph in Figure 3-5, if a flip-flop is inserted at edge $4 \rightarrow 6$, additional flip-flops must be inserted at edges $4 \rightarrow 7$, $5 \rightarrow 7$, $8 \rightarrow 9$. The resulting circuit still functions properly, with one additional cycle of latency across all paths. (Note that other

flip-flop to edge assignments are also possible.) The implementation of flip-flop insertion is described in Section 4.6.2.

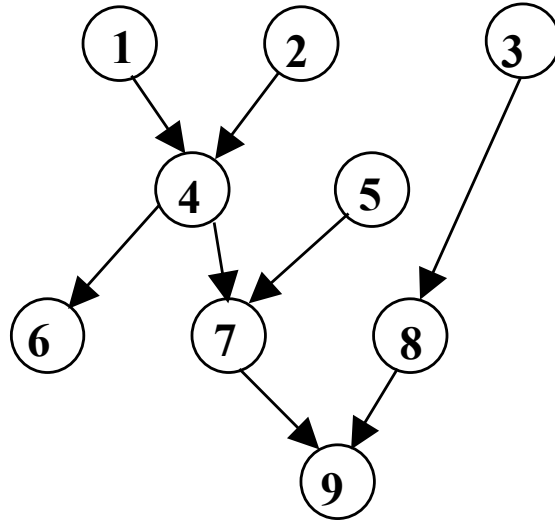


Figure 3-5: A timing graph used in the pipelining mode

3.3.5 Flip-flop Motion

After flip-flop insertion, the user is able to move the newly inserted flip-flops forward or backward using the “up” and “down” arrow keys. When a flip-flop is moved forward or backward across a node, EVE will make sure that the circuit is still functioning properly, by moving other flip-flops affected by the move. For example, for the timing graph in Figure 3-5, assume that flip-flops are inserted at edges $4 \rightarrow 6$, $4 \rightarrow 7$, $5 \rightarrow 7$, $8 \rightarrow 9$. Now if the user moves the flip-flop from $5 \rightarrow 7$ to $7 \rightarrow 9$, flip-flops at edge $4 \rightarrow 7$ and $5 \rightarrow 7$ will be removed, and a flip-flop is added to edge $7 \rightarrow 9$. The implementation of flip-flop motion is described in Section 4.6.3.

3.3.6 Flip-flop Tracing

Flip-flop motion can be used to move pipelining flip-flops onto timing critical nets. However, it is frequently not easy to identify the one flip-flop among others, which can be moved onto a specific net. In EVE, since flip-flops are inserted one at a time, i.e. each time, the circuit's latency will increase by one. For any given edge in the graph, there is at least one flip-flop that is in either its transitive input or transitive output. EVE can show visually a "trace" by marking the path from any edge to its associated pipelining flip-flops with concentric circles. It is invoked by pressing "s" after flip-flop insertion, and selecting an edge of interest. The "spacebar" key can be used to toggle the display of the markers. Flip-flop tracing is illustrated in Figure 3-3 by the concentric circles.

3.3.7 Inserted Flip-flop Synthesis and Placement

The new circuit speed is calculated on the fly as the user changes the flip-flop positions (for implementation see Section 4.6.5). The actual placement of the inserted flip-flops is optimized by a greedy algorithm to minimize critical path delay. The implementation of flip-flop placement is discussed in Section 4.6.4). When the user is satisfied with the flip-flop positions, the "Synthesize" button is pressed, and the inserted flip-flops are synthesized into the netlist and placed.

3.4 Limitations

EVE has the following limitations:

1. Circuits have to be synchronous, employing a single clock.
2. Circuits cannot use tri-state buffers, which exist in the Virtex-E.
3. Block RAM and LUT-RAM are not supported; however, LUTs can be configured as 1-bit shift registers or ROM.
4. Circuits have to be synthesized without I/O pads, because I/O blocks are not handled in EVE because we deal only with sub-circuits.
5. Primary output nets have to have an “END_” prefix.
6. EVE only supports circuits of the Xilinx Virtex-E family, and works with devices of size XCV100E or below.

Chapter 4

Implementation

In this chapter, we will discuss the implementation of EVE. In Section 4.0, we will look at the software architecture of EVE, which consists of four different layers of software. In Section 4.1, we will discuss EVE's Graphical User Interface (GUI) component and how the circuit is graphically represented in the pipelining mode. In Section 4.2, we will describe how EVE integrates with the Xilinx backend tools [Xili2000]. In Section 4.3, we will discuss how EVE obtains the logic and routing delays values of the Xilinx Virtex-E for performing internal high-speed timing analysis. In Section 4.4, we describe how timing analysis is done within EVE. In Section 4.5, we present the Horizon calculation. In Section 4.6, we discuss the implementation of pipelining features, including flip-flop insertion and motion. In finally in Section 4.7 and Section 4.8, we show how we verify that EVE has correct functional behaviour by performing design rule checks and simulations.

4.0 Software Architecture of EVE

EVE is an interactive graphical tool that has a four-layer architecture as illustrated in Figure 4-1. The Event Driven Layer responds to user moves. The Logic Abstraction Layer contains the current state of the circuit, including the netlist, timing graphs and editor states. Routines such as timing analysis, horizon calculation and delay database management also belong to this layer. The Interfacing Layer interfaces between the Supporting Layer and the Logic Abstraction Layer. It contains high-level graphics routines and backend interfacing routines. The Supporting Layer consists of tools or software packages EVE relies on. It includes the *EasyGL* [Betz98] graphics toolkit and the Xilinx backend tools [Xili2000].

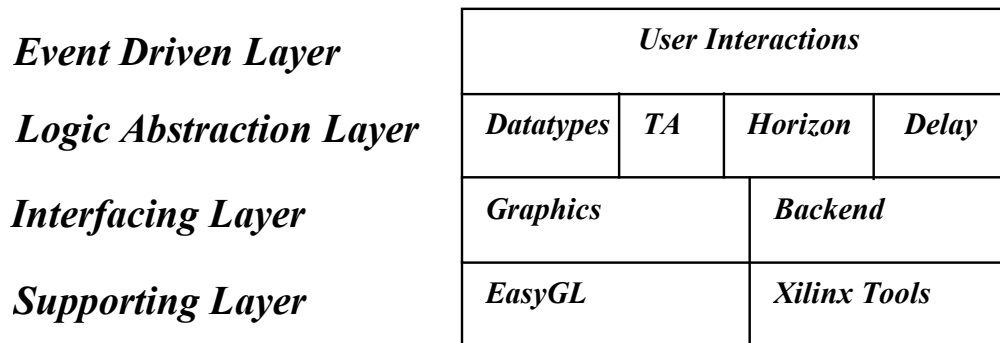


Figure 4-1: Architecture of EVE

4.1 GUI

4.1.1 EasyGL for Windows

The graphical user interface of EVE is built using *EasyGL* [Betz98]. It is a cross-platform graphics toolkit that is a part of the *VPR* [BH98] package. For the purpose of this project, we have ported *EasyGL* to the Win32 platform and enhanced its features. We call the new version *EasyGL for Windows* [Chow2001].

4.1.2 Generating Graph Layout for Pipelining Mode

We initially intended to use external graph drawing packages such as *dot* [GVR2000] to draw the DAG in the pipelining mode. However, the graph drawing algorithm used by *dot* has an $O(n^3)$ complexity, and is thus not suitable for displaying the timing graph, which may easily contain thousands of nodes and edges. As a result, we developed a set of graph drawing routines, which treats graph drawing, ironically, as a placement and routing problem. Graph nodes are placed on a rectangular placement and routing (P&R) grid, and the graph edges are then routed around the graph nodes. The placement of graph nodes uses a cost function that encourages shorter total edge length so that related nodes will be placed closely together. It also encourages more space between adjacent graph nodes on the same row of the P&R grid so the resulting graph looks less cluttered. Figure 3-3 shows a DAG drawn using this approach. We use *dot* for generating graph layouts for timing nodes within loops since the sub-graphs are small, with about only ten or fewer nodes.

4.2 Backend Integration

In Section 4.2.1, we will discuss the insufficiency of the existing Xilinx backend tools, and identify the problems we need to address. In Section 4.2.2, we will present the solution, which is interfacing with the native Xilinx FPGA Editor directly. In Section 4.2.3, we describe how EVE obtain real timing information of the circuit and maintain that knowledge with minimal effort during the course of editing the circuit.

4.2.1 Insufficiency of existing tools

The Xilinx *FPGA Editor* (described in Section 2.4.3) is a low-level graphical design tool that allows the designer to finely control slice configuration, placement of slices, and the use of routing resources. It is powerful enough to edit the circuit microscopically, but it lacks the ease of use of a floorplanner, and it does not provide instant timing feedback. On the other hand, the Xilinx *Floorplanner* (described in Section 2.4.2) does not allow modification of slice configuration, and it also does not provide timing feedback. These are the low-level manual design tools that we want to augment. EVE serves as a combined floorplanner and manual circuit editor, with the additional advantage of instant timing feedback, as well as other helpful features such as pipelining assist.

One challenge during the design of EVE is the ability to provide high interactivity, which requires very quick partial placement, routing, and timing analysis of the circuit after a user move. Using the set of command-line based Xilinx backend tools

including *PAR* (placer and router), *TRACE* (timing analyzer), and *XDL* (Xilinx proprietary circuit format to ASCII conversion utility), each user move could still take minutes to process. Clearly we need to bypass using these tools yet still perform the necessary tasks in a much shorter time. EVE achieves this by interfacing with the *FPGA Editor* directly.

4.2.2 Solution: Interfacing with FPGA Editor

To address the problem of slow CAD flow iterations, we decided to interface directly with the FPGA Editor, which has a full-featured set of textual commands for controlling various operations including slice configuration, placement, and routing. On start up, EVE spawns two copies of *FPGA Editor*. One copy serves as the “backend” where the real circuit changes are applied. The other copy serves as a “net delay reporter” which is described further in Section 4.3. EVE instructs both the “backend” and “net delay reporter” by sending commands to them using named pipes supported by Windows NT based platforms. The execution result in the backend is obtained for further analysis, by capturing text from the *FPGA Editor* window using Windows messaging API calls.

When EVE sends a command to the backend, it also instructs the backend to echo to its output window a command number, which is incremented after each command execution. When EVE captures the results from the backend, it locates the correct command number in the communication buffer to obtain the information it needs. The FPGA Editor receives commands from the named pipe by means of polling. It can only read one command from the pipe every 10ms. To get past this limitation, EVE packs

multiple commands into a single long command separated with command separators understandable by the backend. This technique greatly decreased the communication latency between EVE and the backend.

4.2.3 EVE: knowing circuit timing at all times

EVE determines the timing of the entire circuit at all times. It obtains the initial timing information from the Xilinx timing analyzer (*TRACE*). It then builds a timing graph of the circuit and performs subsequent timing analysis internally. When the user makes a move, EVE will calculate the effect of the move by using delay values stored in a delay database (described in Section 4.3), and estimate the resulting circuit critical path delay. The change is then communicated to the backend using named pipes. The backend makes the corresponding change, including logic configuration modification, netlist modification, and placement. Then it unroutes all nets affected by the change, and re-routes them using the critical path delay estimated by EVE as a timing constraint for timing-driven routing. This routing is very quick since it is only a partial re-route, and the unaffected nets are left untouched. Finally, the net delays of the modified nets are queried by EVE through the backend, and it will once again have complete knowledge of the updated circuit timing information. The interaction between EVE and the backend is summarized in Figure 4-2.

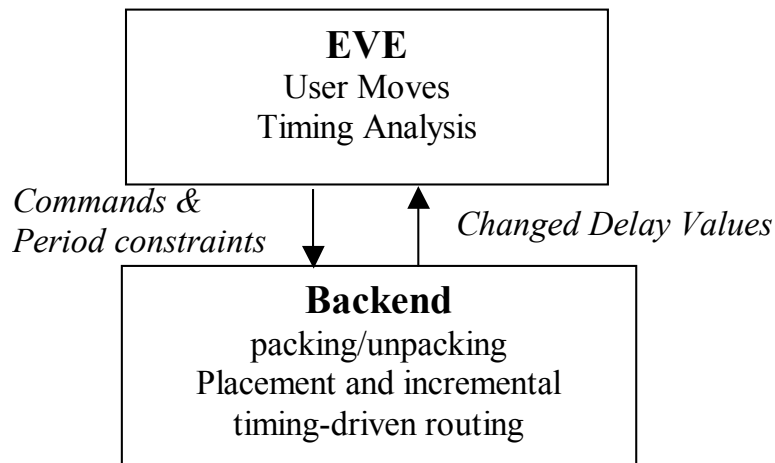


Figure 4-2: Interaction between EVE and the backend

4.3 Delay Extraction and Caching

In Section 4.3.1, we describe the general delay extraction problem, and the challenges that EVE faces for this problem. In Section 4.3.2, we describe how EVE extracts logic delays, which are delay values for all configurable pathways within a logic slice. In Section 4.3.3, we describe how EVE extract routing delay values. In Section 4.3.4, we describe how EVE compress the delay database to speed up database generation and conserve disk space and memory. In Section 4.3.5, we describe how the delay database compression scheme is derived. Finally, in Section 4.3.6, we describe how EVE retrieves delay values from the delay database.

4.3.1 Delay Extraction

Two types of delays are needed for timing analysis: logic delay within a slice, and routing delay. Since timing analysis is a very expensive operation, obtaining accurate delay information quickly is very important. Logic delays are usually modeled as constants since the number of configurable paths that exist within each slice is limited. They are usually pre-calculated and stored in look-up tables for fast future retrieval. Routing delays however, vary according to the routing resources taken up by a route. Each routing delay value is governed by an RC model, such as Elmore [Elmo48] and Penfield-Rubinstein [RPH83] models. These models take into account the length of the routing wires, and the number and type of switches (buffered or non-buffered) that the routes pass through. When doing routing, RC models of each route are built on the fly, and the associated delay is calculated quickly. For EVE, obtaining such delay information is hard because it has no knowledge of the RC characteristics of the commercial routing architecture. Without this knowledge, EVE has to obtain both logic and routing delay values by querying the Xilinx backend tools one delay at a time and then storing the delays in data files, which we refer to as the delay database.

4.3.2 Extracting Logic Delays

Logic delays are calculated and stored automatically for each chip with a given speed grade the first time EVE encounters the chip. It does so by enumerating all the possible configurable pathways present within a slice or in-between slices (as in the case

of a delay involving 6-LUT). It then writes out a Xilinx Description Language (XDL) description of the circuit containing all the paths of interest, with each path using different CLBs. XDL is a text-based language describing the internals of Xilinx circuits, including slice configuration and routing information. It can be translated into the Xilinx native NCD circuit format using the *XDL* utility. The design is then timing analyzed using a command-based timing analyzer program called *TRACE*. The logic delay values for each path are then extracted from the report file to form a delay-matching table. Such a table will provide mapping from various slice configurations to logic delay values. For Virtex-E, there are 230 such logic delay values. This process of delay-matching table construction is illustrated in Figure 4-3.

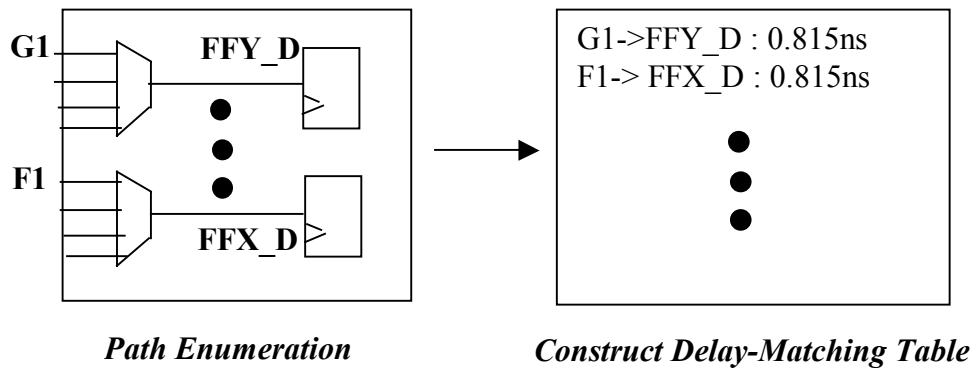


Figure 4-3: Constructing Delay Matching Table

4.3.3 Extracting Routing Delays

Extracting routing delays is much more difficult because of the large number of logic block pin to logic block pin delay values present in the Virtex-E chips. For

example, an XCV100E chip has 20 rows and 30 columns of CLBs and each CLB has two slices. For any given pin-to-pin route, it can originate from one out of six output pins in either slice *S0* or *S1*. It can also terminate in one out of twelve input pins in either slice *S0* or *S1*. The total number of possible routes of Manhattan distance of length five or less is $(2*5*5 + 2*5 + 1)*20*30*2*2*6*12 \approx 10.5$ million delay values. We estimate that a 450MHz Pentium-III processor can process four delay values per second, and each delay can be stored using 10 bytes. We would thus need about 30 days to generate the database and the data will take up 100MB of disk space. We would like to estimate how much data should be stored in the delay database, so we gathered some statistics from a vision application circuit, the number of delay values for each point-to-point routes in Manhattan distance. The results are shown in Figure 4-4. It shows that by storing routing delay values of Manhattan distance zero to ten, we can capture 88% of all pin-to-pin routes in the delay database. We therefore decided to gather pin-to-pin routing delays within this Manhattan distance range. To make the delay search space smaller, we devise a compression scheme making use of the symmetric nature of the Virtex-E routing architecture.

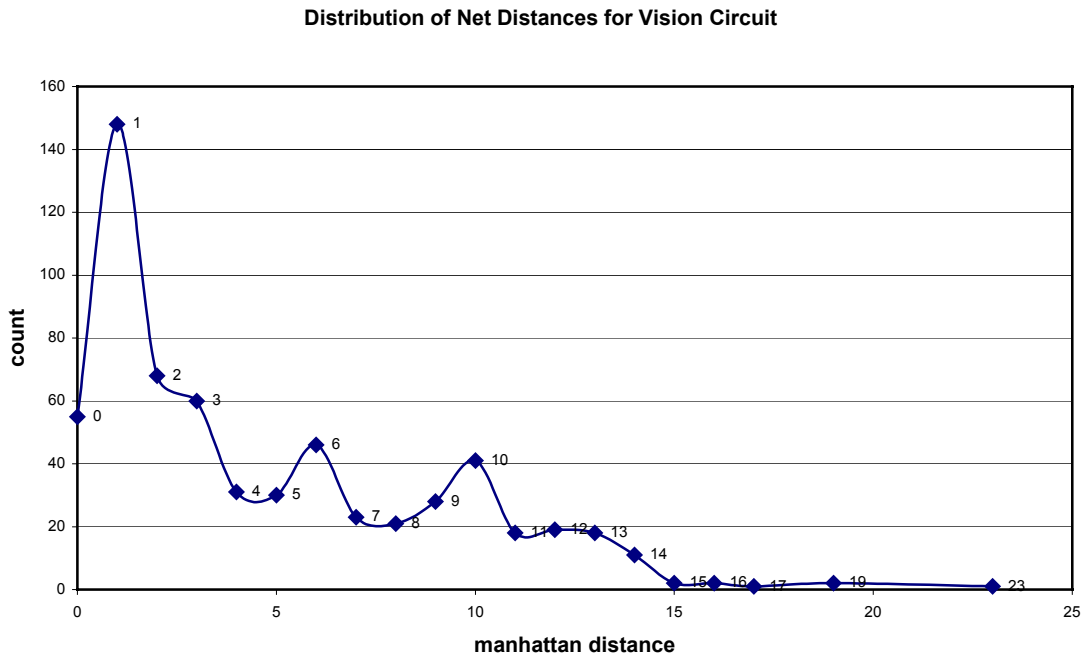


Figure 4-4: # of pin-to-pin connections vs. Manhattan Distance

4.3.4 Delay Database Compression

4.3.4.1 Notations

To describe the compression scheme better, we have to introduce some notations. We group related pin-to-pin routing delay values together in a group identified by the following format: $G=(S1,P1,S2,P2,X,Y)$. $S1$ and $P1$ specify the source slice and pin, while $S2$ and $P2$ specify the target slice and pin. X and Y are integers that represent the relative position of the target pin to the source pin. G is used to refer to this group of delays. Figure 4-5 shows a 3-D plot of the real routing delay values for the delay group $G=(0,XQ,0,G1,-1,-1)$ of the XCV100ECS144-8 device, which refers to the group of delay values with source pin located on XQ pin of $S0$, and target pin located on $G1$ pin of

$S0$, and target slice is one CLB west and one CLB north of the source slice. The pin-to-pin routing delay values in group G will then be represented using the notation (G,R,C) where R and C represents the row and column coordinate of the source pin. Each delay value (in ns) is plotted in 3-D space against the (R,C) coordinate. We can observe from Figure 4-5 that the routing delays are indeed fairly symmetrical across identical rows and columns. Using this representation, for the example shown in Figure 4-6, the delay $(G,19,5)$ will be $0.35ns$.

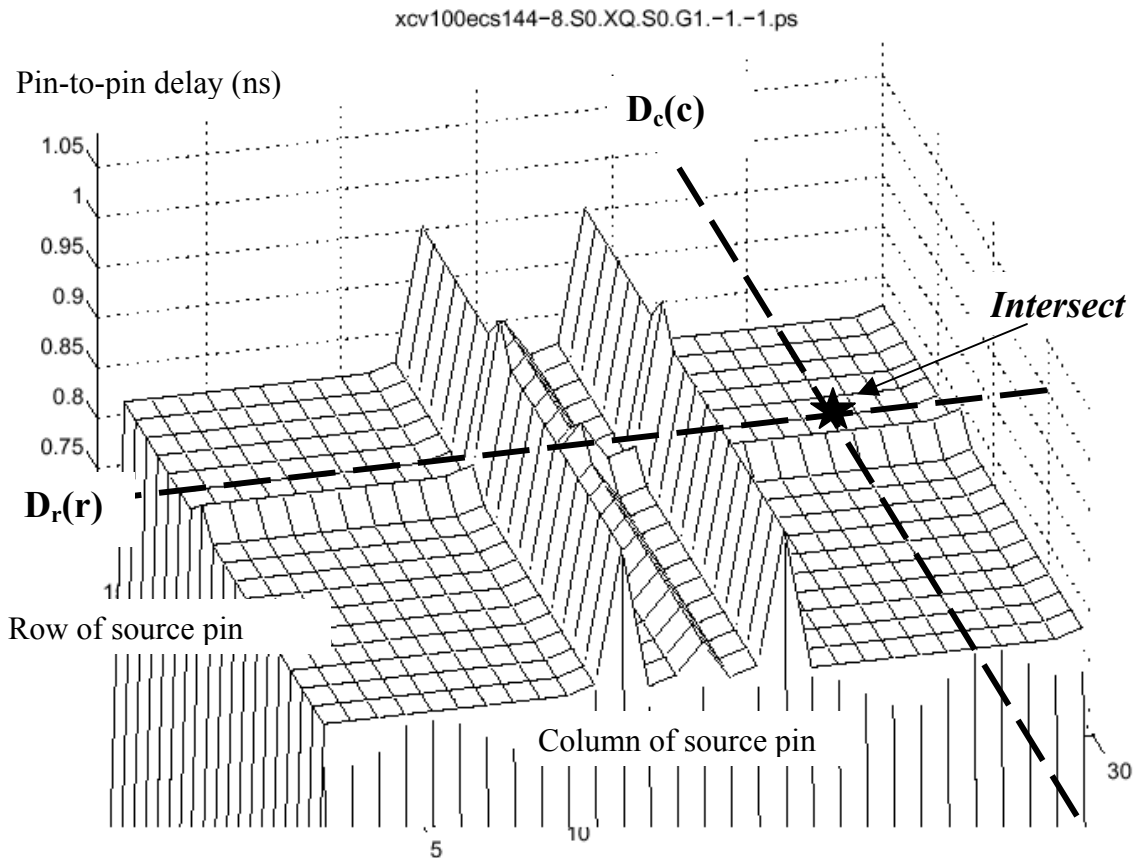


Figure 4-5: Routing Delay Profile for group G

4.3.4.2 The Compression Scheme

Now we will discuss the heuristics we used to “compress” the delay data space:

1. *Using Two One-Dimensional Functions*

A two-dimensional grid, with notation $D(r,c)$, where r and c corresponds to the row and column coordinate, is used to represent all the delay values in the group.

It requires $r*c$ data points. The following procedure is used:

- a. All $D(r,c)$ values are converted from floating point numbers into integers using a scaling factor of $0.02ns$ ($0.35ns$ will become $0.35/0.02 = 17$). We call the scaled values $D'(r,c)$.
- b. We locate an “intersect” point on the 2-D grid at the “base” of the 3-D plot, and record its delay. We refer to it as the “base delay” (b). All $D'(r,c)$ values are normalized by subtracting from the base delay to form $D''(r,c)$. The resulting data points will then contain mostly of zeroes.
- c. From the same “intersect” point, we can form two one-dimensional functions, $D_r(r)$ and $D_c(c)$, using the column and row vectors at the intersect, such that $D''(r,c) = D_r(r) + D_c(c)$ (illustrated in Figure 4-5). With these two functions, the number of data points needed to represent all $D(r,c)$ values becomes $r+c$.

2. *Eliminating Zeroes*

$D_r(r)$ and $D_c(c)$ are found to contain mostly zeroes. For example, $D_r(r)$ may be a vector like $[0\ 0\ 0\ 0\ 2\ 3\ 0\ 0\ 0\ \dots]$. Instead of processing the columns with 0 entries in $D_r(r)$, these column numbers are recorded, and are skipped for all subsequent rows. The same rule applies to $D_c(c)$.

3. Eliminating Duplicates

$D_r(r)$ and $D_c(c)$ frequently contain entries with the same value. Instead of processing all the entries with the same delay value, only one entry is processed. For example, for the vector $[0\ 0\ 2\ 3\ 0\ 0\ 2\ 3\ 0\ 0\ \dots]$, columns 3-4 and 7-8 are the same, so columns 7-8 are not processed. This rule can be applied to both $D_r(r)$ and $D_c(c)$.

4. Using Symmetry of Pins

Delays with $PI = X$ and $PI = Y$ are the same. Only one PI value needs to be processed. The same also applies for $PI = XQ$ and $PI = YQ$.

5. Record Extra Data Points

Data points, which cannot be calculated accurately using the above compression scheme, are recorded individually.

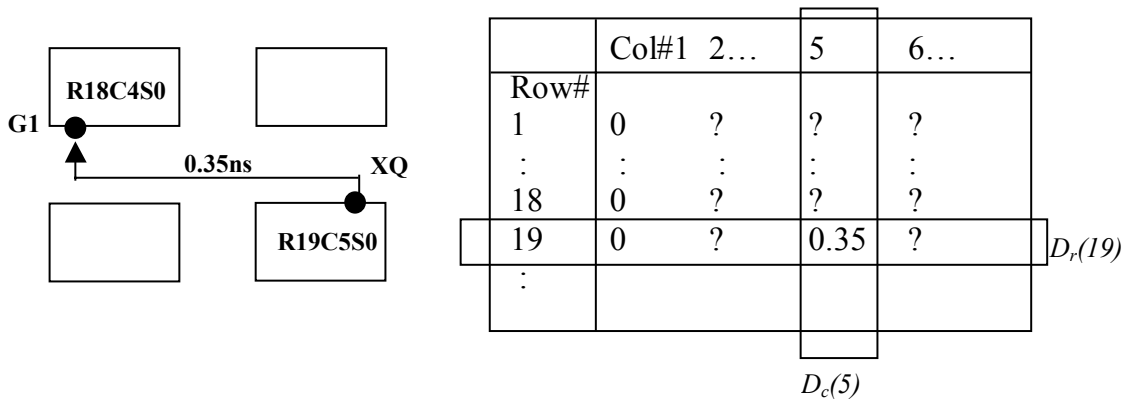


Figure 4-6: Routing Delay Group G

Using the heuristics given above, the search space is compressed by about *100* times. The delay database for routes with Manhattan distance five or less would then need about *0.3* days to calculate and *1MB* to store. All delay values as well as other information including: base delay, intersect point coordinate, zero matching columns/rows, duplicate matching columns/rows and extra data points, are generated and recorded in data files using a set of perl scripts. In EVE, the data files are loaded into a group of efficient data structures, which we refer to as the delay database. Delay retrieval from the database is quick, and the whole database consumes about *20MB* of physical memory.

4.3.5 Generating Compression Scheme

According to the compression scheme laid out above, EVE will first have to generate the delay values for the whole 2-D array for a set of descriptive cases to obtain the rules for the compression scheme. The cases chosen are the set of delay groups with notations: $S1=\{S0,S1\}$, $P1=\{YB,YQ,X\}$, $S2=\{S0,S1\}$, $P2=\{F1,BX,SR\}$, $X\&Y=m(N)$, $N=\{0-10\}$. The notation $m(N)$ refers to the set of (X,Y) values that has a Manhattan distance of N . The compression scheme is derived from these cases, and is in turn used to generate other delay values in the entire search space. To obtain a delay value (G,R,C) with $G=(S1,P1,S2,P2,X,Y)$, EVE will send commands to the delay reporter (a copy of *FPGA Editor* invoked on start up), building a pin-to-pin connection from pin $P1$ of slice ($row = R$, $column = C$, $slice = S1$) to $P2$ of slice ($row = R+Y$, $column = C+X$, $slice = S2$). The delay of the pin-to-pin connection is then queried from the delay reporter.

4.3.6 Routing Delay Retrieval

When a routing delay is needed for timing calculations, EVE will look in its pre-generated delay database for a match, following the stored compression scheme. If the delay is not found (for example, a delay with $m(N)$, $N > 10$), EVE will once again ask the delay reporter to calculate the missing routing delay. The retrieved delay value is then cached in the delay database. Subsequent retrieval of the same delay value will be from the cache instead. The routing delay retrieval algorithm is summarized in Table 4-1.

```

retrieve_delay(r1, c1, s1, p1, r2, c2, s2, p2) { // delay from pin p1 of (r1,c1,s1) to pin p2 of (r2,c2,s2)
    FINENESS = 0.02; //ns

    If (delay is COUT->CIN or F5->F5IN) {
        Return 0.0;
    }
    If (delay is not pre-calculated) { //within the Manhattan distance we used to generate the db
        Check if delay (d) is cached in the db.
        If yes, return d*FINENESS;
        If no, {
            query "delay reporter" (d) for the new delay,
            cache the newly retrieved delay in db as d/FINENESS,
            return d;
        }
    }
    else { // in db
        Check if it's an extra data point.
        If yes, return d*FINENESS;
        If no, { // generate value from column & row vectors
            (x,y) = get_intersect(r1, c1, s1, p1, r2, c2, s2, p2);
            b = get_base_delay(x,y);

            // check zero and duplicate rules
            if (zero rule applies to r1)
                diffrr = 0;
            else if (duplicate rule applies to r1)
                diffrr = get_delay_of_duplicate_of(r1);
            else
                diffrr = get_Dr(r1);

            if (zero rule applies to c1)
                diffcc = 0;
            else if (duplicate rule applies to c1)
                diffcc = get_delay_of_duplicate_of(c1);
            else
                diffcc = get_Dc(c1);

            return (b+diffrr+diffcc)*FINENESS;
        }
    }
}

```

Table 4-1: Pseudocode for Retrieving Routing Delay

4.4 Partial Incremental Timing Analysis

In Section 4.4.1, we first describe how EVE constructs the timing graph by comparing slice configuration against a pre-calculated look-up table. Then in Section 4.4.2, we discuss how EVE performs its internal timing analysis.

4.4.1 Constructing Timing Graph

Given a certain slice configuration string, EVE needs to figure out what delay paths are present so it can setup timing graph correctly. Referring back to the delay-matching table illustrated in Figure 4-3, EVE makes use of this delay-matching table to construct the timing graph for timing analysis. Assume that we encounter a slice with a matching slice configuration of “*GI->FFY_D*”, we can then setup the timing graph accordingly, with two timing nodes: *GI* and *FFY_D*, representing the *GI* input pin and the *D* port on the *FFY* flip-flop, linked together by a timing edge with the previously recorded delay value of *0.815ns* (see Figure 4-7).

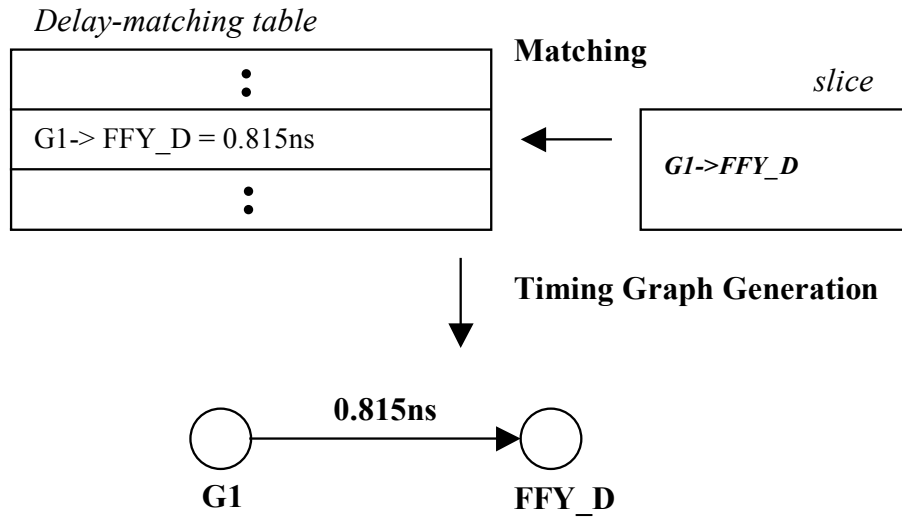


Figure 4-7: Delay Matching

4.4.2 Timing Analysis

Timing analysis is performed in EVE using a forward and backward two-sweep approach based on the following equations, as presented in [HSC83], for each node, i , in the timing graph:

$$T_{arrival}(i) = \text{Max}_{\forall j \in fanin(i)} \{T_{arrival}(j) + delay(j, i)\} \quad (4-1)$$

$$T_{required}(i) = \text{Min}_{\forall j \in fanout(i)} \{T_{required}(j) - delay(i, j)\} \quad (4-2)$$

$$slack(i, j) = T_{required}(j) - T_{arrival}(i) - delay(i, j) \quad (4-3)$$

We assume that the reader is familiar with the concepts of arrival time, required time, and slack. A full timing analysis is performed after every user move. To speed up this task, the data structures representing the current circuit state are updated incrementally. Only the parts of the timing graph corresponding to the modified slices and nets are re-built. The timing graph is then updated and timing analyzed to obtain the new circuit speed.

4.5 Horizon Calculation

The “horizon” is produced by evaluating multiple placement alternatives. A user-defined parameter called the “horizon radius” governs in Manhattan distance, how many CLBs away the selected components should be moved from their original positions, for evaluating the “goodness” of the move. For each target position, EVE first determines if the move is valid, by checking whether the target positions are already occupied and whether the implied packing/unpacking operations are valid. Then, it builds a temporary circuit resulting by moving the target to each valid location and perform a full-timing analysis on it. The result is then subtracted from the current circuit speed and the difference in timing is displayed in the target position (negative means timing improved). A “Horizon” is then displayed, with all valid target positions displayed using a gradient of colours, with lighter colours represent better positions (see Figure 3-2). A horizon of radius three takes about two seconds to calculate on a 1GHZ Pentium-III machine.

4.6 Pipelining

EVE assists pipelining in the following ways: 1) it enforces correct circuit functionality by addition or movement of related flip-flops, 2) it optimizes placement of pipelining flip-flops to reduce critical delay, 3) provides instant feedback on timing of the resulting circuit after flip-flop insertion. The following rules govern how EVE assists pipelining:

Rule 1: On flip-flop motion: The functionality of the circuit is maintained if a re-timing operation is done such that 1) for a forward retiming move, a flip-flop is removed from all input pins of a block, and a flip-flop is added to all the output pins of the block, or 2) for a backward retiming move, a flip-flop is removed from all output pins of a block, and a flip-flop is added to all the input pins of the block. The block can represent and part of the circuit. This is illustrated in Figure 4-8:

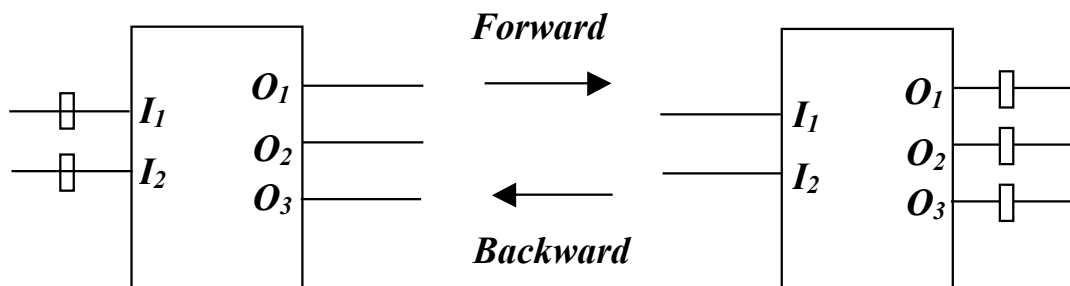


Figure 4-8: Forward and Backward Retiming

Rule 2: Flip-flops with synchronous reset or clock-enable pins can be transformed into basic flip-flops with a mux. The *SR* signal selects between *INIT/D*, while the *CE* signal selects between *D/Q*. Therefore the *SR* and *CE* pins behave like normal input pins to the flip-flop during pipelining operations (see Figure 4-9).

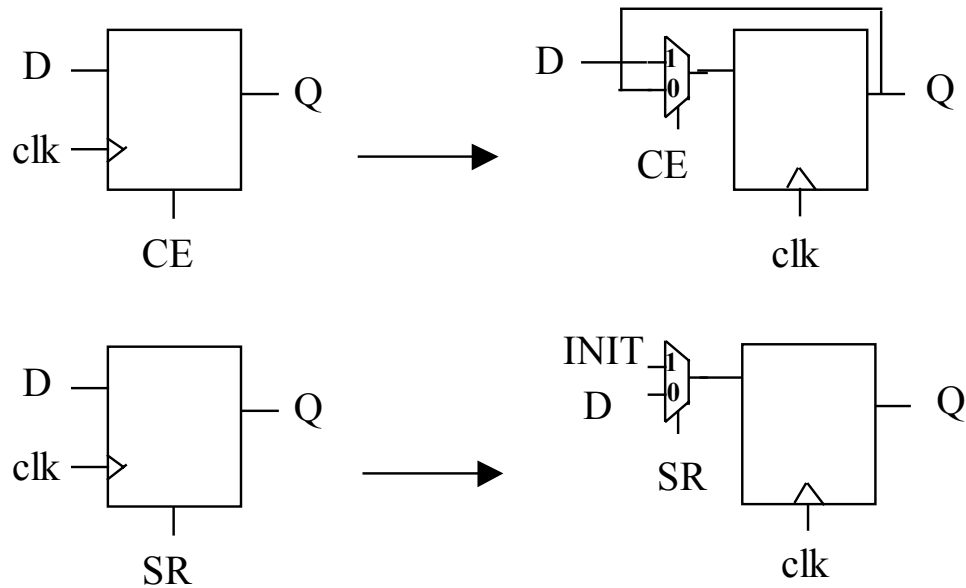


Figure 4-9: Transforming Flip-Flops with SR and CE

With these two rules, retiming operations can be performed across all circuit elements including flip-flops (according to Rule 2), provided that the operation conforms to the conditions described in Rule 1.

In Section 4.6.1, we describe an algorithm for detecting loops in EVE. In Section 4.6.2 and Section 4.6.3, we describe how EVE maintains circuit functionality during flip-flop insertion and flip-flop motion respectively. In Section 4.6.4, we describe how EVE

determines the physical placement of inserted flip-flops to maximize circuit speed. In Section 4.6.5, we describe how EVE estimates the effect of inserted flip-flops quickly. Finally, in Section 4.6.6, we discuss the problem of initial state assignment on inserted flip-flops.

4.6.1 Loop Detection

Loop detection has two steps: 1) determine if an edge $a \rightarrow b$ is in a loop, 2) given a set of edges that are in loops, find the loops. Step one is determined based on the concept illustrated in Figure 4-10. For determining whether an edge $a \rightarrow b$ is in a loop, we need to generate a few data structures from the timing graph: 1) for each node i , a set $FI[i]$ of FFs that fanouts to node i , 2) for each node i , a set $FO[i]$ of FFs that node i fanouts to, 3) a transitivity matrix M which specifies for each FF, what other FFs it can reach. The FI and FO sets are obtained during a forward and backward traversal of the timing graph. The transitivity matrix M is then subsequently generated from the FI sets.

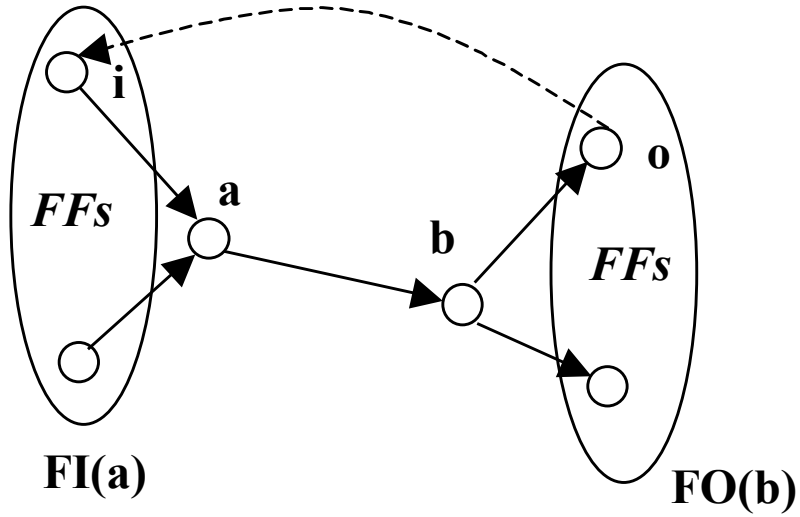


Figure 4-10: Finding Edge in Loop

Step two simply takes the set of edges that are in loops and traverse them, finding one loop at a time. The algorithm is summarized in Table 4-2:

```

Set FI[1..n];
Set FO[1..n];
Matrix M[1..n][1..n];
Set EL; // edges in loop
Set L; // set of found loops;

void collect_data(n) { // n = total number of nodes
    for (each node i = 1 to n) {
        FI[i] = set of FFs that fanouts to node i.
        FO[i] = set of FFs that node i fanouts to.
    }
    M = generate_transitivity_matrix(FI[1..n]);
}

int edge_in_loop(a, b) { // edge is from node a to node b
    return if (there exist some i and o such that
        1) i is in the set FI[a] &&
        2) o is in the set FO[b] &&
        3) M[o][i] = 1)
}

void find_loops() {
    L = {};
    while(EL is not empty) {
        do {
            E = an edge in EL, mark it as visited;
            traverse E to find the next edge that's also in EL, mark it as visited
        } until we encounter an edge that has already been visited.
        Add the loop to L;
        Remove all the edges of the loop from EL
    }
}

```

Table 4-2: Pseudocode for Finding Loops

4.6.2 Flip-flop Insertion

When the user inserts a flip-flop on a flip-flop insertable edge (defined in Section 3.3.2), EVE has to insert additional flip-flops to preserve the functionality of the circuit.

Referring back to the example illustrated in Figure 3-5, if a flip-flop is inserted at edge $4 \rightarrow 6$, additional flip-flops have to be inserted at edges $4 \rightarrow 7$, $5 \rightarrow 7$, $8 \rightarrow 9$. The additional flip-flop positions are determined based on a continuous forward and backward sweeping algorithm presented below:

```

Queue Q;
Array Processed[1..n]; // there are n edges in the graph

Void Traverse_forward_mark_back_edges(int n) {
    Traverse forward from node n,
    Marking all edges as processed.
    If there is a back edge that is not yet processed,
    Add it to Q;
}

Void Traverse_backward_mark_forward_edges(int n) {
    Traverse backward from node n,
    Marking all edges as processed.
    If there is a forward edge that is not yet processed,
    Add it to Q;
}

Void insert_ff(int i, int j) { // an edge from node i to node j
    E = get_edge(i, j); // E is the edge number of edge(i,j)
    Initialize Q to NULL and Processed entries to 0s;
    Add edge E to queue Q;

    While (Q is not empty) {
        E = dequeue(Q);
        Add FF to edge E;
        If (!Processed[E]) { // not processed
            Traverse_forward_mark_back_edges(j);
            Traverse_backward_mark_forward_edges(i);
        }
    }
}

```

Table 4-3: Inserting Flip-Flops

This algorithm first inset a FF on the supplied edge, then it marks all its transitive fanin and fanout edges as processed. For back edges encountered during a forward traversal or forward edges encountered during a backward traversal, insert FFs on them if they're not marked as processed. This process continues until all edges are visited.

4.6.3 Flip-flop Motion

When the user moves a flip-flop forward or backward across a node, EVE will make sure that the circuit still functions properly, by moving other flip-flops affected by the move. Referring back to the example in Section 3.3, for the timing graph in Figure 3-5, assume that flip-flops are inserted at edges $4 \rightarrow 6$, $4 \rightarrow 7$, $5 \rightarrow 7$, $3 \rightarrow 8$. Now if the user moves the flip-flop from $5 \rightarrow 7$ to $7 \rightarrow 9$, flip-flops at edge $4 \rightarrow 7$ will be removed. This is the application of Rule 1 (retiming forward): a flip-flop is removed from each input edge of node 7 and added to each output edge of node 7. The flip-flop motion operation is performed following these rules:

- **Forward retiming:** if T represents the transitive fanin of the node, N , where retiming is applied, add a flip-flop to edges which has the source node in N but the destination node not in N , and is also in the transitive fanout of a pipelining flip-flop. Afterwards, remove all pipelining flip-flops in N .
- **Backward retiming:** if T represents the transitive fanout of the node, N , where retiming is applied, add a flip-flop to edges which has the source node not in N

but the destination node in N , and is also in the transitive fanin of a pipelining flip-flop. Afterwards, remove all pipelining flip-flops in N .

4.6.4 Placement of Inserted Flip-flops

EVE helps the user optimize the positions where the inserted flip-flops are physically placed in the circuit, by minimizing the circuit delay using the following procedure:

1. All edges that need to have a flip-flop inserted are sorted in increasing order based on the edge slack. Edges with smaller slack are more timing-critical, so the placement of the corresponding flip-flop is determined first.
2. For each inserted flip-flop on an edge E , we will explore a number of CLB positions, and evaluate the resulting timing of the net associated with E . A number of CLBs are examined for the suitability for flip-flop insertion. Such CLB locations S are in the neighbourhood of the two end points of E . For each edge E to be pipelined, a list containing all the CLBs of the chip is sorted in increasing order based on the Manhattan distance from the end points of E . We call the sorted list L .
3. Assume that the delay database contains routing delays with Manhattan distance less than or equal to N . Since querying for a delay value not within the delay database takes a considerably longer time, EVE will first explore positions in L for all positions that are within N CLBs from the flip-flop end points. All feasible

flip-flop positions are evaluated according to Section 4.6.5 and the best position is obtained.

4. If a solution is not yet found, EVE will continue to explore the remaining CLB positions in L . If none of the positions are valid, the user will be informed of the lack of space on the chip

4.6.5 Period Estimation

Period estimation has two steps: 1) determine the periods of nets that fans in and out of each newly inserted flip-flop, 2) determine the overall circuit speed. For step 1 (refer to Figure 4-11), we have to calculate two period values: period 1 refers to the cycle ending at the inserted flip-flop, and period 2 refers to the cycle that starts from it. Assume that the edge we want to pipeline is $s \rightarrow d$. There can be two cases when a flip-flop is inserted: For case 1, the newly inserted flip-flop shares the same slice as the node s . Then period 1 is described by:

$$Period_1 = Max_{\forall i \in fanin(s)} \{T_a(i) + T_{setup}(i)\} \quad (4-1)$$

For case 2, the newly inserted flip-flop occupies a new slice, and period 1 is described by:

$$Period_1 = T_a(s) + d(s, f) + T_{setup} \quad (4-2)$$

Period 2 is the same for both cases, and is described by:

$$Period_2 = budget - T_r(d) + d(q, d) + T_{clk \rightarrow q} \quad (4-3)$$

For step 2, we first create a set E containing all edges in the timing graph. Then we look at each edge that is being pipelined, and record the maximum of the corresponding period 1 and period 2 values. We then traverse the edge forward and backward, marking all edges as visited. This process is repeated for all edges to be pipelined. For all edges in S that have not been visited, we obtain its associated period using its edge slack. The overall circuit speed is then the maximum period that are calculated.

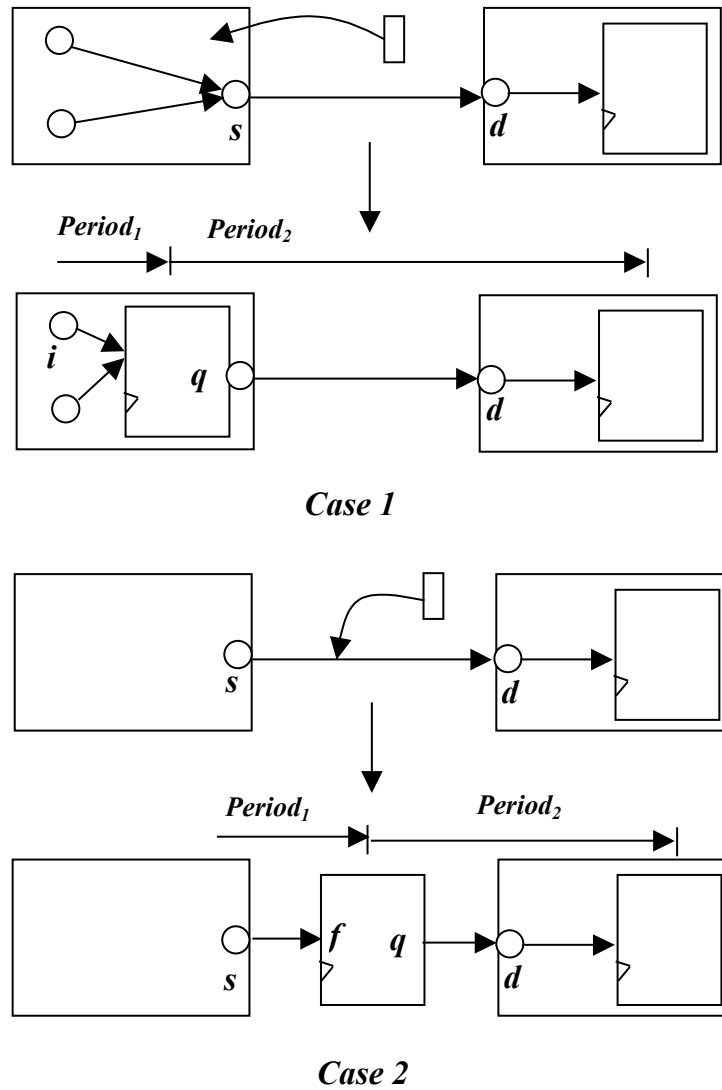


Figure 4-11: Period Estimation

4.6.6 Flip-flop Initial State Calculation

Initial states of inserted flip-flops need to be computed to produce functionally equivalent circuits after pipelining operation. It is a known NP-complete problem, and is

addressed in [CW99]. EVE does not deal with this problem. All newly inserted flip-flops are initialized to 0 on power up.

4.7 Design Rule Check (DRC)

After each user move, a Design Rule Check (DRC) is performed to ensure the correctness of the circuit. EVE instructs *FPGA Editor* (the backend) to perform a DRC check on the whole circuit, for problems related to slice configurations and routing resource usage. EVE retrieves the result and alerts the user if any errors are detected, based on a warning number that the backend provides for each problem. The list of allowable warnings include:

- DesignRules:2 - Netcheck: # unrouted net warnings were not reported.
- DesignRules:10 - Netcheck: The signal \$ is completely unrouted.
- DesignRules:367 - Netcheck: Loadless. Net \$ has no load.
- DesignRules:368 - Netcheck: Sourceless. Net \$ has no source.

4.8 Validation

In addition to changing placement and routing, EVE may also change the slice configuration and netlist, and additional flip-flops may be added to the circuit during

pipelining. To ensure that the modified circuits still perform correctly, test bench circuits are generated and timing simulations are performed following this procedure:

1. *ngdanno* (a Xilinx tool) is used to generate an intermediate file from the circuit in NCD format.
2. *ngd2vhdl* (a Xilinx tool) is used to convert the intermediate file into three files: a VHDL circuit, a VHDL test bench circuit and an SDF delay file.
3. *Synopsis-vss* is used to simulate the test bench.

Two circuits, Vision and Batcher (described in Section 5.1), have been tested to be functionally correct using the procedure described above, after modifications using both the TEMP and pipelining modes.

Chapter 5

Experimental Results

In this chapter, we evaluate the quality of results EVE produced for both the Timing Exact Microscopic Placement (TEMP) and pipelining mode. In Section 5.1, we describe the eight circuits used to evaluate EVE. They consist of FIR filters, multipliers and dividers, as well as components of networking and 3D-graphics applications. Each circuit has approximately 250 or fewer LUTs. In Section 5.2, we describe how we produce a set of baseline results for comparison purposes, using state-of-the-art logic synthesis and placement and routing tools. In Section 5.3, we describe how the author improved the circuit speed of each circuit, manually using EVE's TEMP mode. In Section 5.4, we present results obtained using both EVE's TEMP and pipelining modes.

5.1 Benchmark Circuit Descriptions

The circuits we used to evaluate EVE are described in this section. For the eight circuits used, six were taken from previous or on going projects developed at the University of Toronto, including one circuit from a vision application [MR2000], two circuits from a 3D-graphics application [Fend2002], three others from a networking chip design project [BCK2000], one was obtained from OpenCores.org [Open2001], and one was generated using the Xilinx CoreGen utility (An IP Core generator) [Xili99]. Each of the circuit has about 250 or fewer LUTs. The following sections describe each circuit in detail.

5.1.1 Vision

The Vision circuit is an FIR filter circuit originally used in a vision application presented in [MR2000]. The circuit is highly pipelined using a pyramid structure of shifters and adders. It also contains control circuitry for interfacing with other components of the vision chip. It uses 142 LUTs and 241 FFs. The structure of the circuit is illustrated in Figure 5-1.

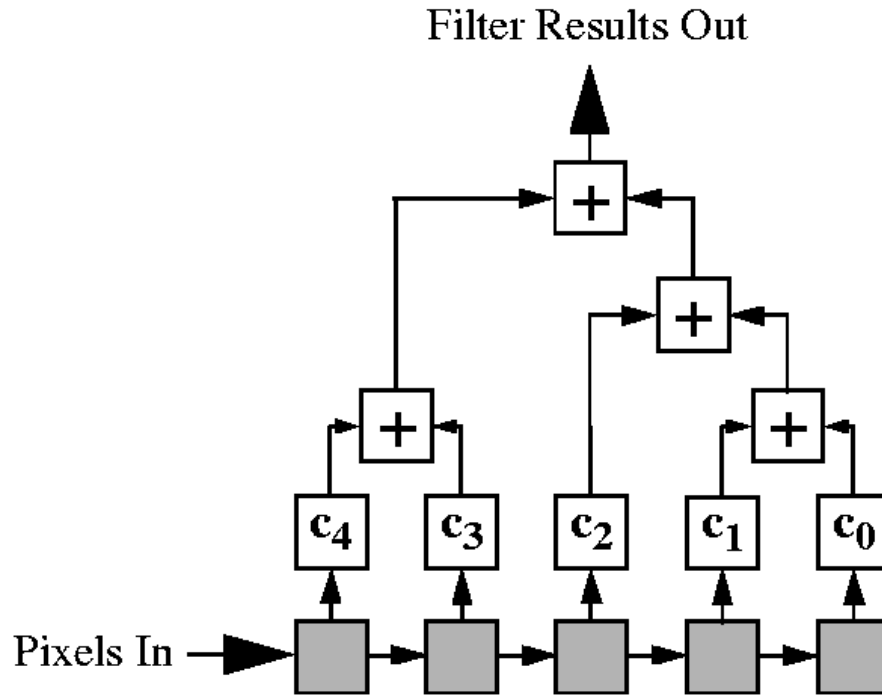


Figure 5-1: Structure of the Vision circuit (taken from [MR2000])

5.1.2 Batcher

The Batcher circuit is an ATM packet-sorting network that sorts incoming packets by serially comparing the bits of two packets. It is made up of an array of smaller batcher sorting elements, each handling the sorting of two ATM packets at a time. The structure of a batcher element is shown in Figure 5-2. The Batcher circuit is a component of the StarBurst ATM chip [BCK2000] project developed at the University of Toronto in 2000. It is currently being fabricated by Canadian Microelectronics Corporation (CMC) using a 0.35um CMOS technology. It uses 252 LUTs and 455 FFs.

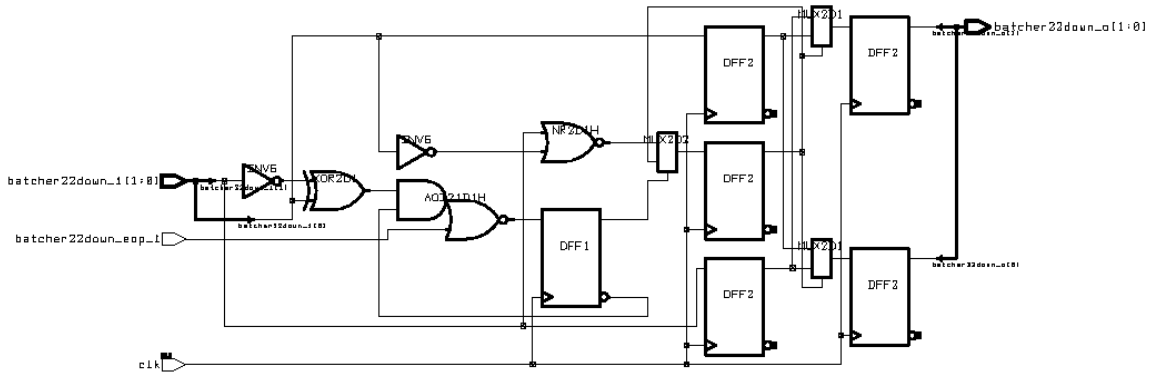


Figure 5-2: Structure of a Batcher Element

5.1.3 Banyan

The Banyan circuit is also a component of the StarBurst ATM chip [BCK2000] described above. It is a packet routing network that is responsible for delivering ATM packets to specific destination ports based on the address field stored in the ATM packets. It is made up of an array of smaller banyan elements. The structure of a banyan element is shown in Figure 5-3. It uses 165 LUTs and 311 FFs.

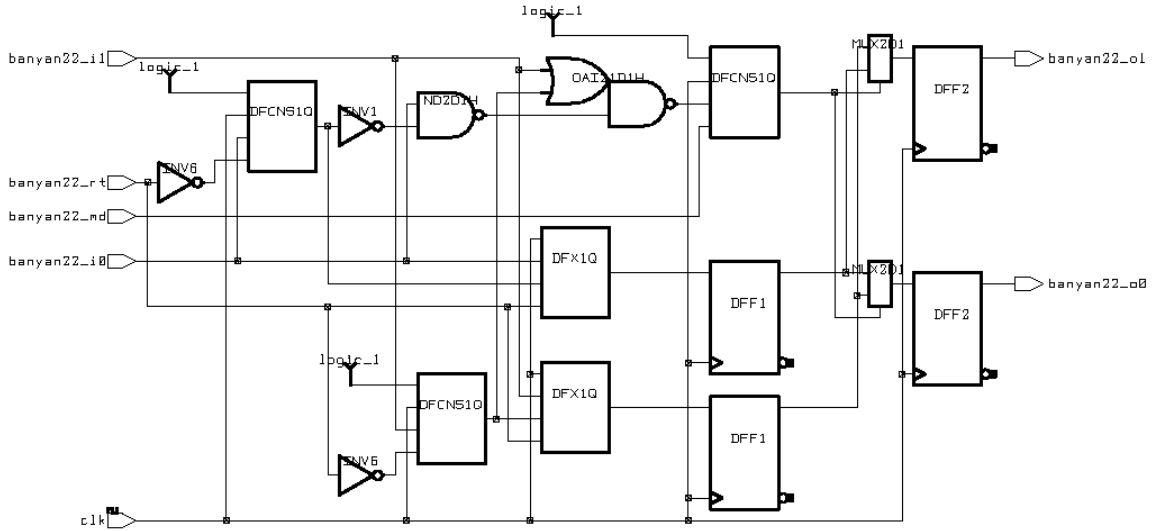


Figure 5-3: Structure of a Banyan Element

5.1.4 Trap

The Trap circuit is also a component of the StarBurst ATM chip [BCK2000]. It is a comparator circuit used to detect duplicated packets (packets having the same destination address), which are subsequently buffered or discarded. It is made up of smaller trap elements, which consist of mainly flip-flops and XOR gates. The structure of a trap element is shown in Figure 5-4. It uses 187 LUTs and 470 FFs.

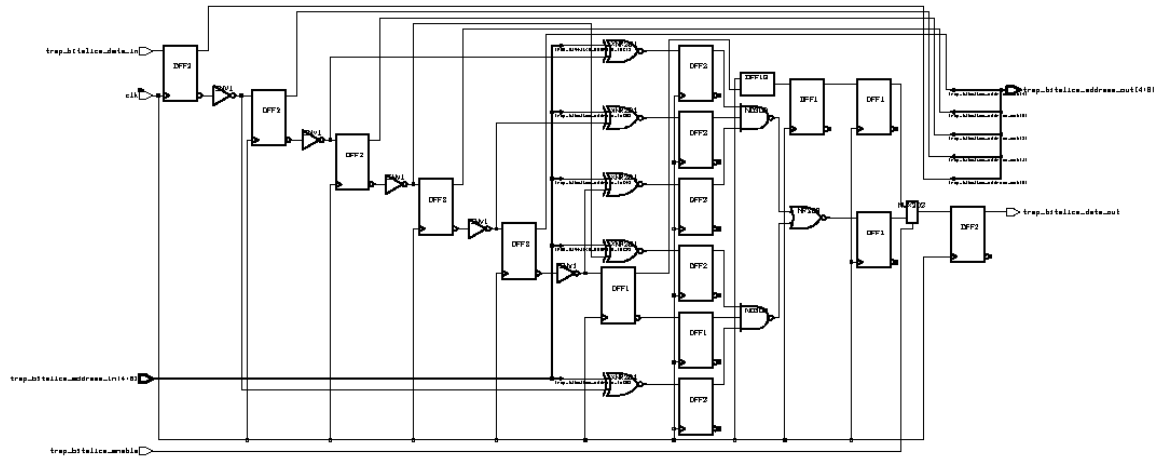


Figure 5-4: Structure of a Trap Element

5.1.5 Miim

The Miim circuit [Open2001] is an MII Management module of an Ethernet IP core obtained from OpenCores.org. The complete Ethernet IP core is designed for implementation of CSMA/CD LAN in accordance with the IEEE 802.3 standards. It uses 122 LUTs and 112 FFs.

5.1.6 Div

The Div circuit is an IP Core circuit generated by the Xilinx LogiCORE Pipelined Divider for Virtex Version 2.0 generator [Xili99]. It has unsigned 8-bit dividend and divisor with integer remainder. It has throughput of one division per clock cycle with a latency of eight clock cycles. It uses 87 LUTs and 255 FFs.

5.1.7 Dotproduct

The Dotproduct circuit computes the dot product of two 8-bit 3D vectors. It is a part of a 3D ray-tracing application under development at the University of Toronto [Fend2002]. It runs on the Transmogri-fier-3 (TM3) [GVR2000], a reconfigurable hardware prototyping system developed at the University of Toronto. It uses 243 LUTs and 178 FFs. The structure of the Dotproduct circuit is shown in Figure 5-5.

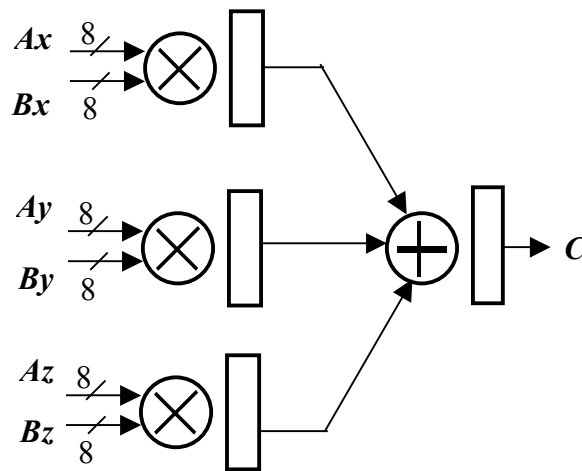


Figure 5-5: Structure of the Dotproduct circuit

5.1.8 Crossproduct

The Crossproduct circuit computes the cross product of two 4-bit 3D vectors. It is also a part of the 3D ray-tracing application [Fend2002]. It uses 129 LUTs and 126 FFs. The structure of the Crossproduct circuit is shown in Figure 5-6.

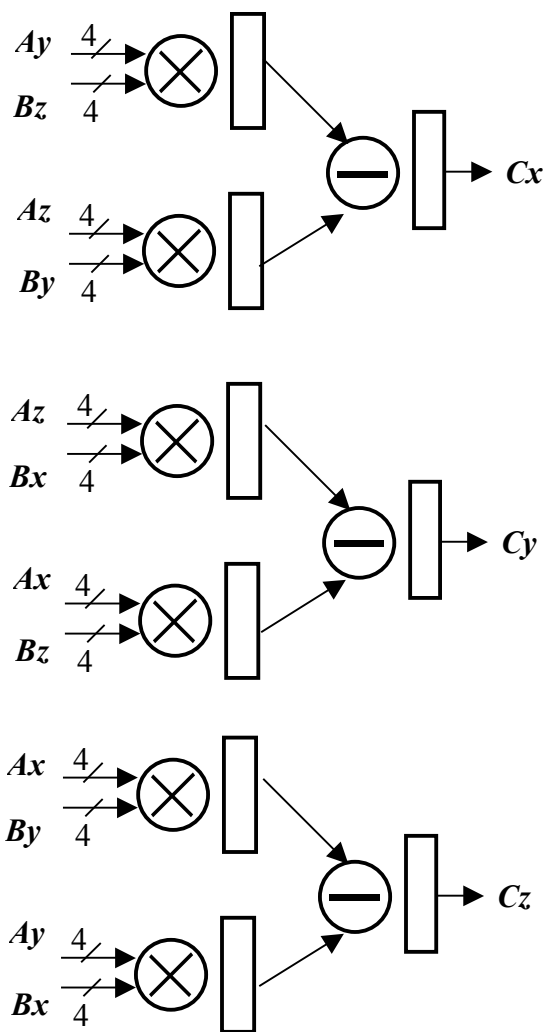


Figure 5-6: Structure of the Crossproduct circuit

5.2 Baseline Circuits Generation

To evaluate EVE, we need to obtain a set of good baseline results from an automatic push-button flow. We use the following state-of-the-art synthesis and placement and routing tools: Synplify Pro 6.20 [Synp2000] (which is generally accepted

as the preeminent synthesis tool for FPGAs) for logic synthesis, and Xilinx Foundation 3.1i [Xili2000] for mapping, placement and routing. During logic synthesis using Synplify Pro, features such as resource sharing, register retiming, and automated pipelining are enabled to produce the fastest logical designs possible. As an exception, the Div circuit does not require logic synthesis because it is directly generated from an IP Core netlist generator from Xilinx [Xili99]. It is placed and routed in the usual way using the Xilinx backend tools. The baseline results are obtained following the steps below:

The input is VHDL or Verilog code obtained as described above.

1. Synthesize the HDL code using Synplify Pro 6.2 with frequency s .
2. Place and route using Xilinx Foundation 3.3i Service Pack 7 tools
3. Obtain final circuit frequency from P&R reports.
4. Adjust frequency and repeat step (1) to (3), increasing s 10% each time until the best frequency is obtained.
5. Use the frequency obtained in (4), place and route again using the Multi-Pass Place&Route (MPPR) option for ten runs, and pick the best resulting design.

The options used to generate the baseline circuits are recorded in Table 5-1.

<p>Options used for Synplify Pro:</p> <ul style="list-style-type: none"> Max Fanout = 100 Disable I/O = on Pipelining = on Retiming = on FSM Compiler = on Resource Sharing = on <p>Options used for Xilinx backend tools:</p> <ul style="list-style-type: none"> P&R effort = 4 Trim unconnected logic = no Replicate logic = yes MPPR initial placement seed = 1 MPPR P&R passes = 10 MPPR save N Best = 1 <p>Frequency setting (for Synplify Pro & Xilinx backend):</p> <ul style="list-style-type: none"> Vision = 200MHz Batcher = 330MHz Banyan = 335MHz Trap = 400MHz Miim = 165MHz Div = 220MHz (for Xilinx backend only) Dotproduct = 150MHz Crossproduct = 220MHz

Table 5-1: Options used for Baseline Circuits generation

5.3 Results: Using TEMP Mode Only

The eight baseline circuits are loaded into EVE. The author spent on average about two hours to improve timing on each circuit. The machine used is a 1GHz Pentium-III PC with 512MB ram running Windows2000 and Xilinx Foundations 3.3isp7.

When we use the Timing Exact Microscopic Placement (TEMP) mode, we fix the size of the circuit boundary in which we can move circuit elements. This ensures that we do not improve circuit speed at the expense of increase in occupied chip area. The results are summarized in the following table:

Circuit	# LUTs	# FFs	Period (ns)	Freq (MHz)	New Freq (MHz)	% Change
Vision	142	241	4.92	203.3	224.8	10.6%
Batcher	252	455	3.06	326.8	380.1	16.3%
Banyan	165	311	2.94	340.7	395.3	16.0%
Trap	187	470	2.45	408.3	460.4	12.8%
Miim	122	112	6.16	162.4	168.5	3.8%
Div	87	255	4.65	215.1	229.6	6.7%
Dotproduct	243	178	6.74	148.4	173.3	16.8%
Crossproduct	129	126	4.54	220.1	261.4	18.8%
Average	166	269	4.43	238.7	268.8	12.7%

Table 5-2: Results for Using the TEMP Mode

On average, the circuit speed improved by 12.71% over the baseline. Below we discuss the properties of each circuit and the nature of the operations we performed using EVE to improve circuit performance.

1. Vision

Since it is an FIR filter, the Vision circuit is very well pipelined, and the initial placement of the circuit occupies an area big enough to allow easy placement modifications. Examining the initial delay profile (which EVE prints out in a command window), we observed that the k-most critical paths have a bimodal distribution, suggesting that focusing on improving the placement of the critical

path elements can result in significant gains in speed. The author thus focused on improving the placement of circuit element on the critical path, or sometimes the k-most critical paths (by setting a slightly tighter timing budget, exposing more nets in timing violations). Sometimes, when the critical path is in a carry chain, the reroute operation is observed to be able to relieve routing congestions effectively. The circuit timing improved 10.58% from 203.3MHz (See Figure A-1) to 224.8MHz (See Figure A-2).

2. Batcher

The Batcher circuit occupies a large area on the chip, so the placement of the design can be changed easily. The circuit is highly pipelined by design with a starting speed over 300MHz. It is interesting to note that even for circuits operating at such a high speed, their placement and packing can be improved further over the result generated with an automatic approach. It shows that microscopic placement is very important for a highly pipelined design. The circuit timing improved 16.31% from 326.8MHz (See Figure A-3) to 380.1MHz (See Figure A-4).

3. Banyan

The Banyan circuit has a high baseline circuit speed of 340MHz. To achieve speeds close to 400MHz, which approaches the physical speed limit of the FPGA, we need to place circuit elements no more than one CLB apart horizontally on the chip. This circuit orientation guides routing to use extremely fast nearest neighbour connections which are present across neighbouring horizontal CLBs, as

described in Section 2.1.2. We used the Xilinx FPGA Editor to view the result of routing of the modified circuit, and saw that in most cases, the router was able to make use of the nearest neighbour connections [Roop2002]. This observation suggests that floorplanning or placement editing tools should be more intelligent to inform the user of any special architectural features of a chip which can provide the necessary speed up to meet the extremely high timing goal. The circuit timing improved 16.03% from 340.7MHz (See Figure A-5) to 395.3MHz (See Figure A-6).

4. Trap

The Trap circuit has the highest speed among all experimental circuits because it contains mostly of flip-flops, enabling it to reach over 400MHz using an automatic approach. However, we found out that many of the circuit elements are actually not optimally packed together in the same logic slice, and we were able to improve the circuit speed further to over 460MHz by doing packing/unpacking operations. This observation highlights the importance of being able to pack and unpack during placement and routing. Also, the operation of grouping multiple timing-critical objects together and rerouting them as a group proved to be effective. The circuit timing improved 12.76% from 408.3MHz (See Figure A-7) to 460.4MHz (See Figure A-8)!

5. Miim

Although we tried very hard to improve the speed of the Miim circuit, we could only improve it by 3.76%. The critical path of the Miim circuit lies in a single

carry chain which loops back to itself tightly. Without breaking up the carry chain, it is not easy to improve timing by just changing the placement. The circuit timing improved 3.76% from 162.4MHz (See Figure A-9) to 229.6MHz (See Figure A-10).

6. Div

The Div circuit is a Xilinx CoreGen generated pipelined divider [Xili99] circuit, which does not have any Relational Placed Macros (RPMs). RPMs are used to fix the relative placement of circuit components within an IP core. Editing the placement of the design can only improve by 6.7%. The critical path has a carry chain feeding into another carry chain. Because we do not allow any increase in total occupied chip area, the restrictive carry chains become the bottleneck for any further speed increase. The circuit timing improved 6.74% from 215.1MHz (See Figure A-11) to 229.6MHz (See Figure A-12).

7. Dotproduct

The Dotproduct circuit is dominated by a large number of carry chains employed for multiplications. The initial placement was not very good, because carry chains were not aligned correctly for signal to flow through naturally. We thus rearranged the carry chains manually by simply examining the signal flow of the nets connecting the carry chains. We achieve a good improvement in speed as a result. This suggests that we can automate floorplanning and placement by examining the signal flow of various circuit components, and it is especially

useful for circuits with large number of carry chains. The circuit timing improved by 16.78%, from 148.4MHz (See Figure A-13) to 173.3MHz (See Figure A-14).

8. Crossproduct

The circuit contains 4-bit multipliers synthesized into short carry chains. Again, as observed in the Dotproduct circuit above, the signal flow of the carry chains is poor. The initial circuit placement has a critical path spanning nine CLBs horizontally. Subsequent rearrangement of carry chains order improved circuit speed by 18.76% from 220.1MHz (See Figure A-15) to 261.4MHz (See Figure A-16).

In this section, we have shown the effectiveness of EVE's Timing Exact Microscopic Placement (TEMP) mode to further improve on high circuit speeds. We make the following summary observations:

1. Having enough unoccupied space around critical path elements of the circuit makes placement editing easier.
2. Showing a dynamic delay distribution of the circuit after each user move helps the user identify circuit improvement opportunities and can drive automatic placement algorithms.
3. Focusing on improving delay on the critical path or the k-most critical paths (by setting a certain timing budget, and observe circuit elements violating the budget) prove effective.

4. Partial re-routing of timing-critical regions of the circuit is effective because routing resources in the surrounding area of critical paths can be freed up, and more critical nets can be reassigned faster routing resources.
5. The more pipeline stages a circuit has, the easier the placement-editing task will be.
6. Floorplanning or placement editing tools should inform the user of any high speed routing resources available in the chip, so the user can make better micro-placement decisions.
7. The ability to do pack and unpack logic slices during placement and routing is essential.
8. An automatic floorplanning algorithm based on signal flow analysis should help timing. This observation has been made by FPGA design experts [Mani2000].

5.4 Results: Using Both TEMP and Pipelining Modes

In this section we present results obtained by using both the TEMP and pipelining modes of EVE to improve circuit speed. We can only successfully obtain results for two circuits: *Div* and *Mult*. The *Div* circuit is circuit number six we described in the previous section. The *Mult* circuit is a new circuit (number nine) written to test EVE's pipelining ability. It is a non-pipelined four bit by four-bit multiplier built using full and half adder blocks. The circuit is synthesized using the procedure described in Section 5.2, except that we does not turn on the pipelining and retiming features of Synplify Pro. The

resulting circuit does not contain any carry chains, so it is highly pipeline-able by design. Results for the Vision and Miim circuits are not available because the critical paths are inside loops, which cannot be pipelined. Results for the Batcher, Banyan and Trap circuits are not gathered because the circuits are already sufficiently pipelined. Results for the Dotproduct and Crossproduct circuits are not available due to software instability. Here is the summary of results:

Circuit	# LUTs	# FFs	# FFs added	Freq (MHz)	New Freq (MHz)	% Change
Vision	142	241		224.8	N/A : critical path in loop	
Batcher	252	455		380.1	N/A : already well pipelined	
Banyan	165	311		395.3	N/A : already well pipelined	
Trap	187	470		460.4	N/A : already well pipelined	
Miim	122	112		168.5	N/A : critical path in loop	
Div	87	255	66	229.6	237.7	3.5%
Dotproduct	243	178		173.3	N/A : due to tool instability	
Crossproduct	129	126		261.4	N/A : due to tool instability	
Mult	39	23	38	123.1	175.1	42.2%

Table 5-3: Results for Using both TEMP and Pipelining Modes

These results are gathered after one stage of pipeline insertion. For the already well-pipelined Div circuit, minimal performance increase after the pipelining operation is expected. For the Mult circuit, however, we achieve a performance increase of 42.24%. It proves that the pipelining feature is functional. However, pipelining at the logic synthesis level could probably have increased the speed of the Mult circuit to about 220MHz.

Pipelining at the logic synthesis level is still the preferred choice over pipelining at the physical level. But for extremely high-speed circuits, pipelining at the physical

level may be the only way to obtain accurate post-placement/routing delay information for performing optimal pipelining operation. The current user interface of EVE's pipelining mode is very limited. It can demonstrate basic ideas for pipelining at the physical level, but the actual pipelining operation is not easy to do. Future projects which look at the Synthesis stage in the Event Horizon methodology may make better use of the pipelining feature that EVE currently offers.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we propose a methodology for manual packing, placement and pipelining, with the aim of aiding designers seeking very high-speed implementation of circuits. We implement the method in a manual editor called EVE.

EVE provides an intuitive GUI interface, which can perform powerful operations such as packing/unpacking, placement, and routing operations. It integrates tightly with the Xilinx backend tools to allow editing of real commercial FPGA circuits based on the Xilinx Virtex-E architecture. It gives the user full low-level control of the circuit, and it provides instant real timing feedback while during placement editing and pipelining operations. It is timing-budget aware and it provides useful features to help designers meet the timing goal. Experimental results show that EVE is capable of improving speed

of real circuits by up to 19%, and we show that it improves a group of eight circuits on average by 12.7%.

EVE also serves as a valuable tool to help researchers understand FPGA architectures, and devise new placement and routing algorithms. The pipelining mode in EVE demonstrates important ideas involved in pipelining at the physical level. EVE will serve as a good reference CAD tool for further research into the area of high-speed manual-assisted design tools.

6.2 Future Work

EVE is the first attempt to help circuit designers building extremely fast circuits on FPGA. It does so by looking at the packing/unpacking, placement, and pipelining aspects of the Event Horizon methodology. Future work can expand on the ideas stated in this thesis to study the problem of:

1. Design creation and synthesis with tight timing budget in mind, and submitting logic elements to placement and routing tools in an optimal sequence that will allow the placement and routing tool to microscopically construct a high-speed circuit.
2. Extending EVE to support more advanced architecture such as the Xilinx Virtex-II, and other advanced commercial architectures.

3. Extending EVE to support more functionality such as the ability to break carry chains, or inform the user of any special fast routing resource that can improve the critical delay.
4. Improving the speed of the delay database by deriving routing delay models, instead of obtaining delay information from the backend.
5. Devise an automatic floorplanning algorithm that EVE can employ.
6. Automating the manual optimizations we performed while using EVE.

References

- [BCK2000] P. Bade, W. Chow, P. Kundarewich, N. Saniei, A. Wang, “Starburst ATM Chip project at University of Toronto”, October 2000. (Available from <http://www.eecg.utoronto.ca/~wangk/report.ps>)
- [Betz98] V. Betz, “EasyGL: An easy-to-use graphics interface for programming X11 and PostScript displays,” (Available from <http://www.eecg.utoronto.ca/~vaughn/easygl/easygl.html>)
- [BH98] P. Bellows and B. Hutchings, “JHDL --- an HDL for reconfigurable systems,” In K. Pocek and J. Arnold, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 175--184, Napa, CA, April 1998. IEEE Computer Society, IEEE Computer Society Press.
- [BR97] V. Betz and J. Rose, “VPR: A New Packing, Placement and Routing Tool for FPGA Research,” *Int. Workshop on Field-Programmable Logic and Applications*, 1997, pp. 213-222.
- [BRM99] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, February 1999.

- [Chow2001] W. Chow, "EasyGL For Windows," 2001. (Available from <http://www.eecg.utoronto.ca/~choww/easygl.html>)
- [CW99] J. Cong and C. Wu, "Optimal FPGA Mapping and Retiming with Efficient Initial State Computation," *IEEE Trans. on Computer-Aided Design*, pp 1595 -1607, November 1999.
- [Elmo48] W. Elmore, "The Transient Response of Damped Linear Networks," *Journal of Applied Physics*, Vol. 19, pp. 55 - 63, Jan 1948.
- [Fend2002] J. Fender, University of Toronto, Bachelor's Thesis in progress, working title: "A 3D Ray Tracing Engine on TM-3", 2002.
- [GLS99] S. A. Guccione, D. Levi, and P. Sundararajan, "JBits: A Java-based interface for reconfigurable computing," in *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, (Laurel, MD), September 1999.
- [GVR2000] D. Galloway, M. Van Ierssel, J. Rose, P. Chow, "Transmogrieffier-3 project," 2000. (Available from <http://www.eecg.toronto.edu/~tm3/>).
- [HSC83] R. Hitchcock, G. Smith and D. Cheng, "Timing Analysis of Computer-Hardware," *IBM Journal of Research and Development*, Jan. 1983, pp. 100-105.
- [KN91] E. Koutsofios and S.C. North, "Drawing graphs with dot," Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, September 1991.

- [LS83] C. E. Leiserson and J. B. Saxe, "Optimizing Synchronous Circuitry and Retiming", *Proceedings of the 3rd Caltech Conference on VLSI*, pages 5-35, 1983.
- [Mani2000] T. Maniwa, "FPGA 2000 Panel," *ISD Magazine*, February 2000. (available at <http://www.isdmag.com/articles/fpga0002.html>).
- [MR2000] R. McCready, J. Rose, "Real-Time Face Detection on a Configurable Hardware System," *FPL 2000*, pp 157-162, August 2000.
- [Nage75] L. Nagel, "SPICE: A Computer Program to Simulate Semiconductor Circuits", Berkeley, University of California, Electronic Research Laboratory, Technical Report ERL - M 520, 1975
- [OHM84] J. Ousterhout, G. Hamachi, R. Mayo, W. Scott, G. Taylor, "Magic: A VLSI layout system," in *Proc. of 21st Design Automation Conf.*, pp. 152-159, 1984
- [Open2001] OpenCores.org, "Ethernet MAC 10/100 Mbps project," March 2001. (available at <http://www.opencores.org/cores/ethmac/>).
- [Roop2002] A. Roopchansingh, University of Toronto, Master's Thesis in progress, working title: "Research on Nearest Neighbour Connections", 2002.
- [RPH83] J. Rubinstein, P. Penfield and M. Horowitz, "Signal Delay in RC Tree Networks," *IEEE Trans. On CAD*, 1983, pp. 202-211
- [Rubi83] S. Rubin, "An Integrated Aid for Top-Down Electrical Design," *VLSI '83* (Anceau and Aas, eds), North Holland, Amsterdam, pp.63-72, August 1983

- [SH89] A. Salz and M. A. Horowitz, "IRSIM: An incremental MOS switch-level simulator," in *Proc. Design Automation Conf.*, June 1989, pp.173-178.
- [SM89] P. Stanford and P. Mancuso, "EDIF Electronic Design Interchange Format, Reference Manual for Version 2 0 0," Electronic Industries Association, Washington D.C., 1989
- [Syno2000] Synopsys Inc, "FPGA Express 3.5," 2000. (at http://www.synopsys.com/products/fpga/fpga_express.html)
- [Synp2000] Synplicity, Inc, "Synplify Pro 6.20," 2000. (Available from http://www.synplicity.com/literature/pdf/SynPro_datasheet.pdf).
- [Term83] C. J. Terman, "RSIM—A logic-level timing simulator," in *Proc. Int. Conf. Computer Design*, Oct. 1983, pp. 437-440.
- [VonH97] B. Von Herzen. Signal processing at 250 MHz using high-performance FPGA's. In *Proc. ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays (FPGA'97)*, pages 62-68, 1997.
- [VonH97a] B. Von Herzen, "Signal Processing at 250 MHz Using High-Performance FPGA's," in *IEEE Trans. on VLSI Systems*, Vol 6, No.2, pp. 238-246, June 1998.
- [Xili99] Xilinx Corporation, "Pipelined Divider Core", May 1999. (Available from <http://www.xilinx.com/dsp/docs/pipediv.pdf>).
- [Xili2000] Xilinx Corporation, The Xilinx Foundation Series 3.1, 2000. (Available from <http://www.xilinx.com>).

- [Xili2001] Xilinx Corporation, “Floorplanner Guide, V3.1i,” 2000 (Available from http://toolbox.xilinx.com/docsan/3_1i/pdf/docs/flr/flr.pdf.)
- [Xili2001a] Xilinx Corporation, “FPGA Editor Guide, V3.1i,” 2000 (Available from http://toolbox.xilinx.com/docsan/3_1i/pdf/docs/fpg/fpg.pdf.)
- [Xili2001b] Xilinx Corporation, “Virtex-E 1.8V FPGA Family: Detailed Functional Description,” 2001 (Available from <http://www.xilinx.com/partinfo/ds022-2.pdf>.)

Appendix A

Screen Captures of Experimental Circuits

This chapter contains screen captures for experimental circuits described in Chapter 5. For each of the eight circuits described in Section 5.3, a screen capture is obtained in the floorplanning mode both before and after about two hours of manual floorplanning effort on the circuit using EVE by the author. We zoom on the area occupied by the circuits to make the pictures easier to read. The circuit speed as reported by EVE can be seen in the status bar at the bottom of each picture, and the most critical path(s) are highlighted. Circuits after floorplanning and pipelining operations are also shown.

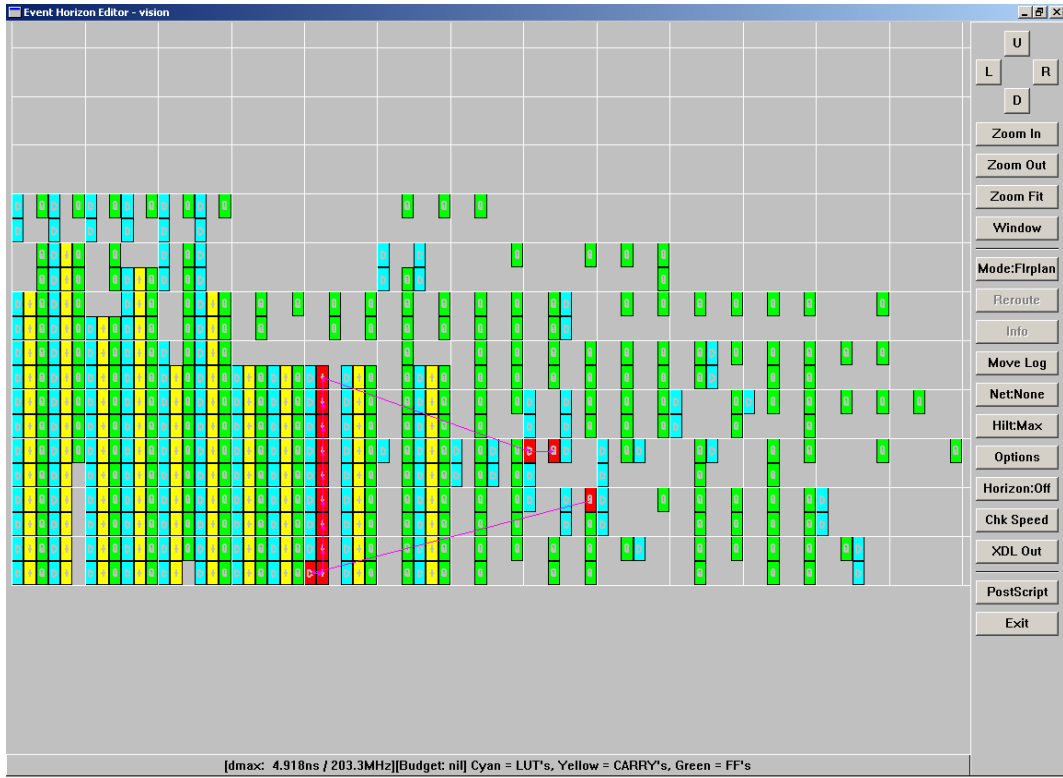


Figure A-1: *Vision* circuit before floorplanning

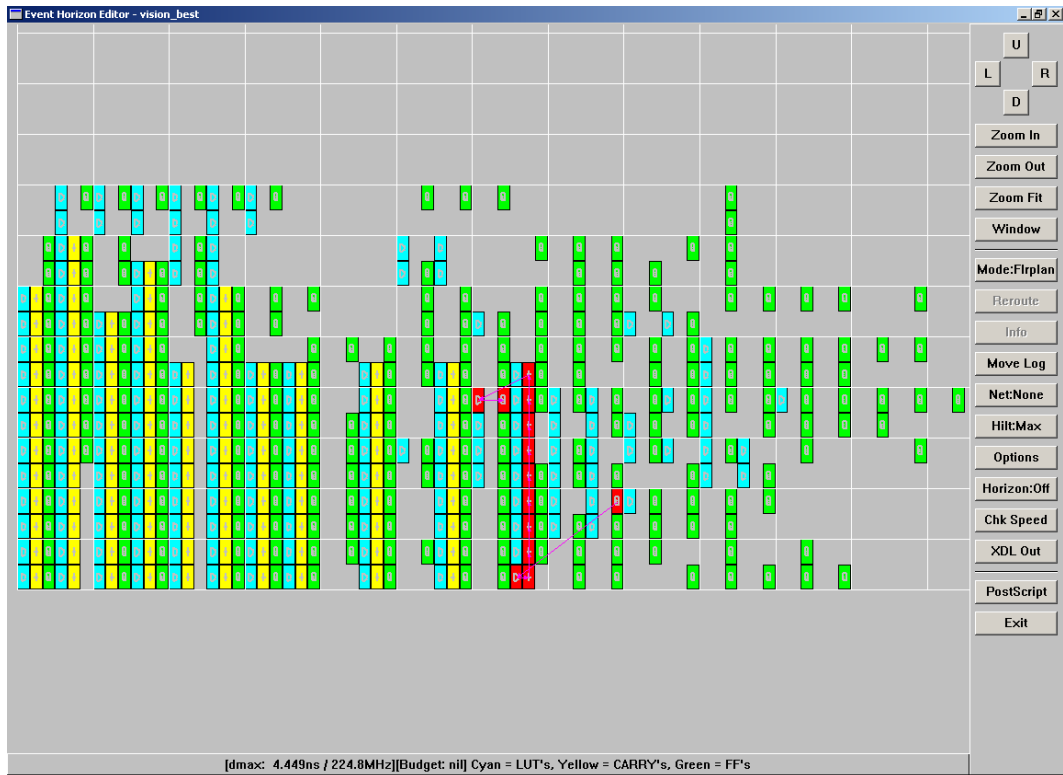


Figure A-2: *Vision* circuit after floorplanning

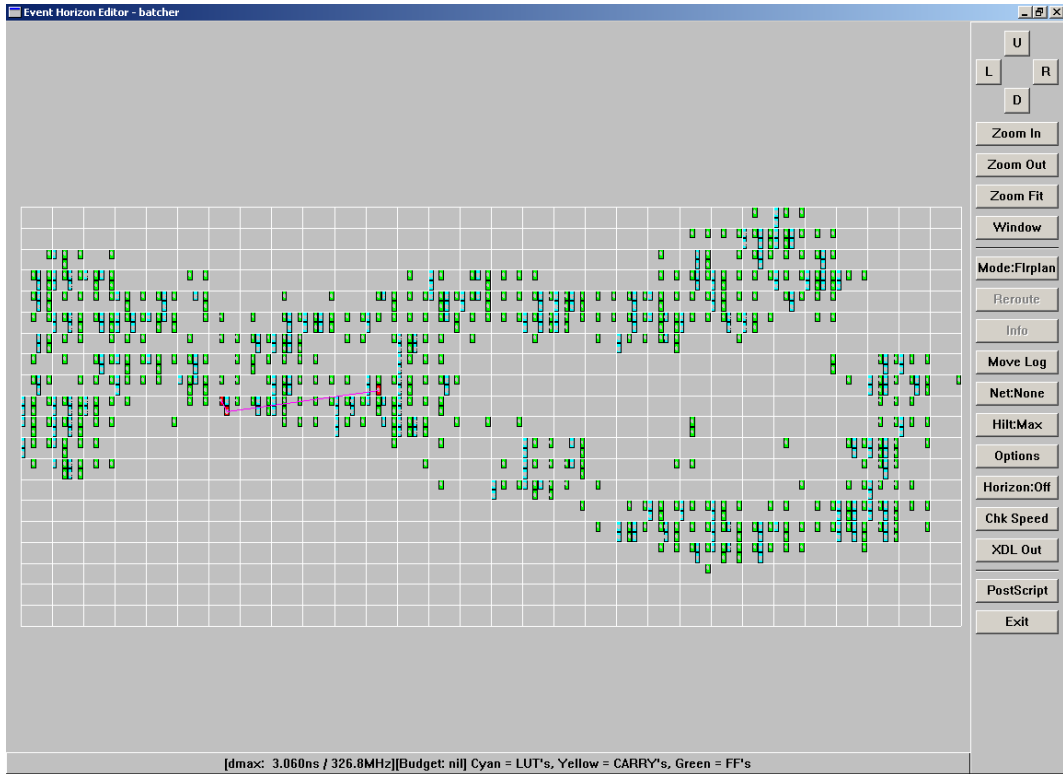


Figure A-3: *Batcher* circuit before floorplanning

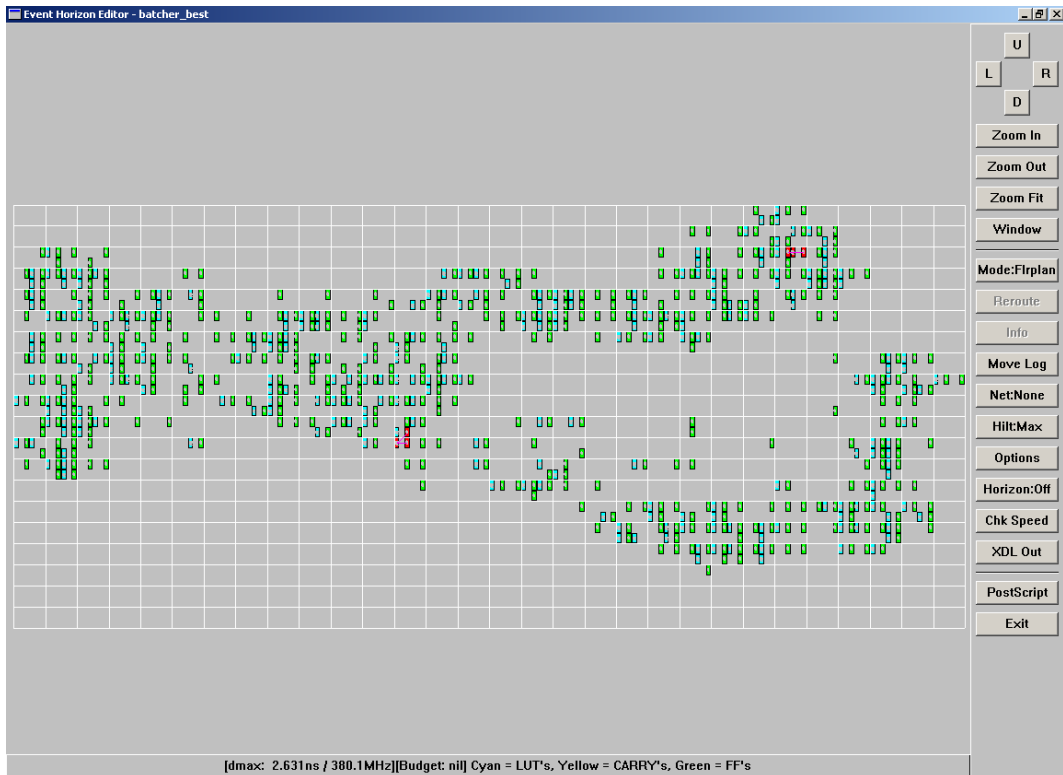


Figure A-4: *Batcher* circuit after floorplanning

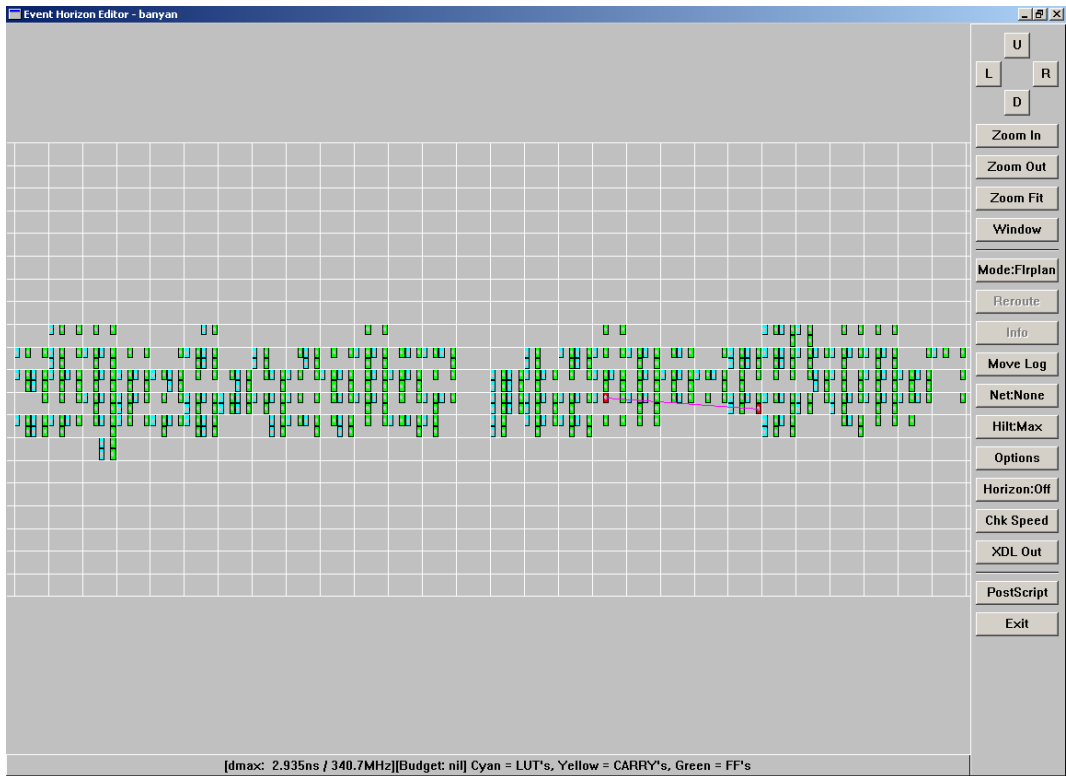


Figure A-5: *Banyan* circuit before floorplanning

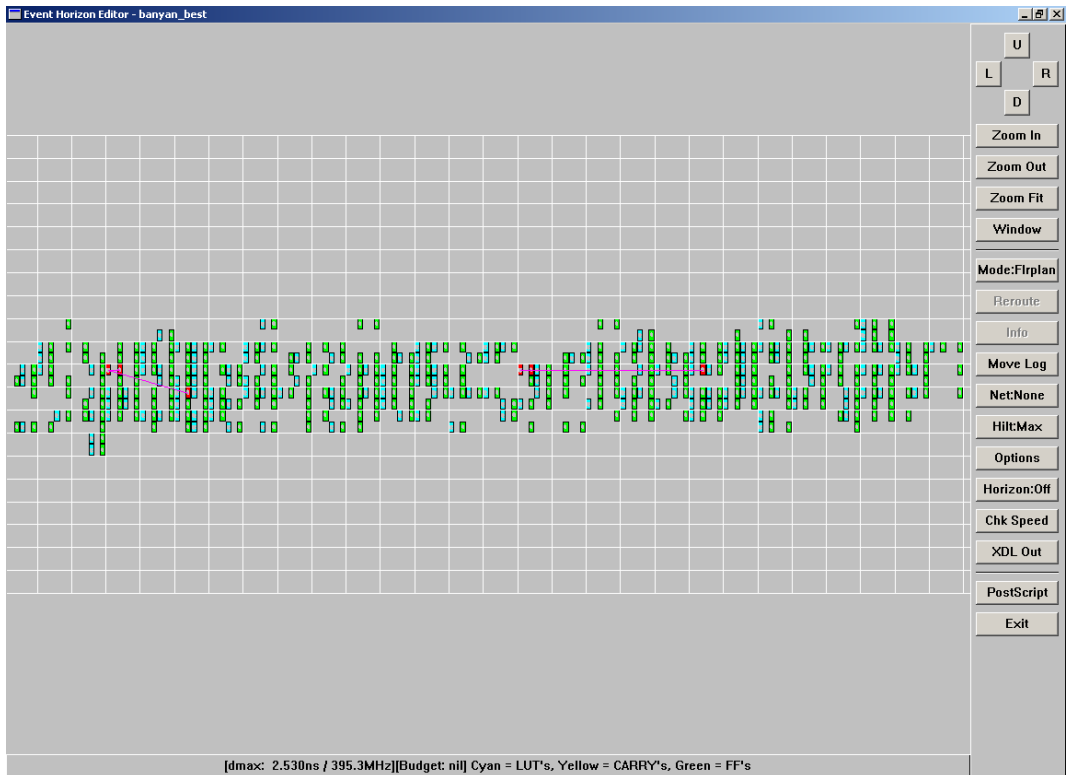


Figure A-6: *Banyan* circuit after floorplanning

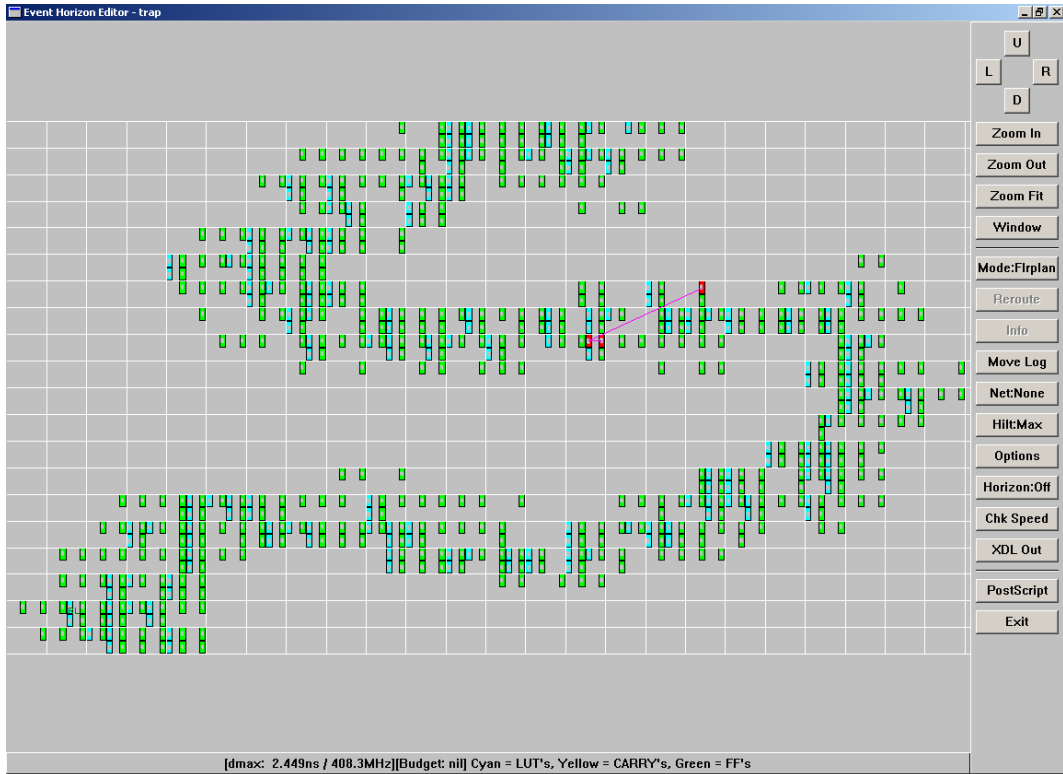


Figure A-7: Trap circuit before floorplanning

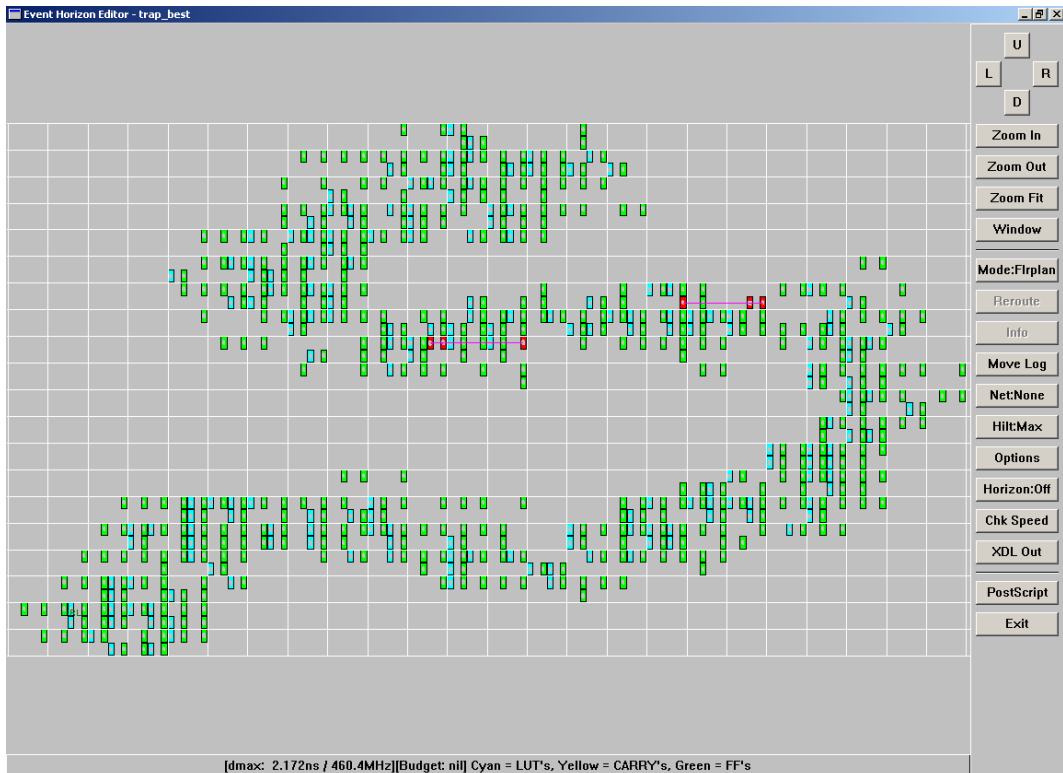


Figure A-8: Trap circuit after floorplanning

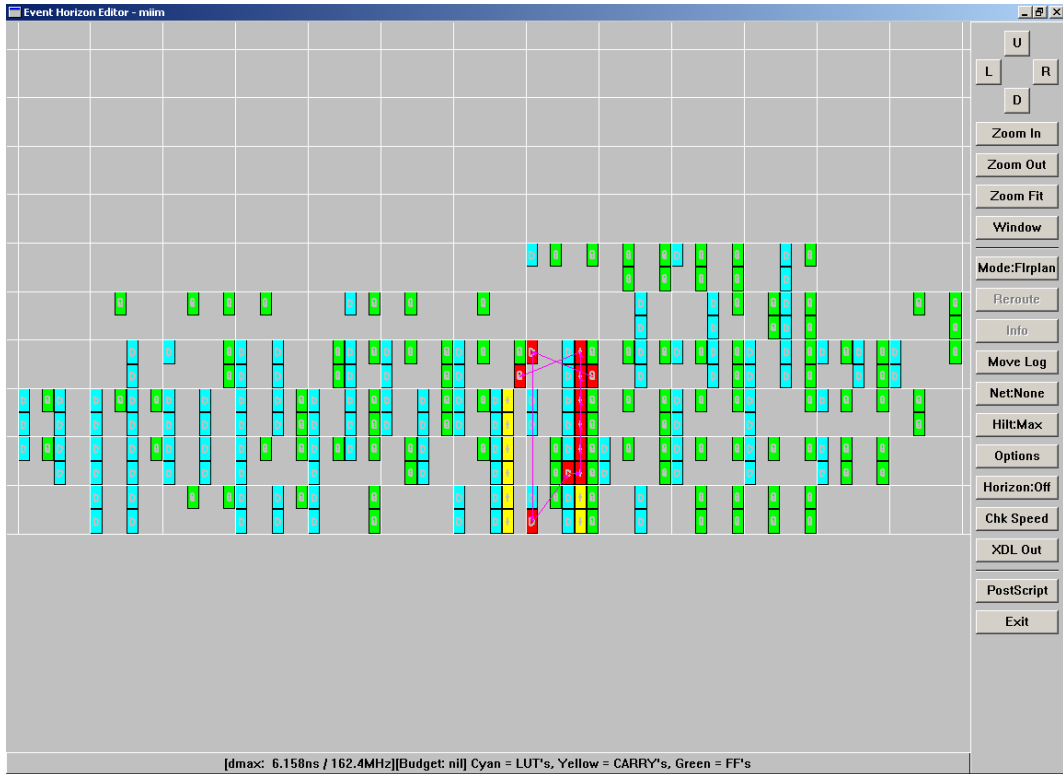


Figure A-9: *Mim* circuit before floorplanning

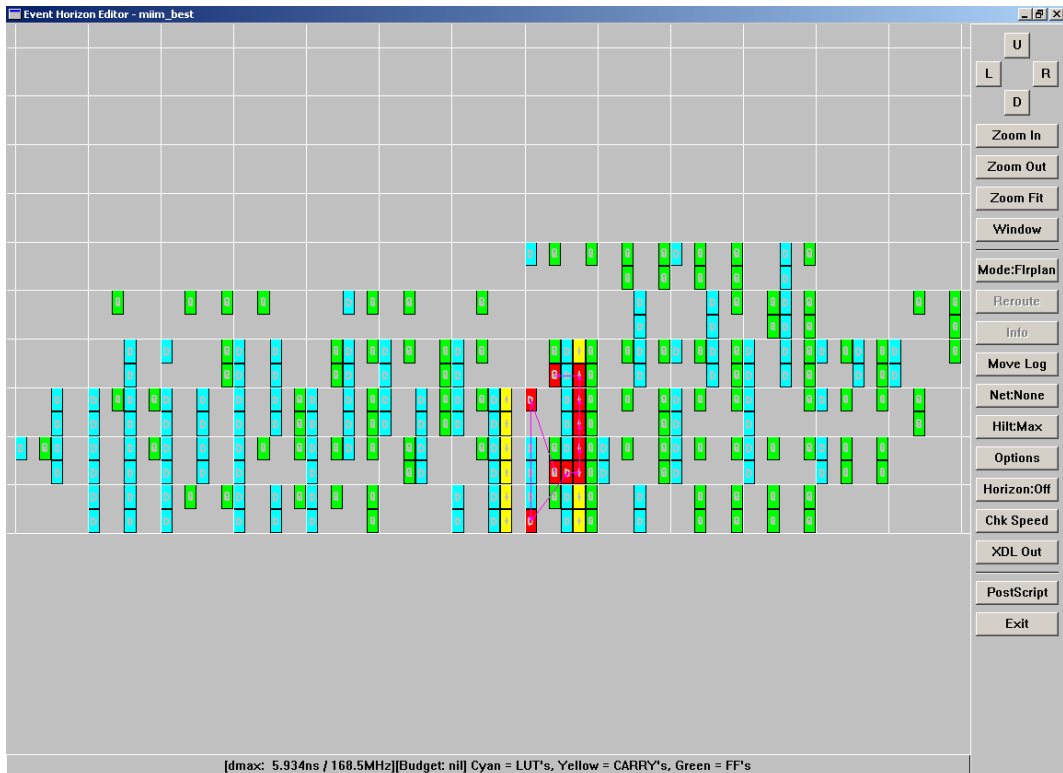


Figure A-10: *Mim* circuit after floorplanning

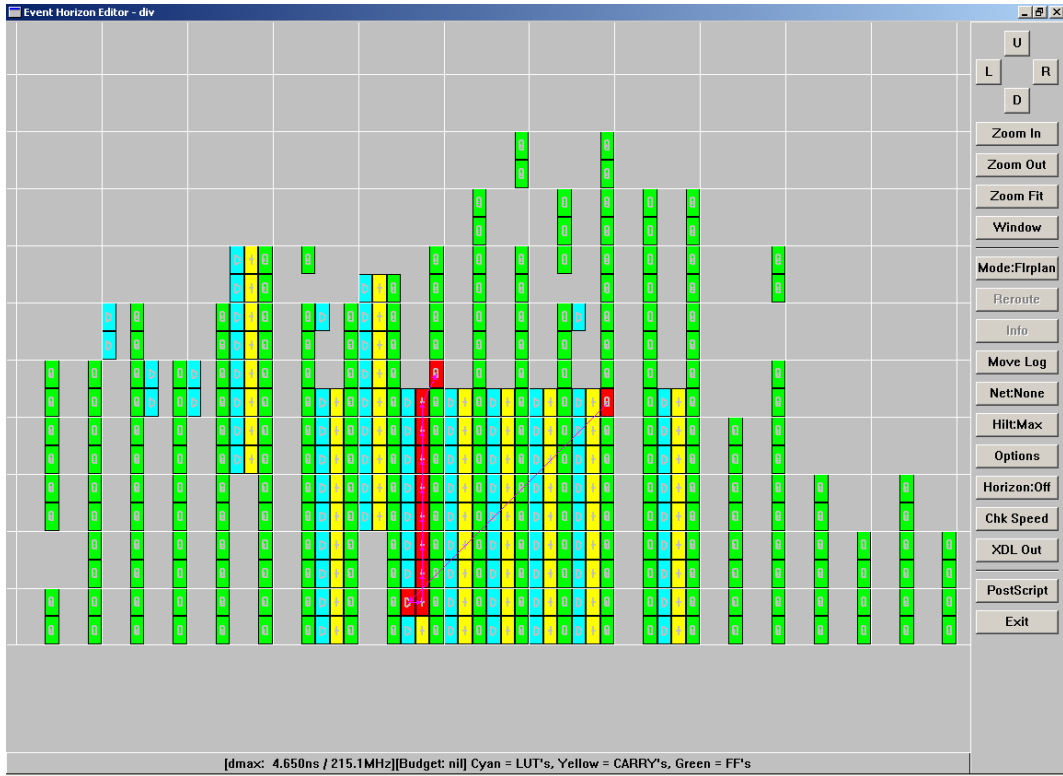


Figure A-11: Div circuit before floorplanning

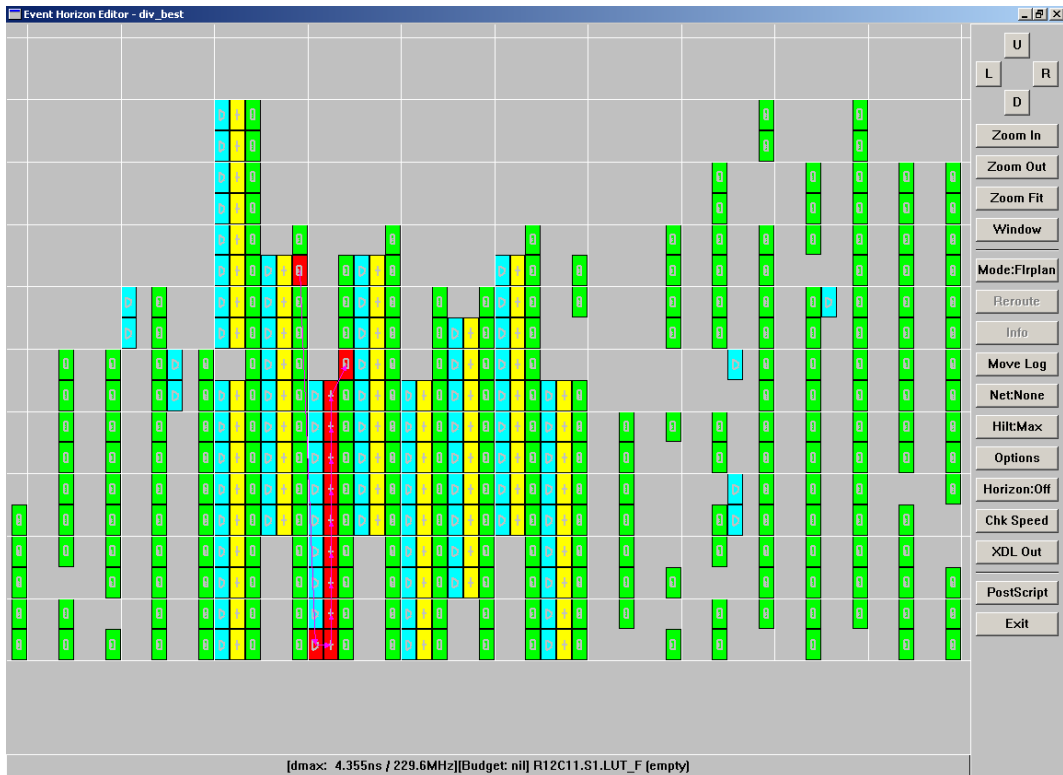


Figure A-12: Div circuit after floorplanning

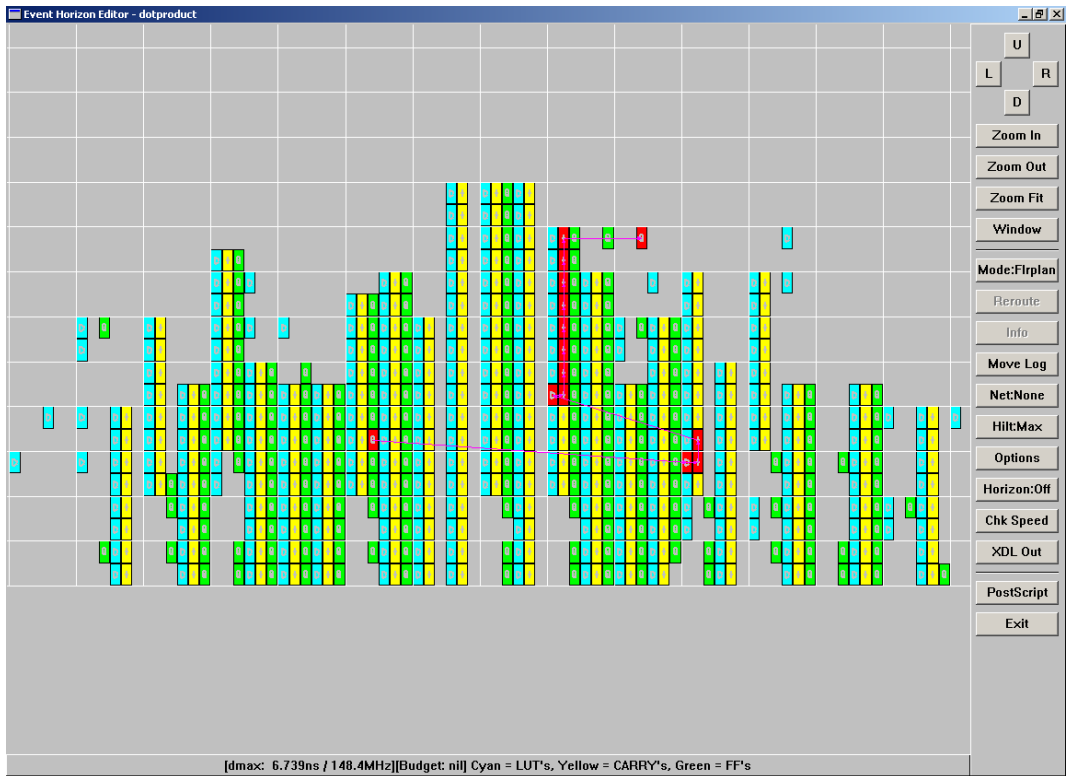


Figure A-13: *Dotproduct* circuit before floorplanning

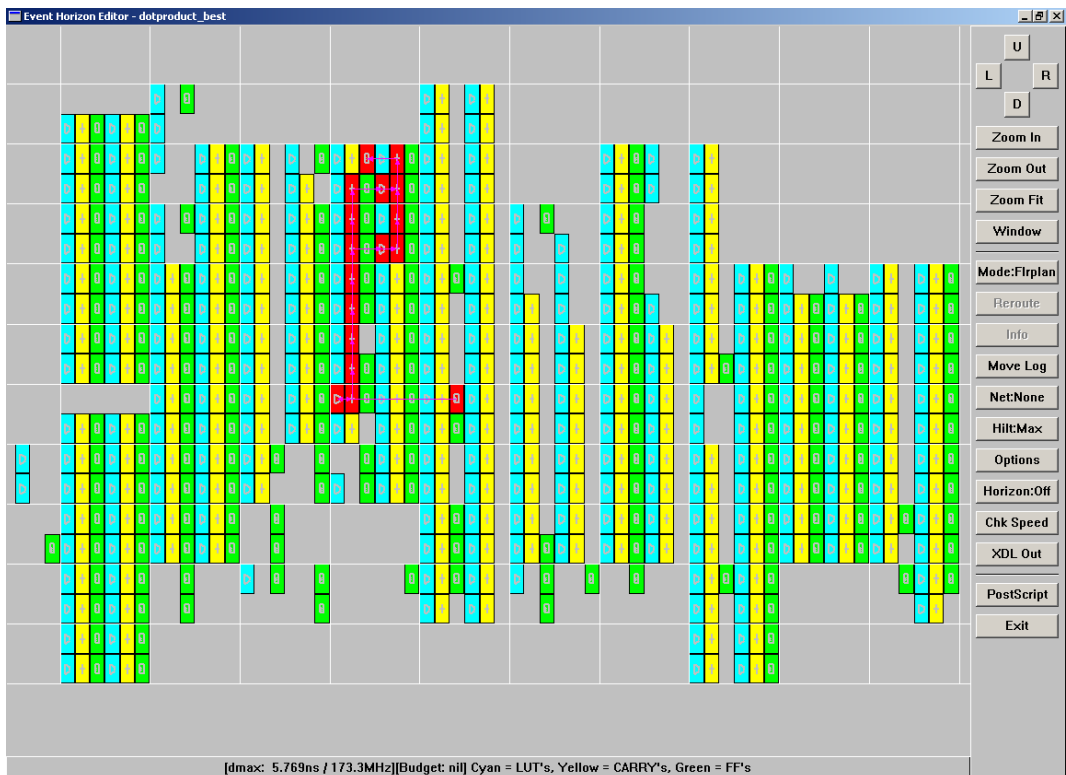


Figure A-14: *Dotproduct* circuit after floorplanning

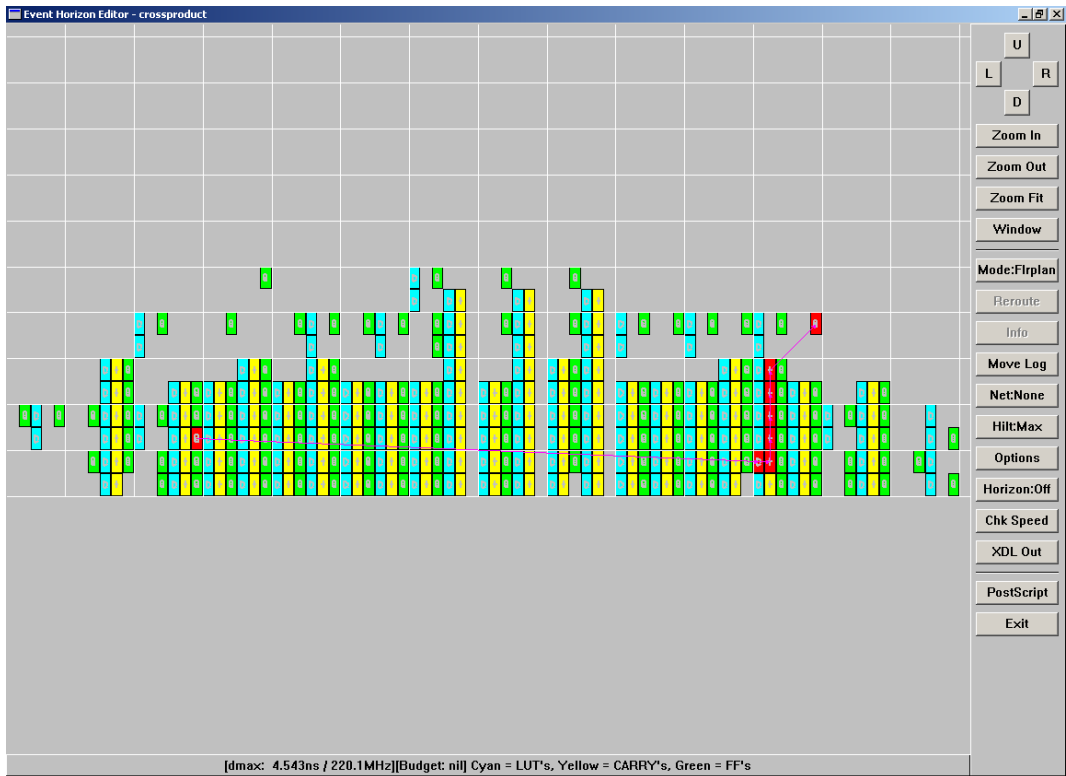


Figure A-15: *Crossproduct* circuit before floorplanning

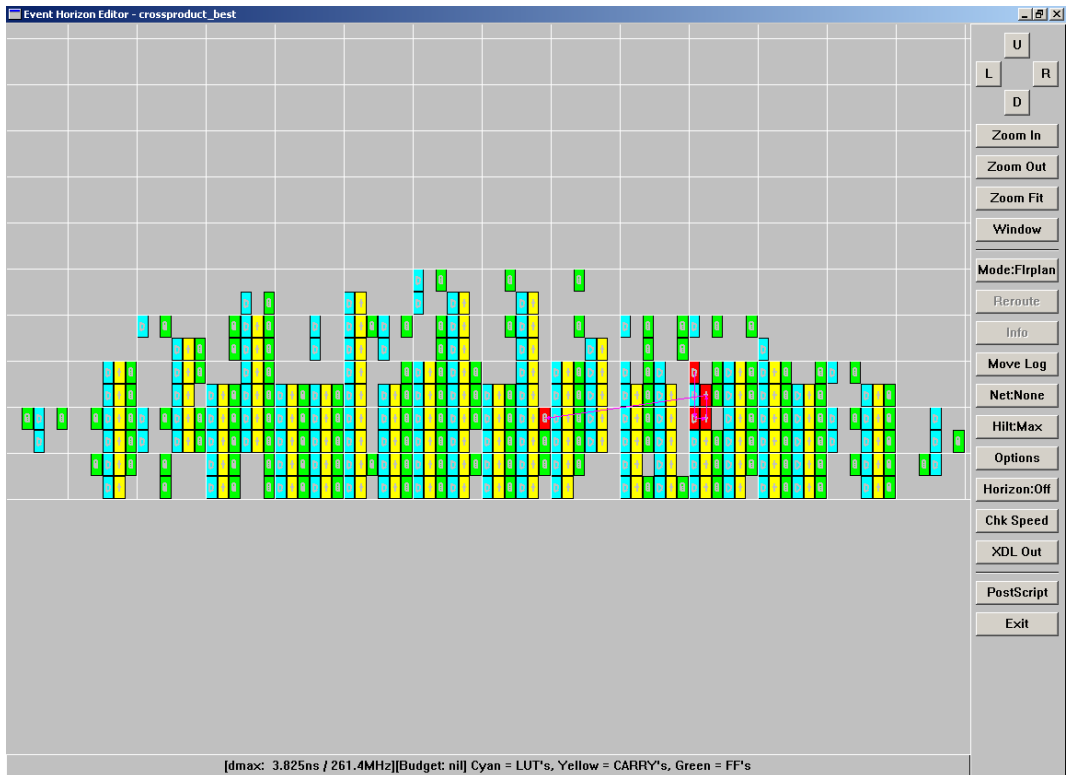


Figure A-16: *Crossproduct* circuit after floorplanning

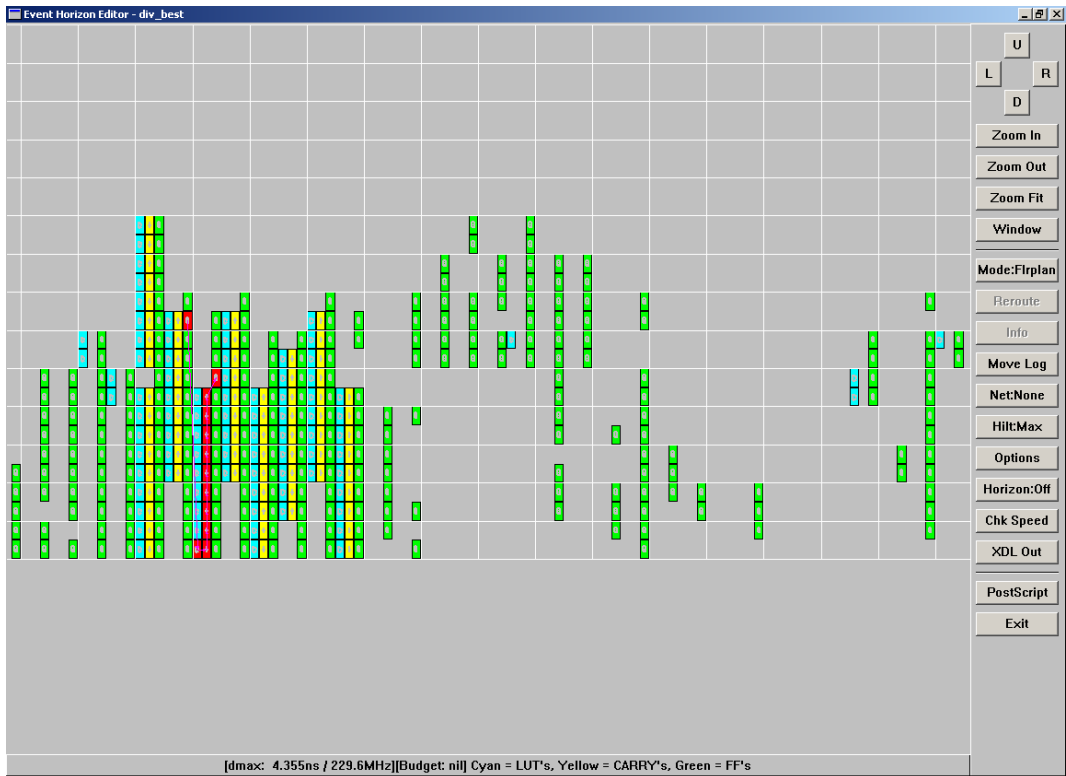


Figure A-17: *Div* circuit before pipelining

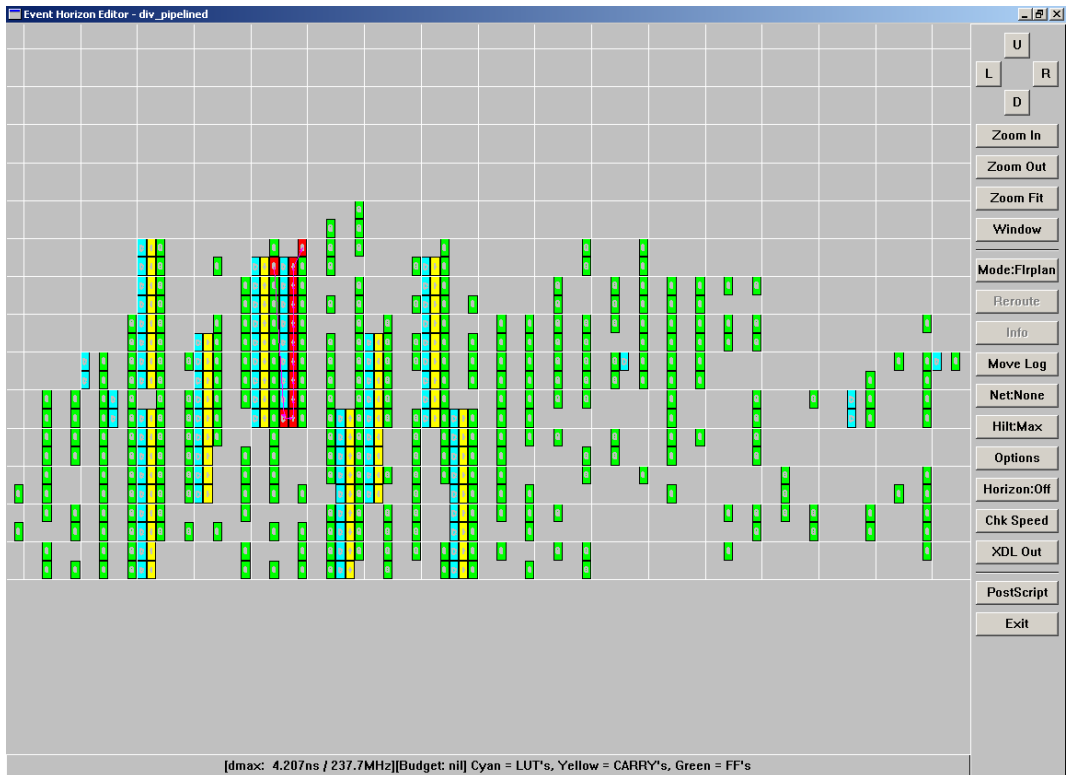


Figure A-18: *Div* circuit before pipelining

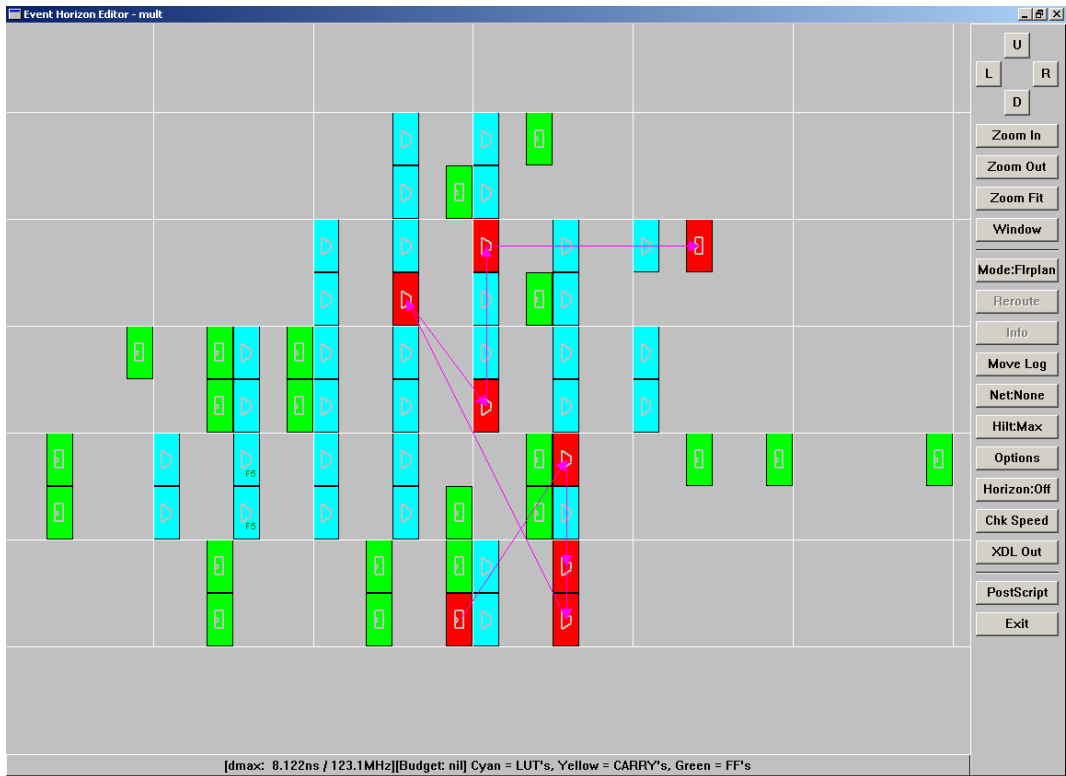


Figure A-19: *Mult* circuit before pipelining

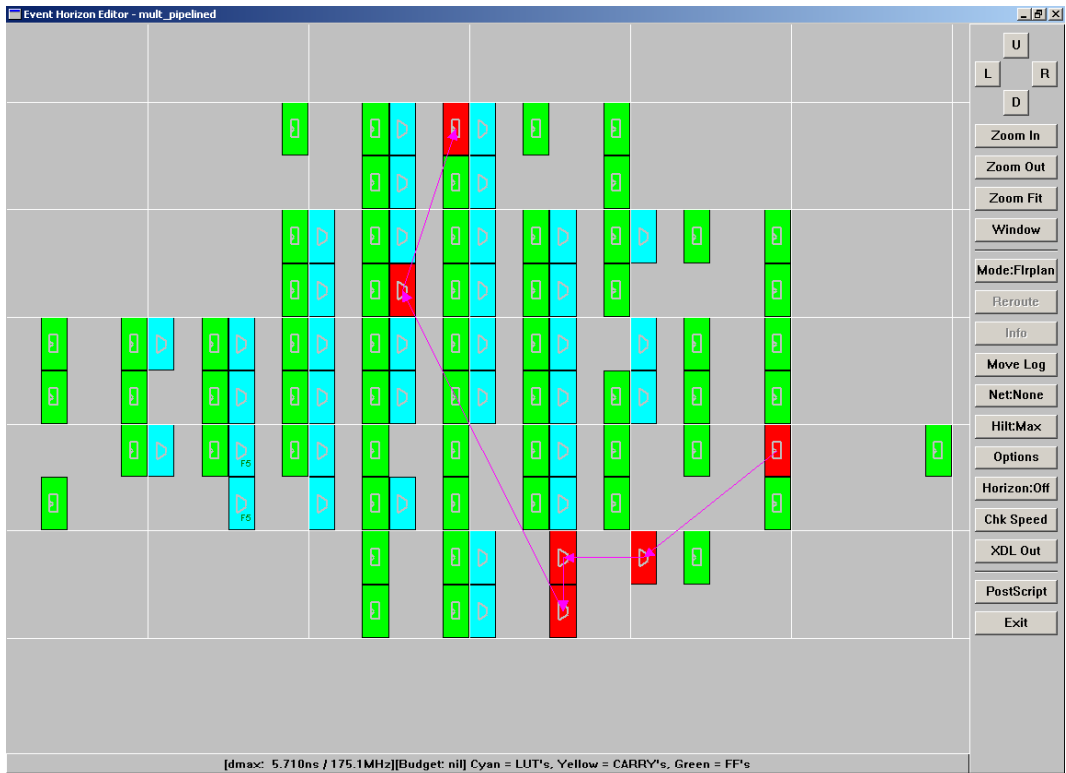


Figure A-20: *Mult* circuit after pipelining