UC Irvine
ICS Technical Reports

Title
A system for microarchitecture and logic optimization

Permalink
https://escholarship.org/uc/item/0qg124vr

Author
Zanden, Nels Vander

Publication Date
1991

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Z
699
C3
no. 91-39

A SYSTEM FOR MICROARCHITECTURE AND LOGIC OPTIMIZATION

by

Nels Vander Zanden

Information and Computer Science Department
University of California, Irvine
Irvine, CA. 92717

Technical Report 91-39

ABSTRACT

This thesis spans two levels of the design process by examining optimization at both the register-transfer level and at the logic level. More specifically, this thesis addresses the following two problems: 1) performing logic synthesis for custom layout rather than the traditional approach that focuses on synthesis for standard cells, and 2) performing optimization for custom layout from register-transfer level netlists. Thus optimization is performed on the microarchitecture design and at a lower level for individual microarchitecture components.

First, techniques are introduced for generating gate-level netlists that take advantage of custom layout capabilities. Such techniques include limiting serial/parallel transistor chains, transistor sizes, and capacitive loads in forming complex gates. These considerations have not been incorporated in previous logic synthesis systems.

Second, techniques are introduced for improving the microarchitecture structure and using estimates from lower-level optimization tools to guide microarchitecture design optimizations that attempt to meet user specified area and time constraints. These techniques include the capability for mixing layout styles such as custom layout for random-logic components and bit-slicing for regularly structured components. In this manner the entire design, control logic and datapath, can be optimized at the same time. Further, this paper presents a new methodology for microarchitecture-level optimization that greatly reduces the amount of technology-specific knowledge necessary to perform the optimizations.

A SYSTEM FOR MICROARCHITECTURE AND LOGIC OPTIMIZATION

Nels B. Vander Zanden, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1991
Daniel D. Gajski, Advisor

In recent years the drive to produce more complex integrated circuits while spending less design time has driven the demand for design automation tools. The search for design automation methods has resulted in the design of numerous behavioral synthesis and logic synthesis tools. This thesis spans two levels of the design process by examining optimization at both the register-transfer level and at the logic level. More specifically, this thesis addresses the following two problems: 1) performing logic synthesis for custom layout rather than the traditional approach that focuses on synthesis for standard cells, and 2) performing optimization for custom layout from register-transfer level netlists. Thus optimization is performed on the microarchitecture design and at a lower level for individual microarchitecture components.

First, techniques are introduced for generating gate-level netlists that take advantage of custom layout capabilities. Such techniques include limiting serial/parallel transistor chains, transistor sizes, and capacitive loads in forming complex gates. These considerations have not been incorporated in previous logic synthesis systems.

Second, techniques are introduced for improving the microarchitecture structure and using estimates from lower-level optimization tools to guide microarchitecture design optimizations that attempt to meet user specified area and time constraints. These techniques include the capability for mixing layout styles such as custom layout for random-logic components and bit-slicing for regularly structured components. In this manner the entire design, control logic and datapath, can be optimized at the same time. Further, this paper presents a new methodology for microarchitecture-level optimization that greatly reduces the amount of technology-specific knowledge necessary to perform the optimizations.

A SYSTEM FOR MICROARCHITECTURE AND LOGIC OPTIMIZATION

BY

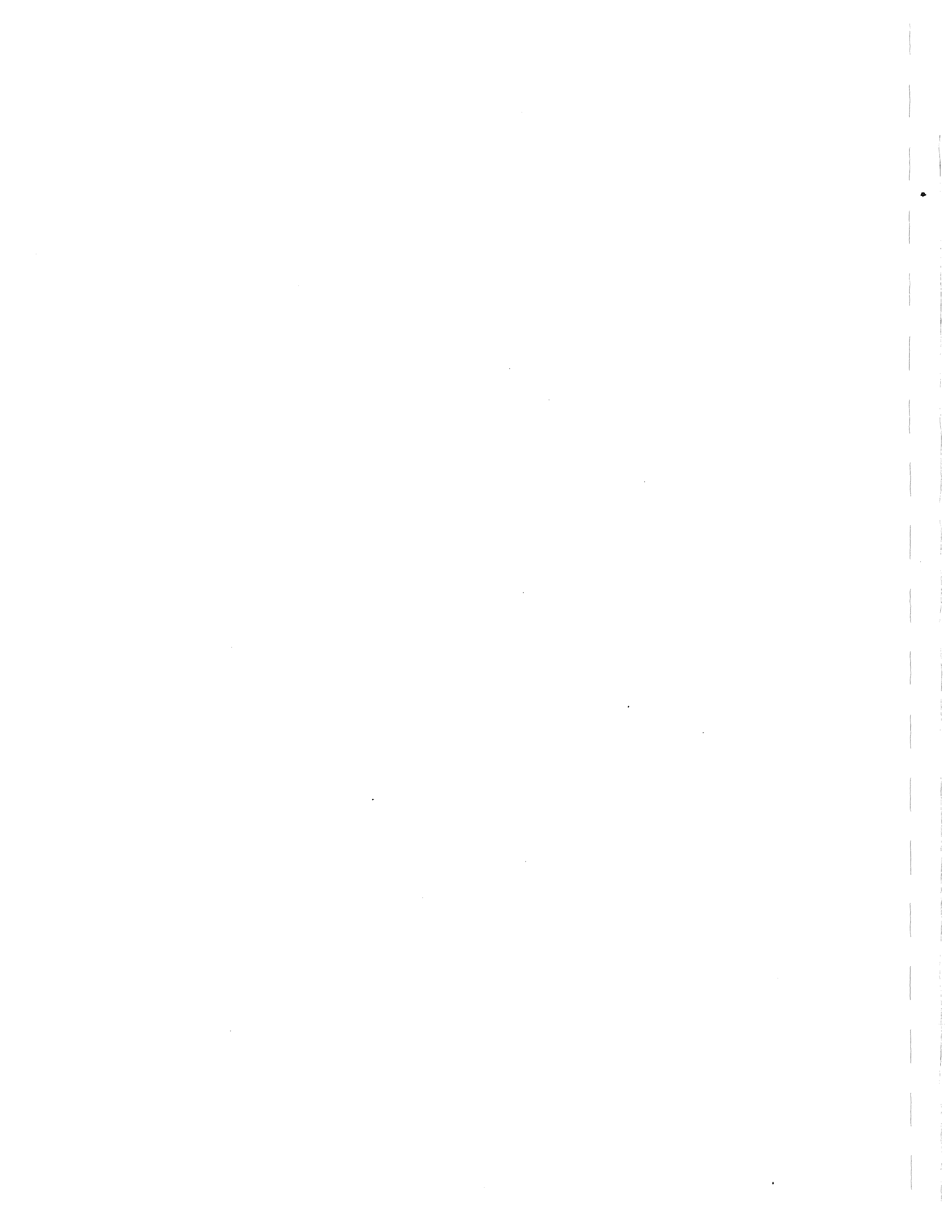
NELS BLAKE VANDER-ZANDEN

B.S.C.I.S., Ohio State University, 1984
M.S., University of Illinois, 1986

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991

Urbana, Illinois



©Copyright by

Nels B. Vander Zanden

1991

To my father,

James Vander Zanden

ACKNOWLEDGEMENTS

I had the good fortune of working with a top-notch advisor, Dr. Daniel Gajski, while pursuing my Ph.D. He contributed to this work in many ways not only in terms of technical discussions, ideas, and advice, but also in terms of a great deal of encouragement and inspiration. I have learned much from him and am greatly honored to have worked with him.

My father also deserves credit for ensuring that I always had the best when it came to education. His support and confidence in me and my abilities have always given me great strength and reassurance. Together with my brother Brad, I have received much love and friendship and known that they are always there in times of need.

I'm also extremely thankful for the exceptional work environment. My colleagues in CADLAB were more than just officemates, they were a great set of friends. They include Dr. Nikil Dutt, Joe Lis, Tedd Hadley, Frank Vahid (and his wife Amy, an honorary member of CADLAB), Allen Wu, Jim Fradkin, Sanjiv Narayan, Viraphol Chaiyakul, Young Kim, Jim Kipps, Gwo-Dong Chen, Elke Rundensteiner, Rajesh Gupta, Loganath Ramachandran, Salman Iqbal, Adil Haque, and Kenichi Kanehara. They have all been of assistance in some manner on this dissertation and hence they deserve much credit as well. In addition I shared many extracurricular activities with them including games of tennis and volleyball, movies,

beach outings, and dinners. They also helped me get through numerous late night hours at the lab with entertaining discussions and the occasional midnight snack excursion. I greatly value each friendship and to hope to have the opportunity to work with them on future projects.

I would also like to thank my good friends Tim and Mary Kraus for the many fun weekends I shared with them in San Diego. In particular, Tim and I have engaged in many activities, discussions, and challenging tennis matches since my early days at the University of Illinois. With another good friend, Gary Bolotin, I explored some of the natural wonders of Southern California with many hikes and outdoor outings. Such weekends and outings helped me to relax and reapproach my work with a fresh perspective.

Bob Larsen, a Fellow at Rockwell International, also deserves credit. He serves as an important link to the industrial world for academic researchers and gave me many excellent pointers and suggestions that enhanced the quality of my work. In addition he supplied a number of the benchmark examples presented in this dissertation.

Barb Cicone was of great assistance to me in the administrative arena during my affiliation with the University of Illinois. She is one of the friendliest and helpful administrators I have known. I took great comfort in her ability to cut through red tape and her constant willingness to lend a helping hand.

Finally I would like to thank my committee members -- Professors Michael Faiman, Larry Jones, Kent Fuchs, and Alex Veidenbaum. In particular I appreciate the help and friendship provided by Dr. Faiman.

TABLE OF CONTENTS

CHAPTER

1. INTRODUCTION 1
   1.1. Design Synthesis Overview 1
   1.2. Contributions 5
   1.3. Thesis Overview 7

2. DESIGN PROCESS 8
   2.1. Introduction 8
   2.2. Design Synthesis Process 9

3. PREVIOUS WORK IN LOGIC SYNTHESIS 14
   3.1. Introduction 14
   3.2. Refinement Techniques 15
   3.3. Optimization Techniques 23
   3.4. Optimization Strategies 38

4. THE MILO SYSTEM 49
   4.1. Introduction 49
   4.2. Previous Work 49
   4.3. System Architecture 52

5. LOGIC SYNTHESIS FOR CUSTOM LAYOUT 61
   5.1. Introduction 61
   5.2. Parameters for Custom Layout 63
   5.3. Strategies for Layout Driven Synthesis 71
   5.4. Algorithm for Layout Driven Synthesis 72
   5.5. Results 86

6. MICROARCHITECTURE OPTIMIZATION 92
   6.1. Introduction 92
   6.2. Types of Microarchitecture Optimization 92
   6.3. Strategies for Microarchitecture Optimization 115

6.4. Experimental Results 126

7. CONCLUSION 153
7.1. Summary 153
7.2. Future Research 154

APPENDIX A. LOPT Program Description 157
A.1. Tutorial 157
A.2. Input Files 160
A.3. LOPT Parameter File Format 162
A.4. LOPT Report File Output Format 163
A.5. LOPT Usage 163

APPENDIX B. IIF 165
B.1. Introduction 165
B.2. IIF declarations 166
B.3. IIF expression 166
B.4. Examples of IIF Usage 168
B.5. Operator Precedence 172

APPENDIX C. MILO Program Description 173
C.1. Introduction 173
C.2. Milo Usage 173
C.3. Milo Parameter File Format 175
C.4. MILO Report File Output Format 176
BIBLIOGRAPHY 177
VITA 181

LIST OF FIGURES

Figure 1. Synthesis Overview 4
Figure 2. SOCRATES System Architecture 17
Figure 3. Logic Consultant System Architecture 18
Figure 4. Factorization for Timing Improvements 20
Figure 5. LSS System Architecture 22
Figure 6. Expert Systems 25
Figure 7. LSS Level 1 Optimizations 33
Figure 8. Techniques for Reducing Delay Along Critical Paths 42
Figure 9. Use of a Hash Table to Reduce the Number of Rules 47
Figure 10. MILO Environment 54
Figure 11. Tool Interface with the Component Database 59
Figure 12. Standard Cell Layout vs. Custom Layout 62
Figure 13. Custom Layout Results Using Synthesis for Standard Cells 65
Figure 14. Complex Gate Formation 67
Figure 15. Effect of Transistor Sizing on Path Delay 67
Figure 16. Effect of Large Transistor Sizes in Complex Gates 70
Figure 17. Algorithm for Layout Driven Synthesis 73
Figure 18. Critical Path Reduction Operations 76
Figure 19. Example of Shortening Critical Path 78
Figure 20. Area/Delay Considerations for 2-Level Complex Gates 82
Figure 21. Area/Delay Considerations for 3-Level Complex Gates 83
Figure 22. Composite Graph of Experimental Results 90
Figure 23. Minimization Rules 93
Figure 24. Factorization of Microarchitecture Components 94
Figure 25. Example of Factorization 98
Figure 26. Signal Swapping 99
Figure 27. Merge Similar Units 102
Figure 28. Three Possible Merging Cases 104
Figure 29. Results of Merging 106
Figure 30. Rule for Merging 106
Figure 31. Merging Unsimilar Units 106
Figure 32. Component Duplication 109

Figure 33. Common Subexpression Extraction 110
Figure 34. Common Subexpression Elimination 113
Figure 35. Common Subexpression Elimination Example 114
Figure 36. Overview of Microarchitecture Optimization 116
Figure 37. Random Logic Grouping 120
Figure 38. Option for performing ALU functions 125
Figure 39. Block Diagram of the Rockwell Counter 128
Figure 40. Three Optimization Approaches for the Rockwell Counter 131
Figure 41. Block Diagram of Armstrong Counter 132
Figure 42. Three Optimization Approaches for Armstrong Counter 135
Figure 43. Block Diagram of DRACO 136
Figure 44. Three Optimization Approaches for Draco2 142
Figure 45. Three Optimization Approaches for Draco3 143
Figure 46. Three Optimization Approaches for Draco Schematic 144
Figure 47. Layout of Module Generator Design for Draco2 146
Figure 48. Layout of MISII Design for Draco2 147
Figure 49. Register with Parallel Load 169
Figure 50. Four-bit Adder/Subtracter 170
Figure 51. Tristate Connection 171

LIST OF TABLES

Table 1. Custom Layout results for MISII and the LDS Algorithm 87
Table 2. Custom Layout Results Using MCNC Benchmarks 88
Table 3. Standard Cell Layout Results Using MCNC Benchmarks 89
Table 4. Time Optimization Results for the Rockwell Counter 129
Table 5. Area Optimization Results for the Rockwell Counter 130
Table 6. Time/Area Tradeoffs for Rockwell Counter 130
Table 7. Time Optimization Results for the Armstrong Counter 133
Table 8. Area Optimization Results for the Armstrong Counter 133
Table 9. Area/Time Tradeoffs for the Armstrong Counter 134
Table 10. Time Optimization Results for Draco2 137
Table 11. Area Optimization Results for Draco2 137
Table 12. Time Optimization Results for Draco3 138
Table 13. Area Optimization Results for Draco3 138
Table 14. Time Optimization Results for Draco Schematic 139
Table 15. Area Optimization Results for Draco Schematic 139
Table 16. Time/Area Tradeoffs for Draco2 140
Table 17. Time/Area Tradeoffs for Draco3 140
Table 18. Time/Area Tradeoffs for Draco Schematic 141
Table 19. Comparison of MILO and MISII Timing Optimization 147
Table 20. Comparison of MILO and MISII Area Optimization 148
Table 21. MILO gate vs. MILO module generator designs (Time) 149
Table 22. MILO gate vs. MILO module generator designs (Area) 150
Table 23. MISII vs. MILO module generator designs (Time) 151
Table 24. MISII vs. MILO module generator designs (Area) 152

CHAPTER 1.

INTRODUCTION

Over the past decade tremendous advances have been made in VLSI design. Greatly increased circuit complexity, however, continues to outpace engineers' abilities to develop chips with up to a million transistors. Conventional design methods have entailed teams of highly skilled and experienced engineers providing many months of effort. More recently, companies have been turning to logic synthesis tools to help alleviate market pressures that demand lower-cost implementations with shorter development times. These tools allow less experienced designers to develop application-specific ICs (ASICs), typically gate arrays or standard cells. Further, the automated systems free a designer from the exploding number of details and provide greater time to examine high-level issues and experiment with various architectures. Thus such tools increase productivity and provide for better complexity management.

1.1. Design Synthesis Overview

Synthesis tools employ two basic techniques in producing a final, workable design. These are refinement and optimization. Refinement is the process of

transforming a behavioral description into an initial design. Usually this design consists of components such as multiplexors, decoders, and gates such as AND and OR. Optimization is the process of transforming the initial design into one that meets some set of objectives relating to time, area, etc. Often a design proceeds through a number of levels of abstractions. At each level, these two stages are performed. For example, typically refinement and optimization can be used on a generic representation and later on a technology-specific one.

The synthesis process is shown in Figure 1. Refinement begins with the user's behavioral description. The input may take a textual or graphical form. Textual representations would be a hardware description language, such as VHDL, a set of boolean equations, or a table, such as PLA format. Graphical representations include a generic or technology-specific schematic, or a menu-driven interactive process that extracts a set of parameters from the user. Behavioral descriptions in language form require a compiler to generate an initial netlist consisting of high-level components such as decoders and registers. Diagrams entered through schematic capture may also consist of high-level components. Boolean equations, PLA format, or low-level schematics are representations for the logic level.

The behavioral description represents a black box whose inputs, outputs, and functionality are described. The representation generally does not convey any information as to the technology type or necessarily any style of implementation -- be it parallel, pipelined, etc. Since many systems accept different behavioral

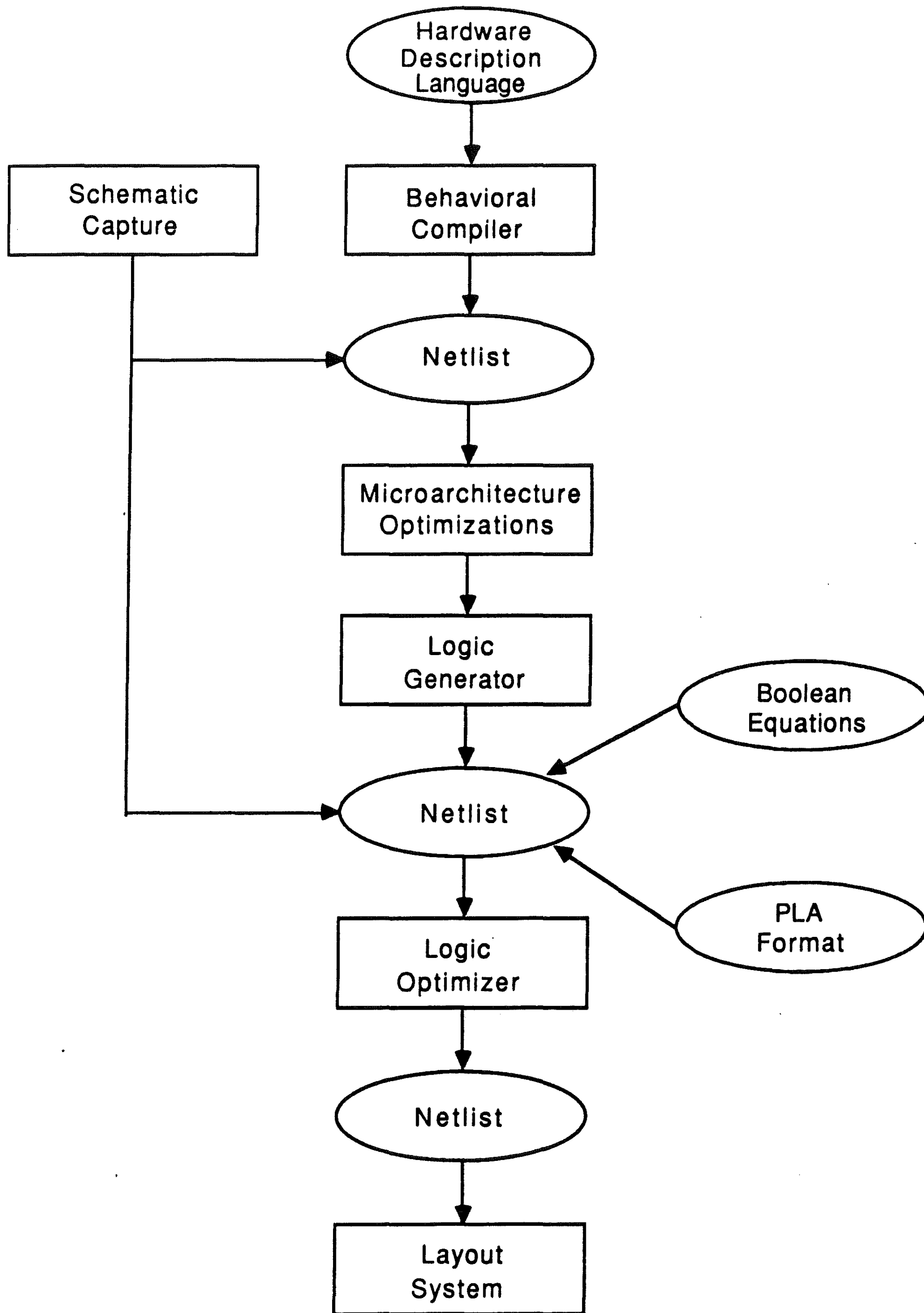


Figure 1. Synthesis Overview

specifications, they must first convert them all to one central format, such as a generic schematic or set of boolean equations. This central format is then used to create a detailed representation of the design.

The design created in the refinement stage is usually by no means optimal and thus the optimization phase is called upon to improve performance. Figure 1 identifies two levels at which optimization can take place -- at the microarchitecture level and the logic level. One could add other intermediate levels. A logic generator is required to decompose the design into the lower level.

Optimization can be guided by two different strategies. The first is to optimize everything until no further improvements can be made. When optimizing for multiple constraints there may be a weighting or priority scheme to determine which design improvements should be performed. Essentially, this is a "brute-force" method as optimizations tend to be made in random parts of the circuit in no directed fashion. The second strategy is constraint driven. It relies on a set of user-entered parameters to direct the flow of optimization. For example, optimizations may be applied initially only along a critical path when attempting to meet timing constraints. Once one constraint is met, the optimizer will turn to another and work toward meeting the user's objectives. Once these are satisfied, the optimization stage may end or may apply the first strategy.

1.2. Contributions

This thesis discusses three aspects of the design synthesis process: 1) logic optimization, 2) microarchitecture optimization, and 3) the integration of logic and microarchitecture optimization tools into a larger behavior to layout synthesis tool. In particular, it describes **MILO**, a system that performs both microarchitecture and logic optimization.

The following novel features are introduced in this thesis:

1. **Control algorithms and rules for optimization of the microarchitecture.** In previous systems, the microarchitecture design is not fully optimized before decomposing the design into logic gates. Techniques are described in this thesis for optimizing components at a microarchitecture level before they are decomposed into gates. Such optimizations are nearly impossible to perform after decomposition.

2. **Development of algorithms and a tool that performs logic synthesis for custom layout.** Traditional logic synthesis tools optimize for standard cell and gate array implementations. Additional parameters need to be considered when synthesizing for custom layouts.

3. **Use of evaluation in the design process.** Feedback to the microarchitecture optimizer from lower level optimization tools permits reexamination of the high-level design. By using logic optimization and technology mapping tools on

each microarchitecture component, the microarchitecture optimizer can obtain accurate estimates of each component's delay and area. Modifications can then be made to bring the microarchitecture design closer to the user-specified constraints.

4. Incorporation of layout styles into microarchitecture optimization. Microarchitecture components can be laid out in various styles including standard cell, bit-sliced, and custom layout. The output of the microarchitecture optimizer is passed to a placement tool that forms a bit-sliced datapath section and then places random logic around the datapath. Giving the microarchitecture optimizer the capability to select different layout styles increases its ability to meet the user constraints for time and area.

5. New methodology for tool integration. In traditional optimization systems, a logic synthesis tool is tied in with higher level synthesis tools. Using the new methodology, logic and layout optimization tools are hidden from the microarchitecture optimizer by a component database. The microarchitecture optimizer accesses the database to perform low-level optimizations. In this manner, logic optimization tools can be added or removed independently of the microarchitecture optimizer.

1.3. Thesis Overview

The thesis is organized as follows. Chapter two outlines the necessary parts of a behavior to layout synthesis system. It discusses the responsibilities of each part and shows how logic and microarchitecture optimization can be applied. Chapter three surveys previous work in the logic synthesis area. Chapter four provides an overview of the microarchitecture optimizer and shows how it incorporates the logic optimizer to get low-level design statistics. The chapter also shows how the microarchitecture optimizer fits into a larger system that produces layout from a behavioral description. Further, it illustrates the types of commands used to retrieve information from the database and how the database is used to hide logic-level details from the microarchitecture optimizer. Chapter five describes a new approach to logic synthesis for custom layout. Chapter six discusses issues in microarchitecture optimization and presents an algorithm for optimization of microarchitectural designs. Finally, chapter seven summarizes the contributions made by this work and examines some future work that can be performed.

CHAPTER 2.

DESIGN PROCESS

2.1. Introduction

Generation of digital hardware generally passes through four stages of development: behavior, microarchitecture, logic, and layout. Behavior describes the functionality of the hardware and has often been written using VHDL, Pascal-like languages or C. Behavioral synthesis tools convert these descriptions into a microarchitecture structure called a Register-Transfer-Level design. This structure consists of components such as ALUs, registers, counters, and multiplexors. Each of these components can in turn be expanded into a logic-level design consisting of gates and flip-flops. Finally a layout can be generated from transistors that compose each gate.

Today's designers are increasingly able to enter their designs at higher levels of abstraction. Recently a number of tools that can translate a behavioral description to structure have been developed. Some of these tools are tuned to a particular style of architecture and hence little further optimization is required on the microarchitecture level. Other tools produce varying styles of architecture usually involving

control and datapath sections. As these architectures are more general, they tend to be less polished and more optimization of their microarchitecture structure is required.

2.2. Design Synthesis Process

Transforming a behavioral description into layout requires the work of a number of stages: behavioral synthesis, microarchitecture optimization, logic optimization, floorplanning, and layout. This section describes the goals and interactions of these tools.

2.2.1. Behavioral Synthesis

Behavioral synthesis tools convert a behavioral description into a dataflow graph with each node representing a functional operator (such as add or compare) [CaRo85] [OrGa86] [McPa88]. These operations must be assigned to a control step, through the process of scheduling [PaGa87] [PaKn87], that chooses a point in time at which the operation will be performed. In addition, the operator is assigned to a particular hardware module, through the process of binding [TsSi86] [PaPM86]. During this process, the synthesis tool explores multiple designs and attempts to determine which designs appear most likely to meet the set of user constraints. Estimators are employed to guide the synthesis tool towards one or more such designs.

Estimates for behavioral synthesis tools are usually obtained in one of two fashions. The first technique uses a set of formulas that when given a component type (ALU, Register, etc.) and its set of parameters (eg., number of inputs, architecture style, technology-type) produces a rough estimate of the time and area. Such estimates are not finely tuned but help to weed out unacceptable designs. A second technique is to expand the design into a lower level design consisting of gates, possibly even mapping the gates into a technology-specific library. Alternatively, a high-level floorplan of the microarchitecture components can be generated to obtain a feel for design characteristics. These methods require more time to produce the estimates and will usually be reserved for use when the number of possible designs has been greatly narrowed.

The use of estimators allows the behavioral synthesis tool to select an overall architecture by making decisions on the number of busses, use of pipelining, etc. In addition, the synthesis tool attempts to minimize the number of connections between modules and reduce the total number of modules. An appropriate architectural style must also be chosen for each microarchitecture component, such as ripple-carry or carry lookahead when using an adder. Because the estimates are only a rough predictor of the final design after layout, more rigorous analysis and optimization is required of the microarchitecture design. Thus there is a need for a microarchitecture optimization tool.

2.2.2. Microarchitecture Optimization

The major goals of microarchitecture optimization are to: (a) remove inefficient constructs, (b) select a style of architecture for each component that suits the area/time requirements, (c) insert buffers on outputs that have a high fanout, (d) select which microarchitecture components to combine and perform logic optimization on as a single unit, and (e) select a layout style for each microarchitecture component such as PLA, random logic, bit-sliced, etc. Once the initial microarchitecture structure has been cleaned up, the optimizer has two options in producing the final design: (a) completely expand and optimize or (b) only partially expand and optimize.

The first approach is to combine all components into a single combinational block and optimize. Logic optimization tools have been shown to be very effective for reducing the area of a design or restructuring logic to meet timing constraints. This approach may not be the best, however. First, logic optimization of large designs may require large amounts of CPU time and memory. The same will be true in the layout phase when floorplanning is performed. Second, some optimizations can be made at the microarchitecture level that cannot be made at the logic level.

The second approach involves only a partial expansion of the design. Various groups of the components can be combined into a single component and optimized. For example, random logic gates can be grouped together and passed to a logic

optimization tool while more regularly structured components such as ALUs are optimized separately and not combined with the surrounding logic. Sometimes logic optimization can be avoided if the random logic is implemented as a PLA. Higher level components such as ALUs and registers can be implemented from libraries of prefabricated integrated circuits. For example, components from the library of the TTL 7400 series. Another alternative is the use of layout module generators.

Module generators can be used for components with a regular style of architecture such as ALUs and registers. For these components a one-bit layout slice is generated and then replicated based on the component's bit width. The bit-sliced layout will typically be more compact than what could be generated using standard cell or custom layout generators to layout the same logic from a random logic description. Thus using module generators for components with regular structures will usually result in denser layouts. In some circumstances, however, a component such as an ALU can be combined with surrounding logic to reduce the number of gates. This saving of gates may produce a smaller layout than if module generators has been used. Thus the microarchitecture optimizer must be able to discover such conditions.

2.2.3. Floorplanning/Layout

As mentioned earlier, different layout styles can be used for each component. The floorplanner employs a set of layout tools to produce the layout for each microarchitecture component. Then it needs to determine how to arrange these

different layout styles on the same integrated circuit. For example, bit-sliced components are grouped into one or more datapaths, each datapath containing components of approximately the same bit-width. Components in each datapath are arranged in a vertical fashion, forming a bit-sliced stack. Random logic components can then be placed around the border of the bit-sliced stack and into unfilled spaces that result from bit-slice mismatches in the stack.

The floorplanning tool must also take into account routing concerns. For example, components along the critical path should be placed close together to reduce the amount of delay caused by routing. Timing information can be supplied by the microarchitecture optimizer for this purpose.

CHAPTER 3.

PREVIOUS WORK IN LOGIC SYNTHESIS

3.1. Introduction

A number of automated logic synthesis systems have been previously reported: LSS [JoTr86], SOCRATES [GrBa86], MIS [BrRu87], BOLD [BoHa87], DAGON [Ke87]. Commercially available tools include the Logic Consultant [Kim87], the Synopsys Design Optimizer, and Silc Technologies Silcsyn. These systems have used algebraic, language compiler, and expert system techniques to reduce circuit delay and gate count in combinational logic. In general, such systems are best suited for optimizing random logic that can easily be represented by boolean equations or PLA format. An exception is LSS which accepts a register-transfer language as its input, allowing entry of high-level operators. However, only limited types of optimization are performed on the high level operators before they are decomposed into gates. Once a design is at gate level it is impossible to recover high-level information that may be necessary for restructuring the design in order to meet timing or area constraints.

3.2. Refinement Techniques

3.2.1. SOCRATES

Figure 2 shows the SOCRATES system. SOCRATES accepts boolean equations, PLA format, or a netlist as a behavioral description. Two-level boolean equations are used as the central format. Multi-level boolean equations can be extracted from the netlist, then run through an expander to generate the two-level format. Likewise, an extractor can be used to generate two-level boolean equations from PLA format. An algebraic minimizer, ESPRESSO IIC minimizes the two-level equations, taking advantage of don't care conditions. The next step uses weak-division to find common subterms in the design and form multi-level equations, thereby reducing the amount of logic required to implement the function. This minimized design is written back into netlist form from which the logic optimizer will operate. The optimizer is discussed in a later section.

3.2.2. Logic Consultant

The modules comprising the Logic Consultant system are displayed in Figure 3. Like SOCRATES, the Logic Consultant accepts boolean equations, PLA format, and netlists generated from schematics as input to its system. Schematics can be created on a Mentor workstation using components from Mentor's generic library (GENLIB) or components from a technology-specific library. A decomposition module converts the input description into components from the Trimeter Generic

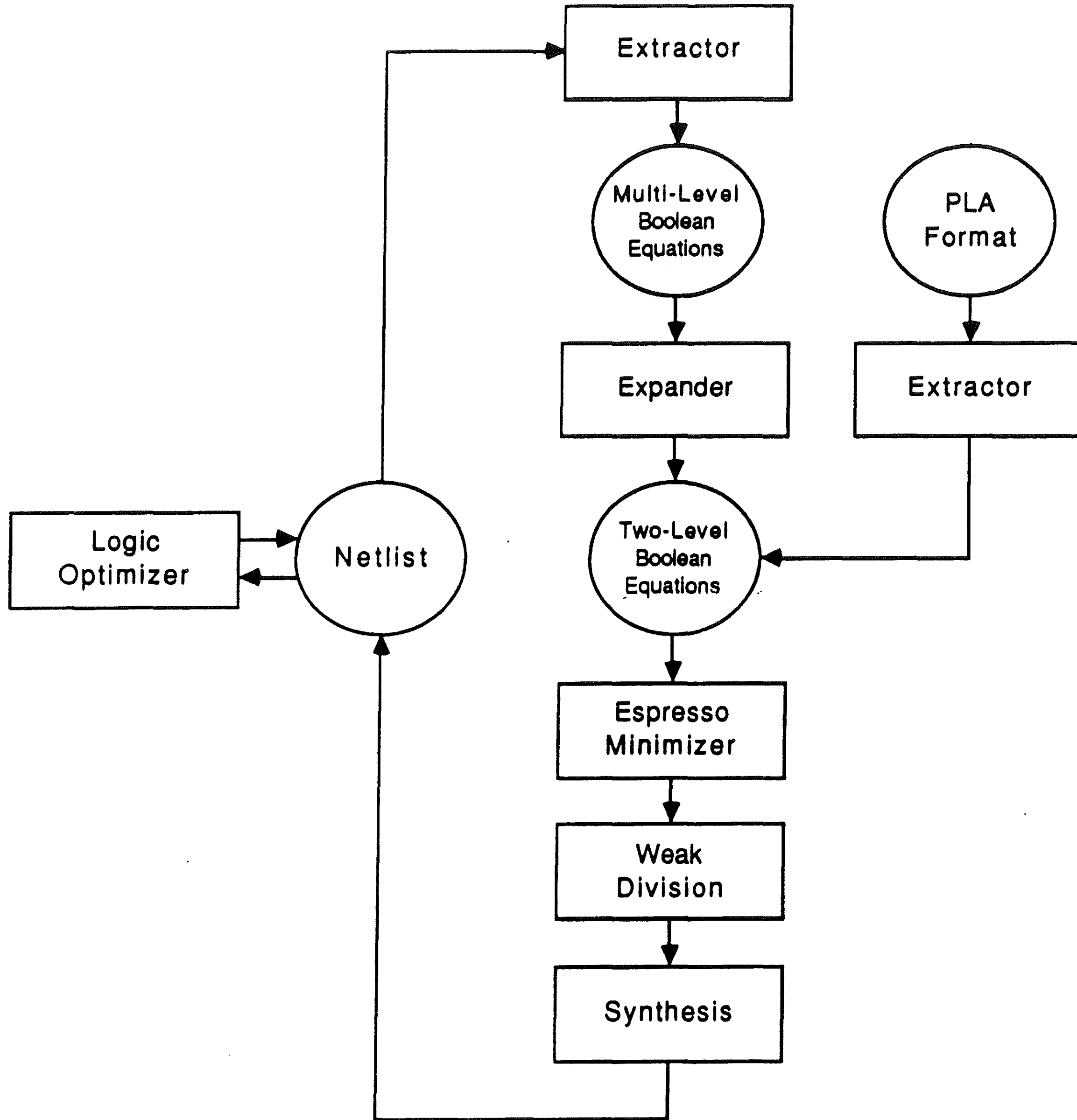


Figure 2. SOCRATES System Architecture

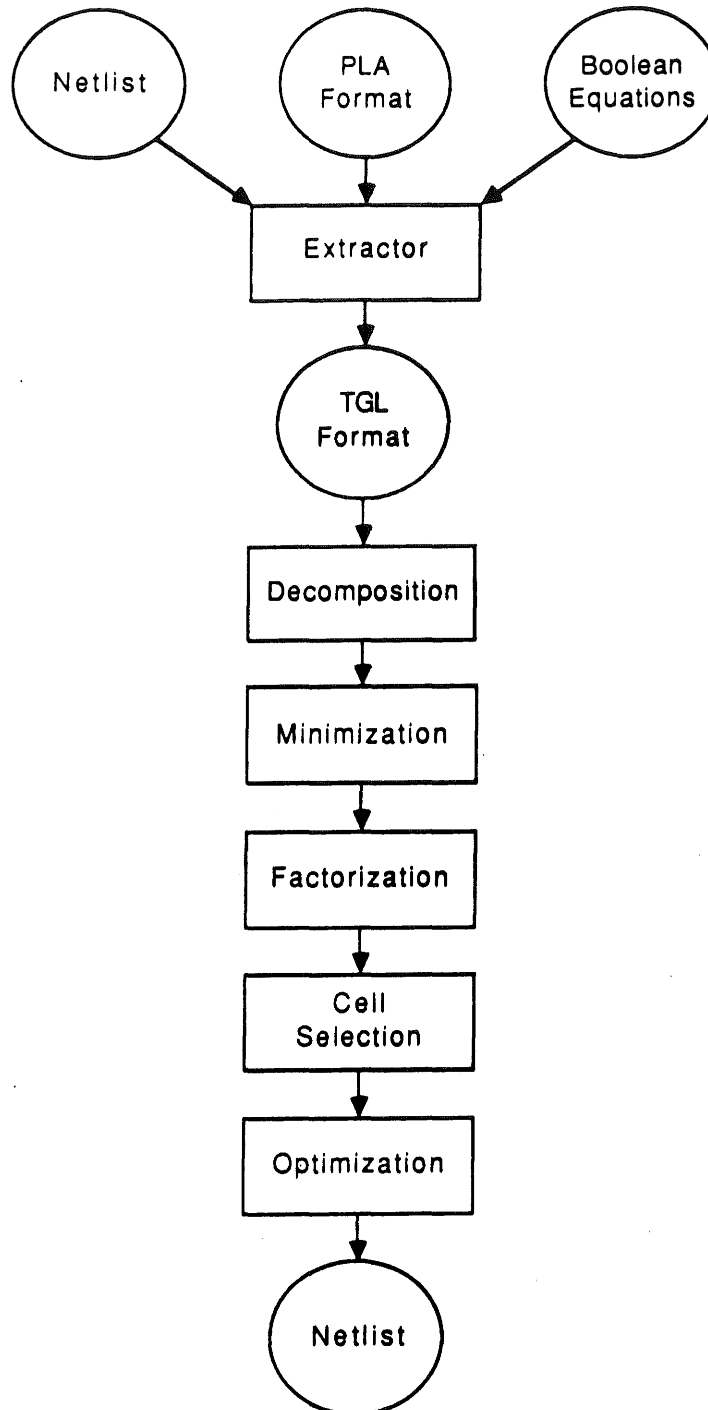


Figure 3. Logic Consultant System Architecture

Library (TGL) and isolates the combinational logic for processing by the minimization module. Technology-specific components are replaced with TGL components through user-entered rules in the knowledge base. For MSI components (ie., multiplexors, decoders, ALUs, etc) this replacement is optional. The user specifies whether the MSI elements should be decomposed into TGL gates (via the rule base) or should be left "as is". Those components that are not decomposed are treated like sequential logic elements and removed from the design that is passed to the minimizer. Thus if an MSI component is not decomposed, it will not be optimized with the surrounding logic. This strategy poses a problem since in decomposing the MSI components, one loses the advantage of using them. Typically high-level components are added to a technology library because the designer of that component constructed it in such a way that it takes up less area and has greater speed than a corresponding gate implementation. But once a component is decomposed for optimization, it may be difficult or impossible to find after flattening and refactoring the circuit. However, if the components are not decomposed, no optimization is performed. As will be seen later in this thesis, a better solution is to perform some optimizations at the microarchitectural level.

The minimizer module accepts combinational logic consisting entirely of TGL components from the decomposition module. The minimizer then develops a two-level SOP design and removes redundant terms. This new design will be passed to the factorization module. For some circuits it may be desirable to bypass the minimization module and go directly to factorization. This is the case when the

generated two-level SOP form contains an explosive number of terms. These circuits require extensive CPU time and after factorization typically contain more components than the original design. For example, certain multi-level circuits may require many times their number of circuit elements to be represented in a two-level format. As factorization is performed on a local basis (and not globally), one cannot guarantee that the factorization will be optimal. Hence factorization may be unable to reduce the gate count to the prior number of elements. Designs with a large number of XOR and XNOR gates are one example of this type of circuit. Thus the Logic Consultant's minimizer does not produce a better design in all cases.

The factorization module attempts to factor out common terms to produce a multi-level design. It also factors in such a way to reduce the delay along the longest path. For example, consider Figure 4. The bottom input of the AND gate is part of the longest path. This path can be shortened by factoring the 3-input AND into two 2-input AND gates. Thus some timing considerations are taken into account. Note however that the longest path may not always by a critical path. When this is the case, the assumption that it is a critical path will prevent optimization for area along that path. Hence, such a factorization strategy is not truly constraint driven.

Factorization continues until no further common terms or timing improvements can be found. The Logic Consultant factors all paths as completely as possible. Since this includes critical paths, certain factorizations must be undone at a later

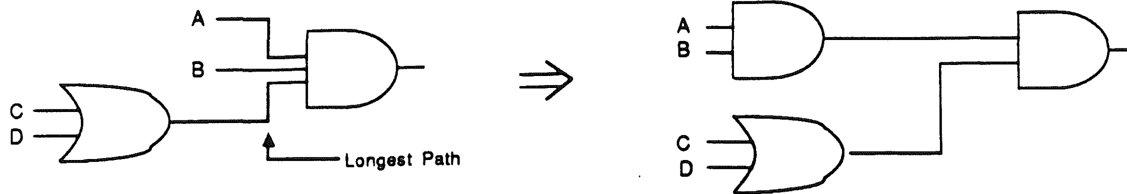


Figure 4. Factorization for Timing Improvements

time.

3.2.3. LSS

The LSS system architecture is shown in Figure 5. LSS proceeds through four different description levels to produce an optimized technology-specific circuit: high-level, AND/OR, NAND/NOR, and technology specific. Each level requires a translator to produce its description format from a higher-level description. Likewise, each level has an optimizer to apply simplifying transformations. By introducing multiple levels, the designers of LSS followed the example of programming language optimizing compilers that produce several intermediate descriptions before generating assembly-language code [DaJo80]. Changes can be made through a number of levels, thereby simplifying analysis and optimization at each level. For

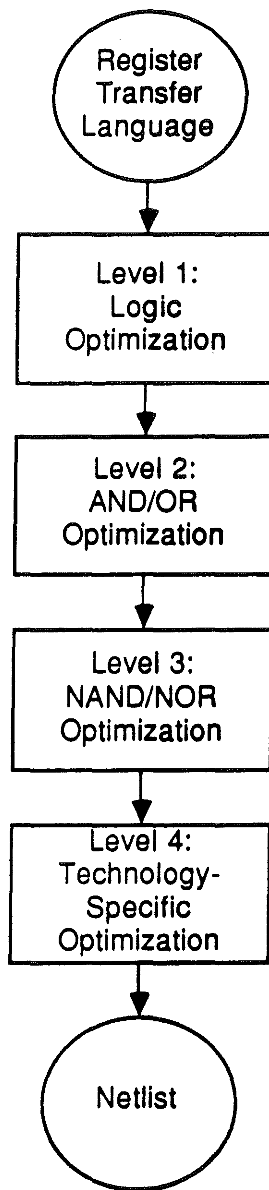


Figure 5. LSS System Architecture

example, simplification can be made in terms of generic components, then converted to technology-specific components for further optimization. Attempting to improve a high-level description directly into a low-level one is a much tougher task. Also by using generic levels, only the technology-specific optimizations need be rewritten when the technology changes.

LSS begins with an algorithmic representation using a register-transfer language. Using a simple translator, LSS produces a logic graph representing the design. Translation is straightforward as operators in the behavioral description are assigned to nodes in the graph. Each node is a generic component (such as an AND gate or decoder) and is connected via the graph's edges to other components. Optimization at this level is discussed in a later section.

Another translator is called to produce the second level AND/OR description. It decomposes high-level components, such as decoders, into AND/OR/NOT gates. Optimizations are then applied at this level. The third description level, NAND/NOR consists of only NAND and NOR gates. Once again, the translator that produces this description is achieved through naive transformations that may produce unnecessary NANDs and NORs. These "extra" gates are removed by the optimizer at this level. Finally, LSS provides a translator to produce a technology-specific design using components from a technology library that consists of gate-array macros (such as NAND/NOR gates, complex AND/OR gates, etc). This technology description may then be optimized.

3.3. Optimization Techniques

After refinement, synthesis tools call upon optimizers to improve the design. Their optimization techniques fall into one of the three expert system types shown in Figure 6. Each consists of some type of blackboard and knowledge base. The blackboard is where the design resides and can be examined by the knowledge base. Included in the blackboard is usually a netlist describing the design, statistics on area and path delays, a set of user constraints, and work space for the system to evaluate various attempts at refinement or optimization. The knowledge base consists of a set of rules and algorithmic techniques that utilize the blackboard data structures and make modifications to them. Generally algorithms are assigned structured and well-defined tasks while rules handle unusual or loosely defined problems.

3.3.1. Rules Only Strategy

The first type of expert system is entirely rule based. This approach generally lacks structure as rules are entered essentially independently of one another. The control unit selecting a rule to fire must examine all rules to determine which rules are applicable. Any of the rules whose conditions are satisfied can be fired. It is up to some rule selection mechanism to choose one rule from that set. An early implementation of this type of system was R1 [McDe82], a rule-based system for configuring Vax-11/780 computers. A more recent version of a strictly rule-based system is the Logic Consultant. Both of these systems use the OPS production

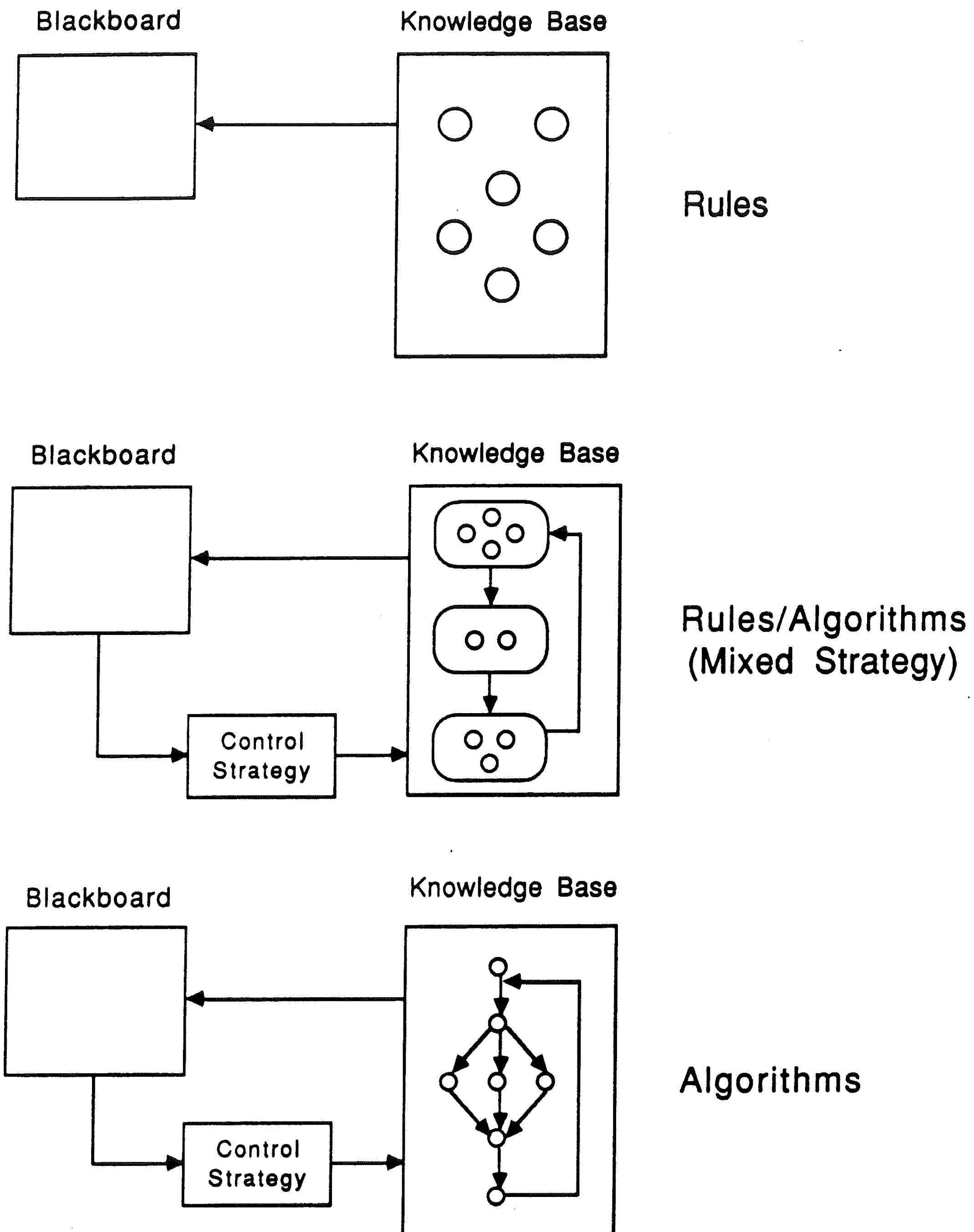


Figure 6. Expert Systems

system language to write rules for their knowledge base. R1 was written in OPS4, the Logic Consultant uses OPS83 -- the lastest OPS version. Each rule consists of a set of conditions and a set of actions to be performed when all of the conditions are met. Control in OPS is exercised through a recognize-act cycle. In this cycle all rules whose If-conditions are satisfied are found using the Rete match algorithm [BrFa85], then one of these rules is selected to be applied. Briefly, the Rete algorithm compiles all of the rule conditions in an OPS program into a set of attributes whose values can be tested. These attributes are placed in a tree-structured sorting network. In doing so, it allows similar tests on attribute values to be shared among different rules. Further, once a test has been performed on a tree node, it is not redone until a change in data occurs upon which the attribute is dependent. These features make the Rete algorithm much more efficient than a simple pattern matcher that examines each rule's conditions on every recognize-act cycle to determine which rules apply.

From this set of applicable rules, called the conflict set, a single rule must be selected. OPS has a conflict resolution scheme to choose this rule based on a number of tests through which rules are eliminated from the conflict set. Priority is given to rules in the following order [Fo85]: rules that have not been executed, rules whose first attribute (value of the first condition) has been most recently altered, rules with the most conditions, rules that entered the conflict set most recently.

In a rule-based system incorporating such schemes, control over which rules are applied can only be achieved by adding more conditions to each rule. In R1, an additional condition was added to each rule corresponding to the stage of the design task. The Logic Consultant system builds in limited structure by examining its set of applicable rules and making an evaluation of the gain produced by each rule. The rule with the largest gain can then be applied on the circuit. Another characteristic of strictly rule-based systems is the lack of backtracking. Neither R1 nor the Logic Consultant permit backtracking -- once a rule is applied, it will not be undone.

The Logic Consultant's first optimizer module is the cell selection module. It converts the design consisting of TGL components to technology-specific components. This module uses rules from the knowledge base that specify how one or more TGL components map into a technology-specific component.

Once a technology-specific design is derived, the design is further optimized by a technology-specific optimization module. This module utilizes rules that make equivalent circuit transformations to improve area or time. The optimizer chooses which rules to apply based upon some formula that examines time/area tradeoffs. For critical paths and the longest path, rules are selected that decrease delay but that may increase area. Along non-critical and short paths, the Logic Consultant applies rules that decrease area at the expense of time.

Certain rules may appear to increase area or time but can actually result in a decrease by applying "clean up" rules. The Logic Consultant has a classification of rules to do this. For example, if a rule adds inverters to the circuit, the optimization module will examine a set of high-priority or "clean up" rules to determine if any of the high-priority rules can eliminate the inverters from the design. Before implementing any rule, the optimizer calculates the result of the rule applied with any high-priority rules to determine how the rule will affect area and time.

When entering rules into the knowledge base, the user indicates whether a rule is high-priority. The rule classification indicates only that the rule should be examined to determine whether it can "clean up" after a regular rule application. It does not give the high-priority rule preference over a "regular" rule. This high-priority rule classification provides a limited lookahead feature of one rule.

3.3.2. Rules/Algorithms (Mixed Strategy)

The second expert system type makes use of a rule base but employs structuring through algorithmic control. The algorithm establishes a hierarchy that determines which set of rules should be examined at any point in time. This added structuring allows greater control over the type of rules that are eligible to fire. The top level of the hierarchy examines the blackboard to determine the current stage of optimization. Depending on the set of constraints that must be satisfied, various strategies may be selected for further investigation. At this stage, more information is needed to evaluate the remaining strategies. Lower levels in the hierarchy are

called upon to produce it. From this information, the high-level decisions can be reached.

Lower levels in the hierarchy inspect the blackboard in greater detail and can choose a single strategy for optimization. Inside each strategy is further hierarchy. Similar to the strategy selection, one technique in the strategy must be chosen from a number of possibilities. The chosen technique in turn calls upon still lower levels of the hierarchy to carry out a design transformation. These lowest levels include rules for manipulating data structures in the blackboard, keeping data in the blackboard up to date, and removing old or useless information.

SOCRATES is an expert system that uses a mixed strategy for optimization. The SOCRATES optimizer is written in C and runs on the VAX 11/780. Like the Logic Consultant, it consists of rules that make local transformations on the circuit. SOCRATES also has procedures to provide feedback on the time and area savings produced by a rule. Through these measurements, SOCRATES can choose which rule to apply. In addition though, the SOCRATES rule base employs a limited hierarchy to reduce the number of rules that must be considered at any given time. It organizes the knowledge base into a number of classes, such as timing or area optimizing rules. Further, each class of rules is divided into subclasses of related rules. For example, one subclass might replace partially redundant multiplexors with NAND and NOR gates, and another might combine and synthesize those gates [GeCo85]. SOCRATES examines only rules in a particular subclass at any one

time. Each class and subclass of rules is prespecified in a certain order in the knowledge base. SOCRATES then examines each rule class and subclass in this order. Optimization in SOCRATES begins with rules that improve both time and area. Then rules are applied that optimize time, possibly at the expense of area, until all timing constraints are satisfied. Finally, area optimizations are made on noncritical paths, possibly at the expense of time, until no other area improvements can be made.

A rule in SOCRATES consists of a target configuration and a replacement configuration. These configurations are patterns of components, pins, and nets that identify a particular circuit structure. SOCRATES has its own pattern matcher that determines which target configurations are present in a design. Like OPS83, it contains a recognize-act cycle in which possible rule applications are examined, one rule is selected and then applied. Because of the hierarchical rule-base, SOCRATES only needs to consider rules in the currently activated rule subclass. To eliminate rules in the conflict set, the recognize-act cycle may utilize lookahead. Each rule in the conflict set is evaluated by implementing its set of actions, then measuring the resulting effect. The rule's future effect (ie., an examination of the rules that may be applied after the initial rule application) can be observed via a state search. This involves setting up a search tree consisting of future design states. The search tree is a graph with the nodes representing possible circuit implementations and the arcs representing rule applications. The root of the tree is the current circuit implementation. Children of nodes in the graph are ordered by desireability. The

leftmost children being derived from "better" rules [CoBa85]. The path through this state graph producing the lowest cost function is the optimal sequence of rules.

Construction of the graph is performed in a depth-first manner. After each transformation, the results are evaluated by a cost function. If the resulting circuit is acceptable the next set of rule applications will be examined, the "best" rule selected and its effects determined. The process is repeated until some maximum depth is reached in the search tree. If the resulting circuit is deemed "unacceptable", SOCRATES backtracks to the node's father and examines alternative circuit transformations [GaGr84]. In constructing the search tree, SOCRATES keeps a log of changes made to the circuit by each rule application. When backtracking is required, the changes to the circuit can be quickly undone by referring to this log.

Each node of the tree will contain the cost function estimate of the circuit implementation. The lower the cost function, the better the circuit. Once the search tree has been completely built, it can be traversed to determine which sequence of rules is best. Those rules will then be applied.

Since search trees can become massive and require large amounts of CPU time, the designers of SOCRATES introduced metarules that control the size of the search tree. Metarules are rules that contain control knowledge, as opposed to design knowledge that suggests circuit alterations. The SOCRATES metarules are based on the parameters presented in [CoBa85]. The first parameter, **B**, restricts the number of sons any node may have. Thus it limits the breadth of the search

tree. The parameter **D** limits the depth of the tree by restricting the number of consecutive rule applications. Using the neighborhood control parameter, N, restricts rule applications to gates of path distance N from some center gate. This prevents rules that apply to different circuit regions from being considered. The parameter $\mathbf{D}_{app}$ restricts the rule application depth. Although the search tree extends to depth $\mathbf{D}_{max}$, only a portion of some sequence of rules will be executed. A parameter $\mathbf{\Delta}_{class}$ was introduced to limit the number of rule classes to be examined beyond the current subclass. This variable decreases the number of rules that can be applied at any given time. Finally, the parameters $\mathbf{\Delta}_{cost}$, **R**, and **S**, relate to the cost function. Δ_{cost} limits the increase to the cost function by a single rule application. Parameters R and S are used in the cost function to determine the desireability of applying a particular rule. The cost function takes into account the area saved and the number of rules that can be applied after a transformation is made. The weighting of terms in the cost function affects the size of the tree by changing the desireability of various rules.

Initially SOCRATES used fixed values for these parameters, regardless of the optimization phase. However, the ideal parameters vary greatly over the course of optimization. For example, greater lookahead is required for area-saving rules than general rules. Also, little or no lookahead is required for the most powerful rules [GeCo85]. Thus based on the state of the optimization, metarules determine what values the control parameters should have. These rules supervise the control module and dynamically vary the parameters. Such a technique permits more

selective use of lookahead. Control parameters can be changed depending on the rule class, rule subclass, or even an individual rule.

Through this process of lookahead, the best sequence of rule applications can be determined. By examining the future effects of a rule, SOCRATES can determine whether the decision to apply a rule was good. Poor decisions can be undone through backtracking and other rules can be considered. The designers of SOCRATES report this approach produces superior results to those where only one rule is examined and then applied [CoBa85]. Their examples indicate that the use of lookahead without metarules required roughly four times longer on average to run, producing designs with 12 percent less area on average. Adding metarules only doubled the run time and still provided the same decrease in area.

LSS is a system that also employs limited hierarchy as it performs optimization at each of its four levels of representation. The LSS refinement stage produces an initial graph with AND/OR gates, registers, decoders, etc. The first level of optimization is performed on this network. It uses transformations that reduce the gate-level logic and transformations that use information about a high-level component to reduce the surrounding gates. Figure 7 demonstrates transformations of this type from [JoTr86]. The first rule uses knowledge about a decoder to eliminate the OR gate. The second rule in Figure 7 is a simple logic reduction transformation.

Once all level one transformations have been performed, the high-level components can be expanded into AND/OR/NOT gates. This forms the second

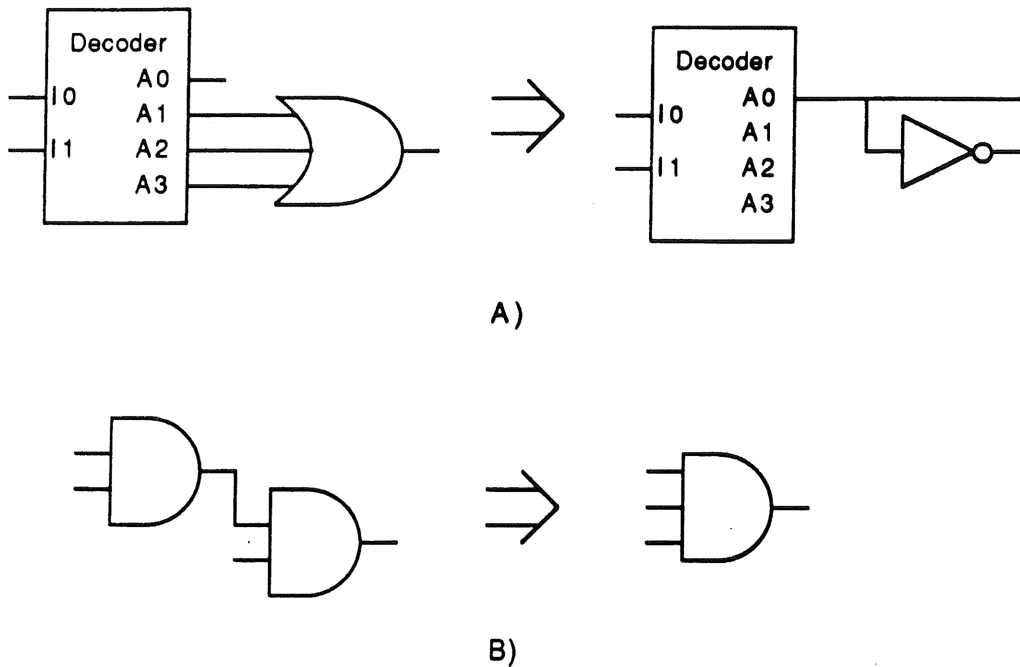


Figure 7. LSS Level 1 Optimizations

description type, the AND/OR level. At the AND/OR level, transformations perform AND/OR simplification, common subexpression elimination, and constant propagation (ie., OR(a,1) = 1, AND(a,1) = a) [DaJo81].

The third level, NAND/NOR, introduces some technology considerations. Depending on the technology, the design will be converted to one consisting entirely of generic NAND and NOR gates. The same type of transformations that were performed at the AND/OR level are applied at this level, only with NAND/NOR simplifications in mind this time. Transformations at this level attempt to

incorporate technology tables that supply information on generic NAND/NOR gates. For example, information on each generic primitive (such as a NAND3) is maintained on its size, driving capability, delay, fan-in, etc. [JoTr86]. Hence at the NAND/NOR level it is possible to make decisions about what type of NAND/NOR gates should be used. For example, LSS can tell whether to use three-input NANDs or two-input NANDs to reduce area.

The final level of description is technology specific. Transformations convert generic components to components from a technology library that may include complex gates such as AND/OR, multiplexors, etc. The transformations also enforce technology constraints such as fan-in and fan-out. To deal with complex gates, LSS makes use of tables that list the components available in each complex gate type (ie., AND/OR, decoder, OR/AND). In each category the components are ordered according to the savings created. When conflicts arise as to which transformation to apply, the one with the largest gain is selected based on this ordering.

The transformations performed at each level in LSS are executed through PL/I procedures that manipulate the logic graph. Transformations at the final two levels make use of a number of technology tables that can be readily updated for a new technology. It has been reported [JoTr86] that the use of local transformations as opposed to two-level boolean minimization tends to keep synthesis times linear for increasing design sizes. For circuits of 200 to 2000 two-input equivalent gates, a time of one second for roughly nine gates was reported (based on IBM 3081 CPU

time).

3.3.3. Algorithms Only Strategy

The final type of expert system is entirely algorithmic. It uses an algorithmic controller to determine how to apply algorithms in the knowledge base. Such a system performs the same operations always in the same order.

An example of an algorithmic system is **MIS** [BrRu87]. MIS is a set of heuristic algorithms that perform minimization and factorization. MIS optimizes boolean equations through a sequence of commands that can be entered interactively or controlled automatically via a user-written script that consists of a series of MIS commands.

Area optimization in MIS is carried out using three techniques: 1) extraction, 2) resubstitution, and 3) phase assignment. Extraction is the process of generating common factors from a set of logic equations. For example, consider the equation:

$f = ad + bcd + e.$

It has common divisor g = a+bc that can be extracted to produce f = gd, and common divisor h = a+b that can be extracted to produce f = hd(a+c). Possible common factors are generated by a set of heuristics. Area is estimated by counting the number of literals in all of the boolean equations. The area improvement obtained by extracting a particular common factor is then the number of literals saved.

Because MIS uses heuristics to identify common factors, it may not find all factors that can be extracted. For this reason, resubstitution is used to ensure that all common factors of a special type are identified. Resubstitution is the process of checking whether an existing function is a factor of another function in the network. For example, consider the two functions:

$$x = ac + ad + bc + bd + e$$

$$y = a + b.$$

Function y is a divisor of x and x can be rewritten as:

$$x = y(c + d) + e.$$

Phase assignment attempts to reduce the total number of inverters in the design without increasing the size of other functions. Essentially, this involves determining whether a function or its complement requires fewer inverters. This includes checking intermediate functions (ie., subfunctions) for inverter reductions.

Another algorithmic based system is **DAGON** [Ke87]. DAGON performs technology binding and can optimize for time, area, or some function of the two. It utilizes programming-language compiler techniques that are strictly algorithmic. In doing so, DAGON can guarantee locally optimal matches over several thousand patterns.

Similar to language compilers, DAGON matches a graph description of a technology-independent circuit against a technology library consisting of numerous patterns. The problem is viewed as finding the best technology patterns to cover a

directed acyclic graph (commonly termed DAG). A DAG is a graph containing no cycles that consists of nodes and directed edges. DAGON builds a DAG for boolean functions using nodes to represent AND and OR operators and edges labeled 0 or 1 to indicate a true or inverted output (thereby providing NAND and NOR operators as well).

A globally optimal solution to the DAG covering problem could be generated by comparing all possible technology implementations (each a collection of patterns from a technology library) for time and area. However, such a problem is NP-complete. Therefore, DAGON's first step is to partition the graph into trees. This is accomplished by making every component in the graph whose fanout is greater than one, the root of a new tree. DAGON may then find the minimal cost technology pattern for each tree, producing a locally optimal solution.

Finding a minimal cost match for a tree consists of two major tasks: recognizing the set of possible matches and determining the pattern from that set providing the minimal cost (in terms of time and area constraints). Finding applicable pattern matches is performed by *twig* [Tj86], a program designed to construct code generators for programming language compilers. It generates the set of matches in O(TREE SIZE) time. From this set the minimal cost match can be found using a recursive algorithm that determines the least cost match for each subtree.

3.4. Optimization Strategies

The quality of a synthesized design can vary dramatically depending upon the strategies used to optimize it. Strategies can be examined at both a high and a low level. At a high level there are essentially two types of decisions for which a strategy must be chosen. They are: what to optimize for and what subsection of the overall design to optimize at any given time. At a low level there are also two types of decisions: what types of transformations or algorithms to apply and what order to apply them to the specified subcircuit.

The first decision depends heavily upon the user-entered constraints. Typically three types of constraints are entered: time, area, and power. The optimizer must choose which order (if any) to optimize the constraints and which constraint(s) should carry the greatest weight in directing the optimization.

The second decision concerns the portion of the design toward which optimizations should be directed. One could simply search the entire design for all the rules that are applicable and select one to apply. This has the effect of applying transformations randomly throughout the design. Generally the technique is time consuming and produces less than optimal results. Human designers break up large circuits into smaller ones, making optimization more manageable and allowing more control over the course of the optimization. This approach is best for optimization programs as well. By focusing on only a small section of the design, one not only reduces the number of rules that must be considered but also allows a rule's effect

to be more closely examined -- one need only consider its effect in the subcircuit to know what happens in the larger design.

When performing timing optimizations one tends to select a path-oriented approach. Thus the subcircuit might be a critical path. Area optimizations are not path directed except for avoiding critical or near-critical paths. In this case, critical paths could be removed to eliminate the time wasted by considering rules that affect some component that is part of a critical path.

The third decision involves the specific rules or algorithms that can be applied. This decision is in part based upon the earlier issue of what to optimize for. Certain techniques work best when optimizing for time, others when optimizing for area. As an example, consider a rule that greatly decreases delay but improves area only incrementally. Clearly such a rule is best suited for use in timing directed optimizations. An alternative rule that produces greater area improvements can be used when optimizing for area.

The final decision involves the order in which optimization techniques should be applied. The use of lookahead can aid greatly in assuring that the "best" rules will be used. Lookahead, however, can be quite time consuming and simply considering certain rules before others may be a better solution in some cases. Another example of the use of ordering can be seen in the use of algorithms. For instance, some algorithmic techniques, such as collapsing a network into two levels to reduce delay, require a great deal of time and effort. Yet if only a small improvement is

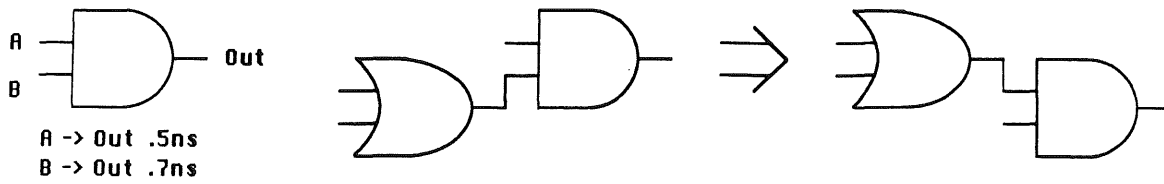
required, the effort is, in effect, wasted. Another technique, perhaps a rule-based approach producing smaller gain, could have been used.

3.4.1. Control Strategies for Timing Optimization

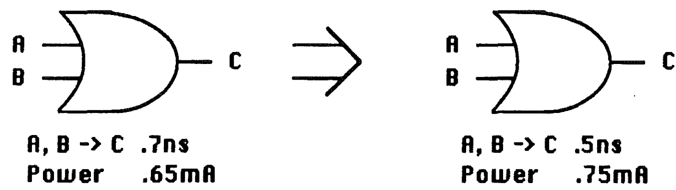
Different control strategies are required for timing optimization. For example, critical paths whose delays differ greatly from user specifications tend to require some type of circuit restructuring. In contrast, those critical paths that are close to the specifications may require that only a few gates be replaced or only a small portion of the critical path to be modified.

3.4.2. Types of Strategies

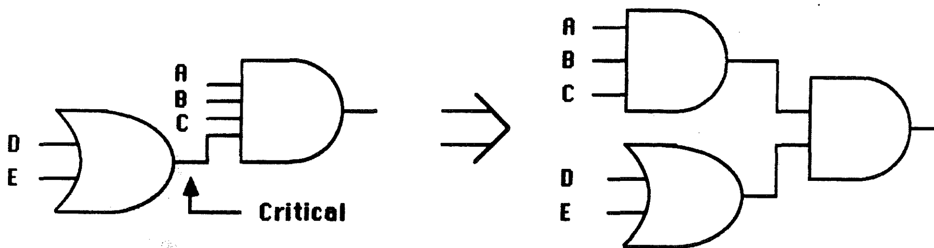
There are a number of different strategies that can be used to improve speed. These are illustrated in Figure 8. Strategy 1 swaps equivalent signals on the same component. For example, the 2-input AND gate in Figure 8a has a different delay from each input to the output. Thus the critical path should be connected to the input with the shortest delay. Strategy 1 can be implemented by examining a technology file that contains timing information on each component. This strategy has no cost as it does not change any circuit elements. However, the gain produced is small. Thus strategy 1 is useful when the slack (difference between the actual and required times) is extremely small or when strategies that produce larger gains have been exhausted.



(a) Swap equivalent signals on the same component

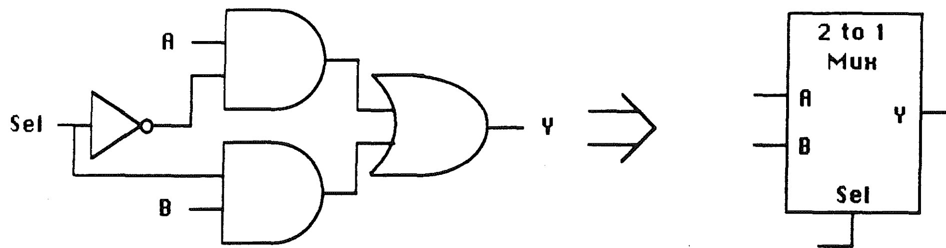


(b) Replace macro with one of higher speed

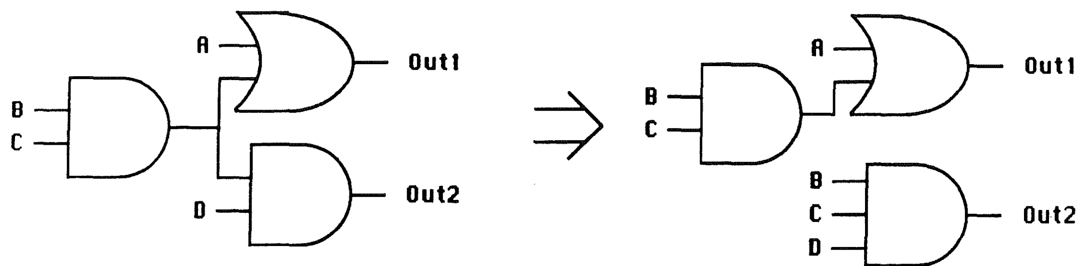


(c) Factor

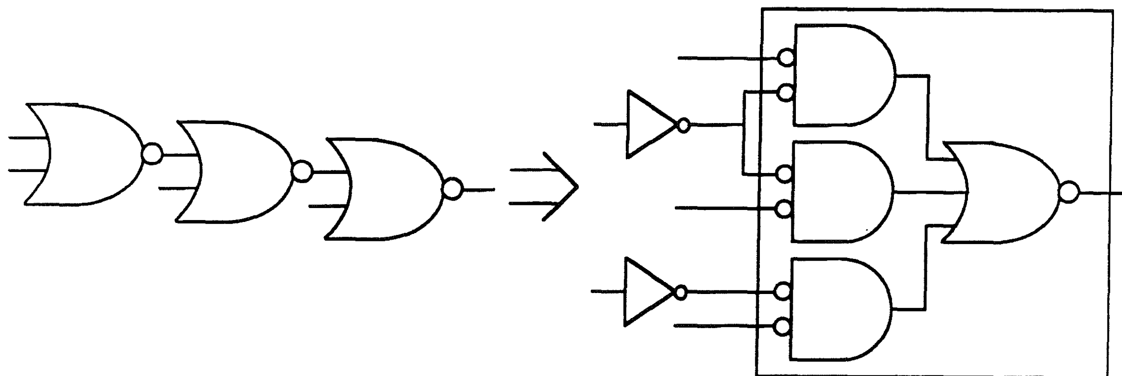
Figure 8. Techniques for Reducing Delay Along Critical Paths



(d) Better macro selection that does not increase area or power

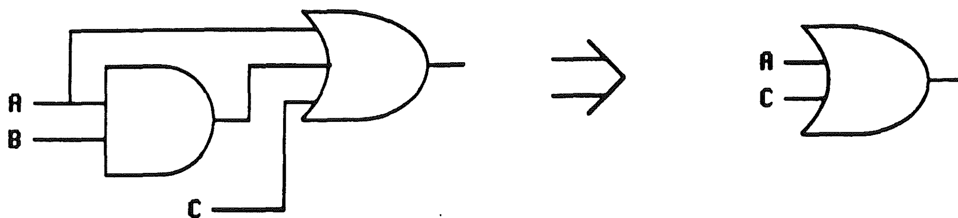


(e) Duplicate logic

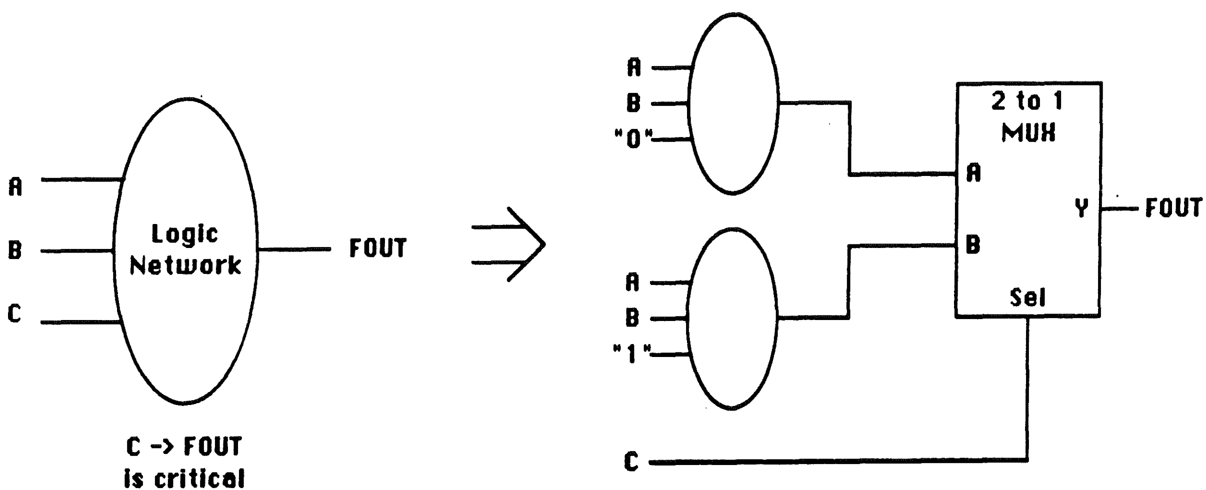


(f) Better macro selection at the expense of area/power

Figure 8 (cont.)



(g) Minimize



(h) Duplicate logic with multiplexor

Figure 8 (cont.)

Strategy 2 replaces a low-power or standard-power macro with a high-power macro of greater speed. This type of strategy replaces a macro with another that uses larger transistors to provide faster switching times. Strategy 2 can be implemented by examining the technology file to determine if the same macro exists with higher power, higher speed. The strategy is similar to strategy 1 as it produces only a small gain. Thus it would be used in the same situations as strategy 1. However, since strategy 2 increases the power, strategy 1 is preferred.

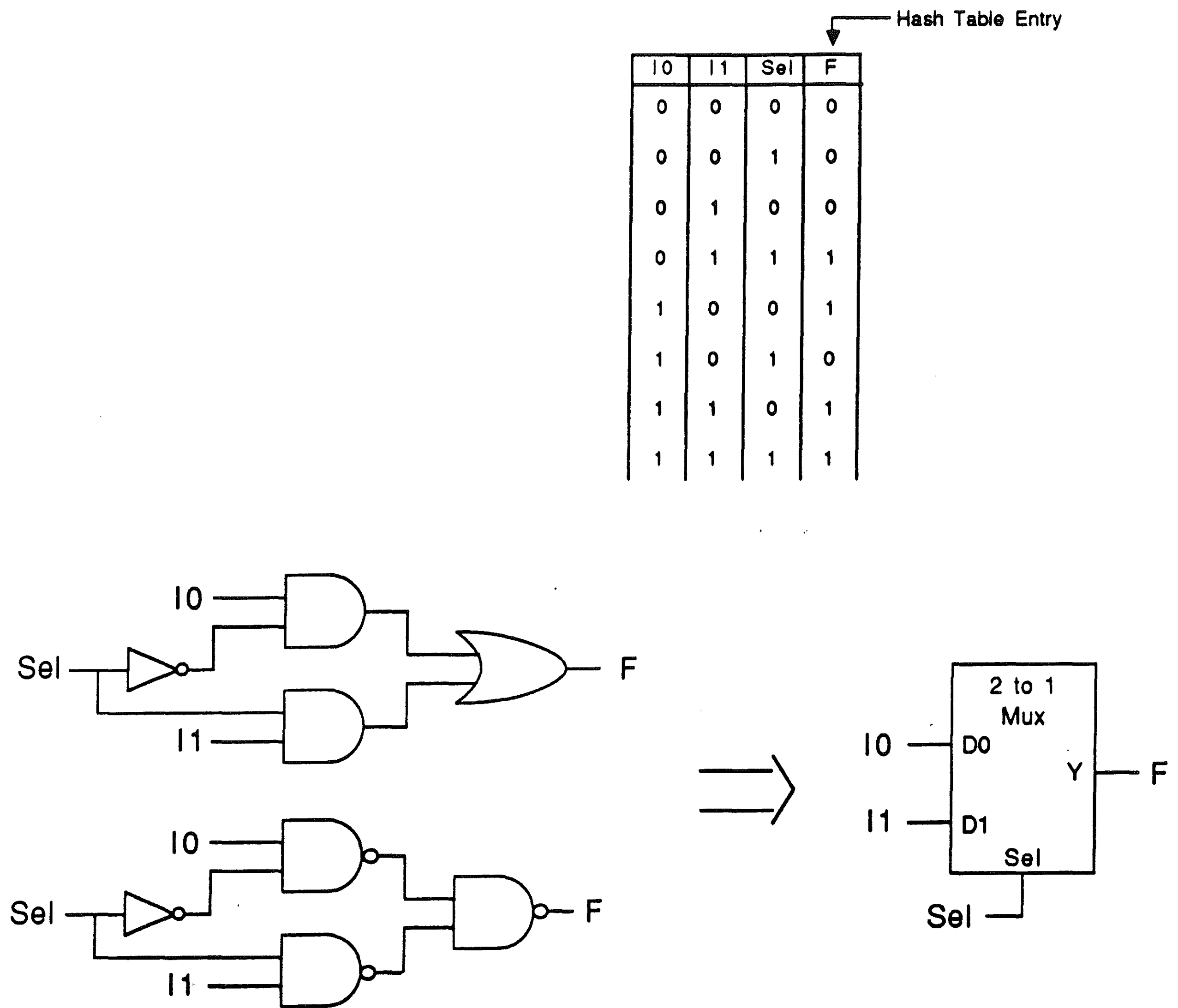
Strategy 3 employs factorization to reduce the delay of a critical path. Figure 8(c) shows how a four input AND gate can be factored to speed up path D. Implementation is a factorization program such as that found in MIS [BrRu87] or ESPRESSO [Br84]. The gain is typically small though greater gains are achieved for larger gates. In addition, using factorization along the entire critical path can add up. Power generally increases but the effect on area may vary. Strategy 3 tends to produce a slightly larger gain then strategies 1 or 2. Hence, it will be used when the slack is small.

Strategy 4 attempts to make a better macro selection that reduces time without an increase in area or power. This strategy may utilize a rule base containing only rules that will produce a gain for no cost. For instance, a rule could be written in OPS83 to perform the transformation of Figure 8(d). An alternative method is use of a hash table. Lookup in the hash table is accomplished through a key that is the truth table entry for a particular function. The hash table is typi-

cally limited to entries of up to five variables, making each hash table key a maximum of 32 bits -- a common computer word. Certain transformations cannot be represented by a hash table. For example, functions with multiple outputs, such as a decoder, or circuits with sequential logic elements, such as a counter. In these cases it is necessary to use the rule-based approach. A hash table has an advantage over the rule-based approach in that fewer transformations need to be entered. For example, Figure 9 shows two different implementations of a multiplexor that can be represented by the same hash table entry, but require two separate rules. Another advantage of hash table lookup is speed. It requires only one table lookup per function. Of course time is required to find a circuit's function but this may require less time than having to search through a set of rules to determine which are applicable. Since Strategy 4 has no cost, it will be used before strategies 2 and 3, and will be the first strategy examined for moderate gain.

Strategy 5 duplicates logic along a critical path, thereby doing the reverse of common term factorization. Like strategy 4, it can be implemented using either a rule-based or hash table approach. The gain from strategy 5 is typically small and hence the strategy would be applicable at the same time as strategies 1 - 3. As the cost in area and power tends to be greater than strategies 1 - 3, strategy 5 would be the last examined.

Strategy 6 is similar to strategy 4 except a better macro selection is made with an increase in area and/or power. It can make use of a hash table or rule based



I0	I1	Sel	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Figure 9. Use of a Hash Table to Reduce the Number of Rules

approach as well. Typically a moderate gain can be achieved through strategy 6 with a small to moderate increase in the power and area constraints. It is often considered for moderate slack improvements or for large slack improvements after large gain strategies have been examined.

Strategy 7 is the foundation of both SOCRATES and the Logic Consultant. It expands the design into two-level SOP form then minimizes by removing redundant terms. This strategy is often combined with strategy 3 to factor the circuit along non-critical paths and take advantage of common terms. This strategy is the most time consuming but can produce large gains, often with no increase or only a small increase in area and power. Thus this strategy is the preferred method for improving large slacks.

Strategy 8 is one discussed in [JoMc87]. It duplicates logic that strategy 5 can't by adding a multiplexor. Figure 8(h) has a function FOUT = F(A,B,C) where C -> FOUT represents the critical path. The logic network may be duplicated with the C input connected to GND in one, and VDD in the other. The real C input is then hooked up to the select input of a multiplexor. The gain from Strategy 8 is large but so is the cost if little optimization can be performed after increasing the amount of logic. New circuit elements are added which increases both area and power. Hence, strategy 8 will be examined for a large slack but will be considered after less costly strategies have been attempted.

3.4.3. Choosing a strategy

The control strategy can be changed depending on how far the critical path is from the timing constraints. When the time difference is small, a local optimization can be attempted using some combination of strategies 1 - 4. Rules from three different categories can be examined: those that do not increase area and power, tradeoff rules that improve speed but increase either area or power, and tradeoff rules that increase both area and power. Those rules in category 1 will be considered before examining rules in categories 2 or 3. Similarly category 2 rules are preferred over those in category 3.

Within each rule category, the smallest rule with the maximum gain will be chosen. Thus rules involving only two inputs will be examined first. If one of them meets the timing specifications, that rule will be applied. Otherwise rules with three inputs must be examined. This process continues until all rules have been examined at which time a new strategy must be applied.

When the time difference is great or all other strategies fail, the circuit can be minimized into a two level circuit using strategy 7. It can then be expanded through weak division into multiple levels as in strategy 3.

CHAPTER 4.

THE MILO SYSTEM

4.1. Introduction

This chapter discusses some of the previous approaches in converting microarchitecture designs to a technology-specific design and presents some of the unique features of the microarchitecture optimizer in the MILO system. In addition, the synthesis framework in which MILO is used is described.

4.2. Previous Work

Various approaches have been taken to convert the microarchitecture design into a design that can be passed to a layout tool. Some tools extract the behavior as a set of boolean equations and flip-flops, then rely heavily on logic synthesis tools to reduce the logic and make an efficient design [Br86] [StMu86] [TsWe88] [WeRo88]. They employ logic generators that produce a design of generic logic gates from each microarchitecture component's description, then use tools such as [BrRu87] to reduce the number of gates in a component, restructure critical paths, and map the design into a particular standard cell or gate-array library. The design

can then be passed to a standard cell or gate-array layout tool.

SILC [GuPa90] includes a component rearchitecting step that selects a different style of architecture for components along a critical path. For example, a ripple carry adder can be converted to a carry-lookahead adder or something in between to improve the speed. Thus the style of component can be changed after logic optimization fails to meet the necessary constraints.

Other behavioral synthesis tools designed for datapath generation can base their architecture on a standard cell layout or a bit-sliced layout [TrDi89]. Optimization is carried out for that particular layout style. Still another approach is to construct the design using off-the-shelf components [BiBr88] including microprocessors, DMA controllers, dynamic RAMS, etc.

Numerous tools for behavioral and logic synthesis have been previously reported. This chapter describes a system that fills the gap between the behavioral synthesis tools and logic synthesis tools by using a microarchitecture optimizer. Behavioral synthesis tools often use estimators in design refinement. These estimators may be technology independent and are usually not accurate enough to make decisions for fine tuning the microarchitecture design. On the other hand, logic synthesis tools can accurately gauge area and time but operate on too low of a level to adequately make microarchitecture modifications.

In chapter 6, techniques are introduced for improving the microarchitecture structure and for employing constraint driven synthesis based on the user's

requirements for time and area. These techniques include the capability for mixing layout styles such as custom layout for random-logic components and bit-slicing for regularly structured components. In this manner the entire design, control logic and datapath, can be optimized at the same time. Further, a new methodology is presented for microarchitecture-level optimization that greatly reduces the amount of technology-specific knowledge necessary to perform the optimizations. Microarchitecture components are generated by a database based on a set of parameters from the microarchitecture optimization tool. Thus the microarchitecture optimizer does not need to deal with multiple logic optimization tools, layout module generators, transistor sizing tools, etc.

Often the structures produced by behavioral synthesis tools contain inefficiencies such as constants that can be propagated through a design, and common subexpressions that appear multiple times in the design, each time with replicated hardware. These can partly result from the fashion in which the user wrote the behavioral description. Also the design needs to be directed towards a certain set of constraints for time and area. Tradeoffs must be made along different paths. On critical paths optimizations that reduce time are required, possibly at the expense of increased area. Non-critical path optimizations attempt to reduce area as long as doing so does not create a new critical path. In performing these tradeoffs, the microarchitecture optimizer can select a different architectural style for the component, merge components and reoptimize their logic, insert buffers to improve drive capability, replace a set of components with a single component that

performs the same function but more closely meets the constraints, restructure components to reduce delay (such as factoring multiplexors), duplicate logic to reduce delay, or change the layout style of the component. These type of improvements are nearly impossible to pursue once the design has been expanded into lower level logic.

4.3. System Architecture

This section describes a microarchitecture optimization tool and illustrates how it fits in to a larger system that synthesizes layouts from VHDL behavioral descriptions. The system architecture is shown in Figure 10. It consists of six major pieces: a component database, logic optimization tools, a behavioral synthesis tool, a technology mapper, a microarchitecture optimizer, and a floorplanning/layout system.

4.3.1. Component Database

The central system tool is a component database [ChGa90]. It supplies components and statistics on components to the synthesis tools. Synthesis tools can pass a set of parameters and specifications to the database and then receive a list of components that meet the requirements. Parameters include the component type (eg., ALU, Counter, MUX), number of inputs, clock type (rising-edge, falling edge), etc. Specifications include the load that each output pin must drive, the maximum delay to each output pin, and an area requirement.

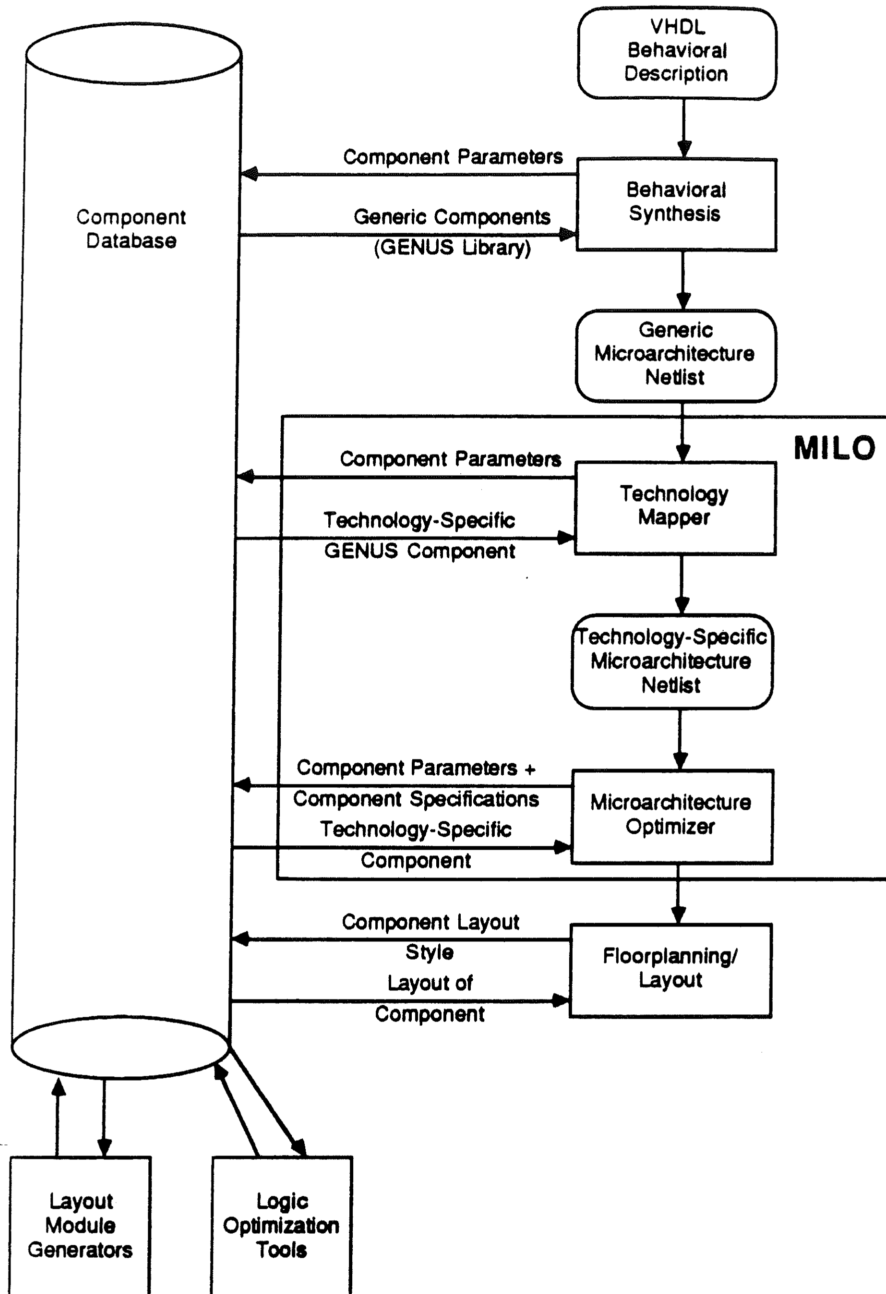


Figure 10. MILO Environment

The component database contains a library of logic generators that produce a boolean equation representation that describes the low-level behavior of the component. One or more generators can be selected based on the parameters supplied by the synthesis tool. The boolean equations include constructs for describing sequential logic so that logic generators for components such as registers and counters can be constructed. The boolean description is passed to a logic optimizer [VaGa88] with a set of time constraints. The logic optimizer produces a technology-specific design using components from a designated library or can generate complex gates and select transistor sizes for use in a custom layout. The logic optimizer produces a report file listing delays and area. This information can be passed to synthesis tools when they request such information about a component.

The database also contains knowledge about components that can be produced by layout module generators. Estimators provide data on delay times and area based on the bit-width.

4.3.2. Behavioral Synthesis

A behavioral synthesis tool [LiGa89] accepts a VHDL behavioral description and produces a VHDL structural netlist consisting of generic components from GENUS [Dutt88], a library of generic microarchitecture components. One special property of GENUS components is the use of one control line per function. Thus a four-bit multiplexor has four data-in lines and four select lines -- one to control each data line. In an ALU, there are separate control lines for ADD, SUBTRACT, AND,

OR, etc. This component property removes the problem of control encoding from behavioral synthesis as component encodings may depend on a particular technology library. If necessary control encoding can be performed later during technology mapping.

The behavioral synthesis tool begins by converting the input description into a dataflow graph. A graph critic then operates on the dataflow graph, removing redundancies in the behavioral description. The behavioral operators are then bound to GENUS components. The final architecture produced by the behavioral synthesis tool consists of random logic blocks of control logic and a datapath containing components such as ALUs, shifters, and registers.

The components in the generic netlist are converted to technology-specific components by a technology mapper. The technology mapper queries the database by providing the set of component parameters. The database returns one or more components that meet the specified parameters. From this set of components the technology mapper selects the component that contains the smallest set of functions required. For example, if a component with the ADD and SUBTRACT functions is requested, the database may return two components: an ADD/SUBTRACT unit and an ALU. The technology mapper would select the ADD/SUBTRACT unit. Since the technology mapper does not pass a set of timing or area constraints to the database, the database will return the most area efficient design. Currently the technology mapper maps generic components into only components that are imple-

mented from gates and optimized by the logic optimizer. Later implementations will include mappings of other types of components, such as those from layout module generators. In any event, these types of components are currently inserted later, during the microarchitecture optimization phase if appropriate.

4.3.3. Microarchitecture Optimization

At this point the design consists of two levels. One is the microarchitecture netlist, the other is a technology-specific gate-level netlist for each microarchitecture component. The microarchitecture optimizer first employs rules that make transformations that should improve both time and area. For example, converting a register and incrementer into a counter. Next the critical paths are identified. The optimizer requests faster components from the database, selects different layout styles (random logic or bit-sliced), and decides which components to merge and apply logic optimization. Once critical paths have been processed, the microarchitecture optimizer operates on non-critical components, making similar decisions as in the critical path improvement phase but this time with an eye toward area improvements. The microarchitecture optimizer then produces a VHDL netlist that is passed to the floorplanner/layout assembler for layout.

The microarchitecture optimizer uses a new methodology for selecting microarchitecture components to be used in the design. The microarchitecture optimizer does not perform component rearchitecting and does not have knowledge of tools for logic optimization, transistor sizing, and other component reoptimization

techniques. Instead, these tasks are left to the component database. The microarchitecture optimizer passes a set of time/area constraints to the database and the database examines possible ways to achieve the constraints. The database can choose from different architectural styles and can choose from multiple optimization tools to redesign the component. This frees the microarchitecture optimizer from dealing with technology concerns and having to know what set of component optimization tools exist at any one time. All of this is centralized in the database.

Integration of the database with the microarchitecture optimizer and the logic optimization tools is achieved with two servers [ChGa89] as shown in Figure 11: a component server, and a knowledge server. The component server is the part that interfaces with the microarchitecture optimizer. Queries are made from the microarchitecture optimizer to the component server through the Component Query Language (CQL) and a list of components or a set of component attributes are returned. In this manner, the microarchitecture optimizer can simply request the functions required of a component, an layout implementation style, and a set of delay parameters. From this information the component database checks its component list which includes fixed components (components that have already been generated) and parameterized components (those that can be generated when provided a set of parameters). The component database knows from its component list whether a component generator needs to be called to generate a design for the component or whether the component design already exists (as in the case of a fixed component). Once a component is generated, the database can call an appropriate

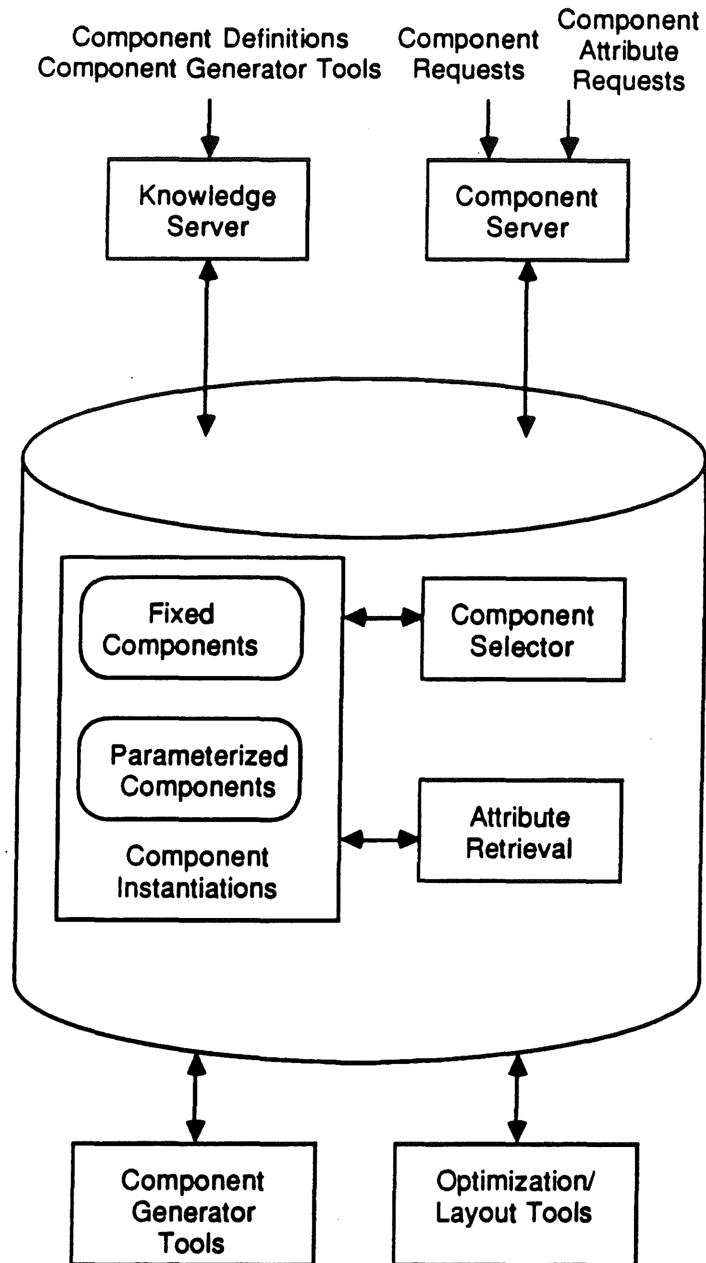


Figure 11. Tool Interface with the Component Database

logic optimization tool or layout tool.

The knowledge server is used to insert new fixed components, insert new component generators, and insert logic optimization and layout tools. Thus when a new logic optimization or layout tool is available, the knowledge server will be accessed to store information about how to call the new tool. Also designers can build their own components and insert them into the component database through the knowledge server.

4.3.4. Floorplanning/Layout

Finally the technology-specific microarchitecture netlist is passed to a layout synthesis system that performs floorplanning on the microarchitecture design and creates a layout for each component [WuGa90]. The layout tool decides how to partition the random logic into blocks for layout. It also can modify the microarchitecture optimizer's selection of bit-sliced components, converting them to random logic if doing so will result in a better layout. Module generators are called to produce the bit-sliced layouts and a custom layout generator called to produce a layout for each random logic block. The floorplanner selects how to partition the random logic based on shape sizes that can be used to fill in the bit-sliced logic mismatches.

The floorplanner attempts to place similar-sized bit-sliced components together and place random logic into slots where mismatches in the length of the bit-sliced logic occurs. Other random logic is placed along the border of bit-sliced components.

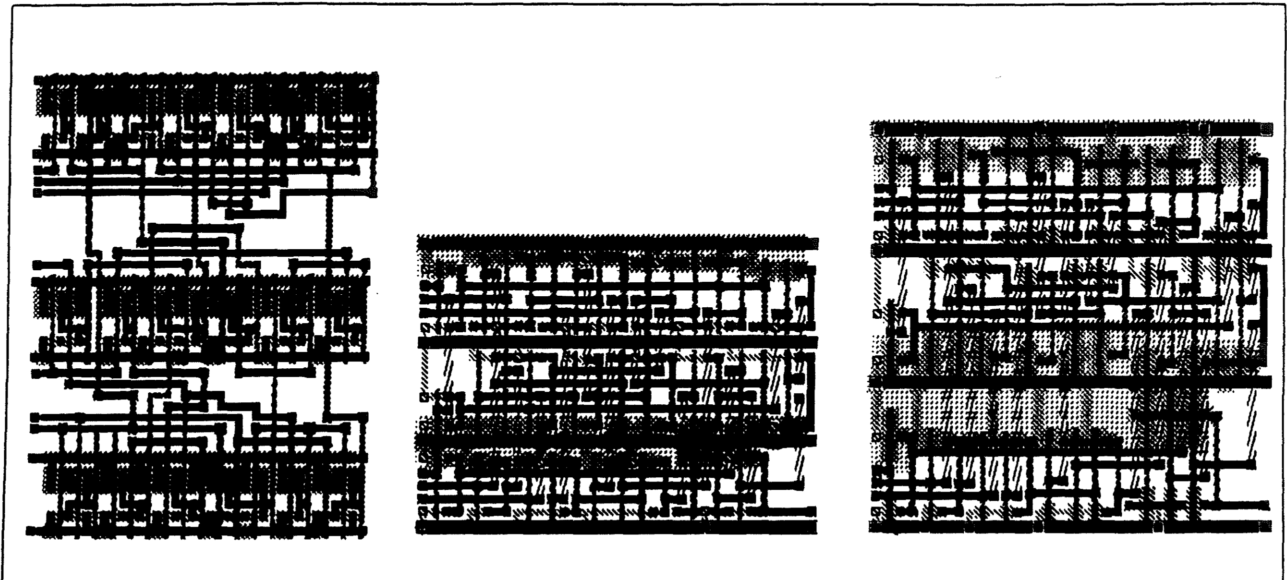
CHAPTER 5.

LOGIC SYNTHESIS FOR CUSTOM LAYOUT

5.1. Introduction

Conventional logic synthesis systems produce designs for gate array and standard cell libraries. Such semi-custom libraries are popular for their simplicity in layout and the ability to use automatic layout tools. A major weakness of the semi-custom approach is the larger area required for routing, producing a lower layout density than custom methods. In addition, all transistors are of the same size, capable of driving some expected average load. This average load is usually greater than what is required, making cells larger than necessary. When a larger load than the preset value must be driven, speed is sacrificed.

In order to achieve better layouts, more attention has been focused on tools that offer custom layout. Custom layout permits greater control over layout parameters, providing smaller area and faster designs than its gate-array or standard cell counterparts. For example, Figure 12 shows three layouts for the same logical function. The first layout was done using standard cells; the second and third, using custom layout, were optimized for area and time, respectively, by varying transistor sizing and complex gate formation. The second layout has 22% less area than the standard cell version and the third layout is 30% faster than the standard cell ver-



(a) Standard Cell Layout (b) Custom Layout for Area (c) Custom Layout for Speed

Figure 12. Standard Cell Layout vs. Custom Layout

sion.

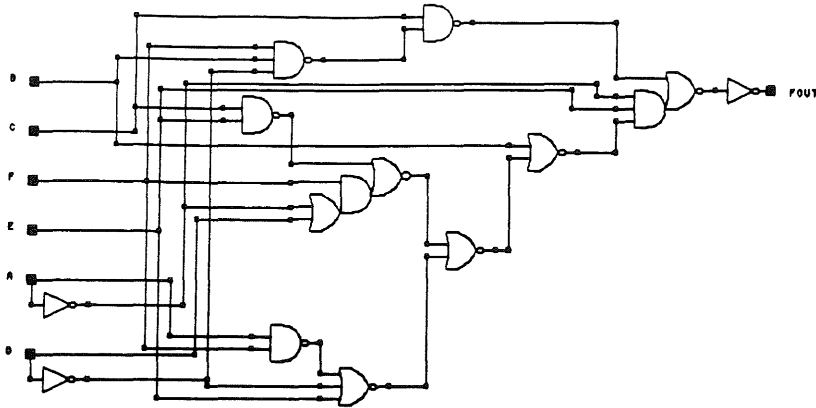
In custom layout, cells are generated dynamically instead of having fixed libraries and transistors can be sized according to the load that must be driven. Thus while standard cell and gate-array layouts are very important for quickly bringing chips to market, custom layouts still provide faster designs with smaller area. When performance is crucial or a large volume of chips is needed, custom layout is necessary. Custom layouts can be generated from a gate-level netlist either by hand or by custom layout generators such as LES [LiGa88], CLAY [KoLu88], and CLEO [DoLe89].

This chapter introduces techniques to generate gate-level netlists that take full advantage of the custom layout capabilities. Such techniques include limiting serial/parallel transistor chains, transistor sizes, and capacitive loads in forming complex gates. These considerations have not been incorporated in previous logic synthesis systems. Hence the intended results sometimes fail to be achieved when a custom layout is used. By extending existing logic synthesis methods that were aimed at standard cell layout, we can produce gate netlists that assure good results for custom layout as well. For example, consider Figure 13 which shows two different implementations of the same logical function. The design in Figure 13(a) has been optimized specifically for area. Figure 13(b) displays the implementation that has been optimized for time. Both designs were produced using standard cell synthesis techniques. When the layout was implemented in standard cells, the optimization achieved the desired effect. However, when these implementations were run through custom layout generators, the design of Figure 13(b) had the best speed AND the best area as shown in Figure 13(c).

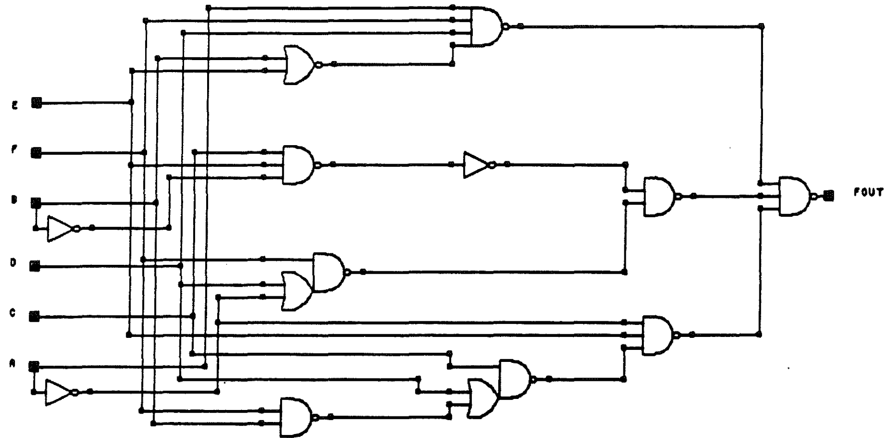
5.2. Parameters for Custom Layout

5.2.1. Complex Gates

Layout driven synthesis is not restricted to a fixed library of components. In many cases multi-level Boolean functions can be implemented as a single complex gate. Complex gates have fewer connections and fewer transistors than multiple



(a) Optimized for Area



(b) Optimized for Time

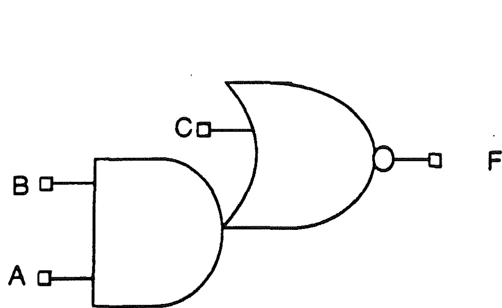
Optimized for:	Custom Layout		
	# of Tran-sistors	Layout Area (um^2)	Delay (ns)
Speed	56	56048	9.61
Area	54	59800	9.78

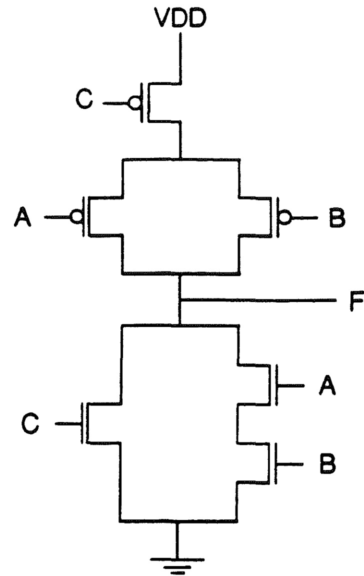
(c) Layout Comparison

Figure 13. Custom Layout Results Using Synthesis for Standard Cells

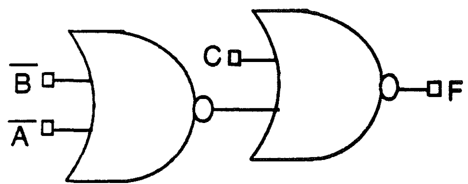
gate implementations and hence may reduce area and improve performance. An example of a complex gate is shown in Figure 14(a) and contrasted with the multiple gate implementation in Figure 14(b) (with inverters assumed to be pushed back to previous levels). Both implementations are shown in the CMOS technology with cells having a P and N transistor section.

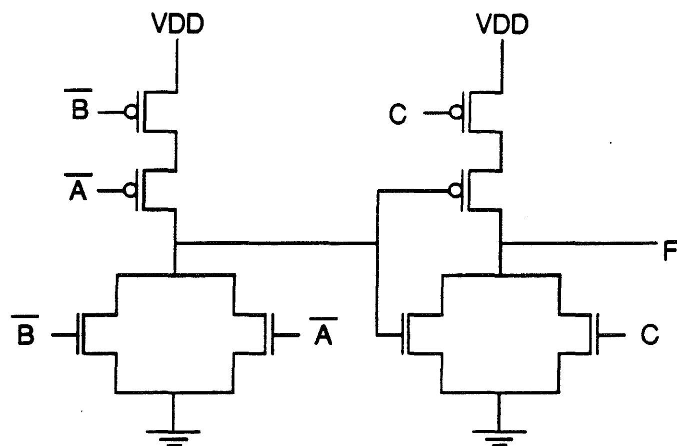
In layout driven synthesis, the optimizer has great flexibility in deciding which gates to combine into a single gate. The type of complex gates formed is limited mainly by the buildup of capacitance, the load the gate must drive, and the transistor speed. As more transistors are placed in series, the resistance and parasitic capacitance of the gate increase, resulting in longer delay times. In CMOS, where gates have an N and P transistor section, carrier mobility in P-type transistors is twice as slow as N-type transistors. NOR gates, which have P-type transistors in series, can be 2 to 3 times slower than gates with N-type transistors in series (such as NAND gates). This makes it beneficial to limit the number of transistors that a complex gate has in series and parallel. In CMOS, fewer P-type transistors should be allowed in series than N-type transistors. Hence, a compromise must be made between the solution consisting of only single gates and the solution involving complex gates with a large number of transistors. As a general rule, complex gates with few transistors are desired along paths where timing is critical while complex gates with more transistors are desired in sections where area is most important.





(a)





(b)

Figure 14. Complex Gate Formation

5.2.2. Transistor Sizing

Another concern in custom layout is transistor sizing. As discussed in [FiDu85], [He87], [Ci87], and [ObKa88], choosing proper transistor sizes is key to circuit performance. Figure 15 demonstrates how changes in transistor size affect speed. Increasing the size of transistors in Gate B improves its speed. However, the larger transistors create a larger capacitive load for Gate A. This slows down Gate

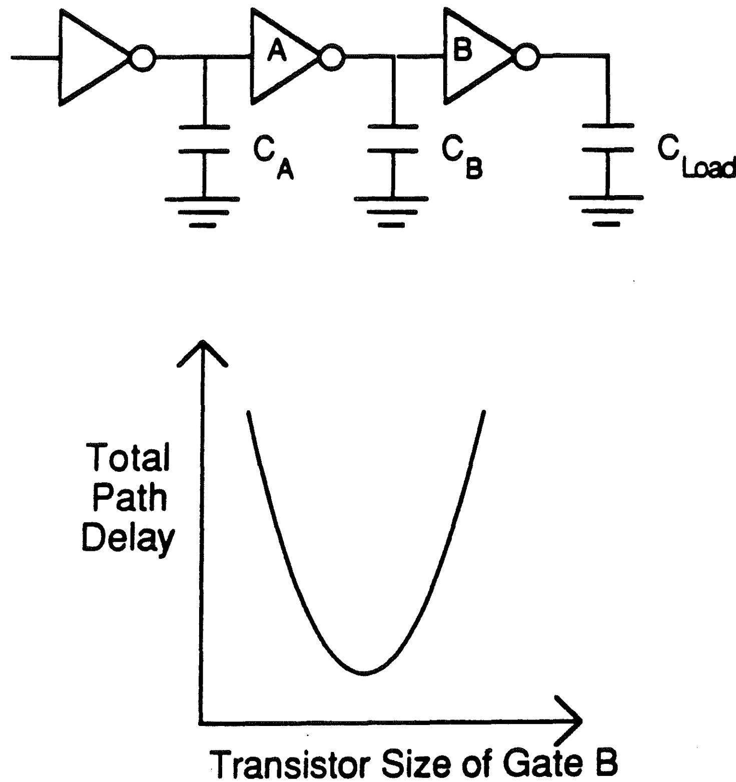
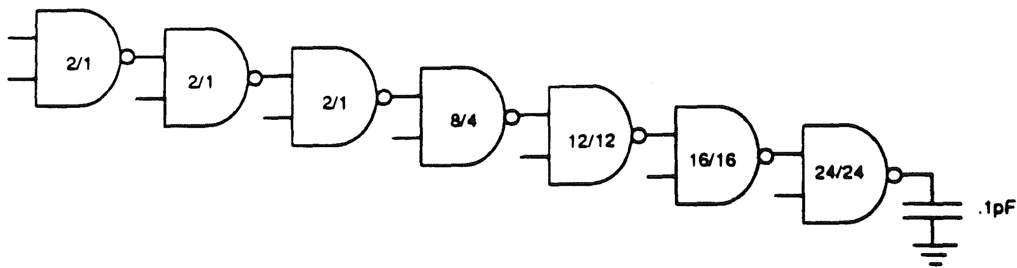


Figure 15. Effect of Transistor Sizing on Path Delay

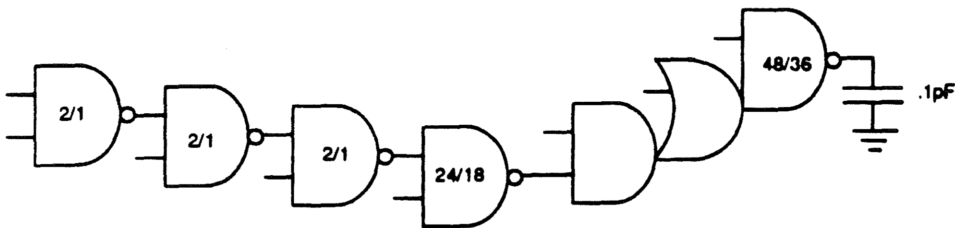
A and hence the entire path unless Gate A's transistors are increased based upon the new load. The size vs delay relationship is a convex curve as in Figure 15(b). Thus sizing a gate's transistors excessively large can increase the total path delay if the transistor sizes of the gates that drive it are not increased as well.

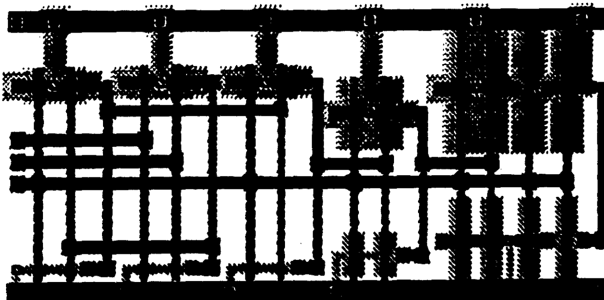
Transistor sizing gives greater control over area/speed tradeoffs. For example, in CMOS where NOR gates tend to be slower than NAND gates, synthesis systems try to avoid using NORs or use NORs with fewer inputs. Layout driven synthesis systems can increase the size of the P transistors, thereby improving speed (i.e., making the gate as fast as a NAND) at the expense of increased transistor area.

Since transistor sizing is not variable in standard cells there is no need to consider load when using complex gates. However, for custom layout, gates must be combined in such a way to avoid a number of problems. For example, as the load a complex gate drives increases, transistor sizes must be made larger to prevent a decrease in speed. Larger transistors require strips with greater height creating the problem shown by the two layouts in Figure 16 The first layout consists of seven NAND gates. In the second layout the final three NAND gates have been combined into a complex gate. Both designs drive an output of .1 picofarads (roughly 2-4 fanouts). Even though the transistor area for the complex gate is less than that for the individual gates, total area has been increased. One can see from Figure 16(b) that a tall cell in a strip is not compatible with shorter cells, generating wasted space along the top and bottom of the channel. In addition, larger transistor sizes



Area = 312 * 90 = 28080um^2, T_{rise} = 14.78ns, T_{fall} = 13.89ns
(a)





Area = 256 * 122 = 31232um^2, T_{rise} = 14.44ns, T_{fall} = 15.11ns
(b)

Transistor sizes shown inside gates as PFET size/NFET size
Only the last 4 NAND gates' sizes are shown

Figure 16. Effect of Large Transistor Sizes in Complex Gates

increase the load for all gates in the preceding stage, requiring them to be increased in size if the present speed is to be maintained. This creates a chain reaction affecting all preceding stages that can significantly increase the area. The design of Figure 16(b) is slower than that of Figure 16(a). The delays could be made equal by using even larger transistors sizes, resulting in a much larger area for the same delay. Hence the formation of complex gates should be dependent upon the load. **Single gates or complex gates with few transistors should be used to drive heavy loads.** In general capacitive loads are larger closer to output pins and smaller closer to input pins. This indicates that complex gates with a larger number of transistors can be created near input pins, while fewer transistors should be used in complex gates near output pins.

5.2.3. Placement and Routing

Control over component placement is important as cells along paths having critical delays should be placed close to one another. This prevents long wires from connecting them and introducing further delays. Further, large size gates should be placed on the edges of the floorplan since routing is sparse near boundaries. Same size gates can be placed in the same row to reduce wasted space from uneven cells. This increases the routing but decreases overall area. Such a scheme could have been used to correct the problem of Figure 16(b).

A related concern is in routing. Routing should be performed on critical components first to ensure that they get the shortest paths. In addition, long wires

should be placed in the metal layer for better speed. A synthesis program can aid layout by assigning priorities for the layout program, indicating which components need to be placed close together or which cells are similar in size.

5.2.4. I/O Positioning

Placement of I/O pins also is crucial for quality layouts. Pins can be placed near the modules having critical timing or placed in a position that reduces overall routing and saves area. They must be kept well distributed to prevent congestion in the center of the module and wasted space around the boundary of the module. Thus information on good I/O positioning can help layout tools create faster or more dense layouts.

5.3. Strategies for Layout Driven Synthesis

There are several strategies for controlling complex gate formation and transistor sizing.

One simple strategy is to form complex gates first and then size the transistors. During the complex gate formation stage, transistors are assumed to be unit sized. For example, in a 3 micron CMOS technology all NFETs and PFETs would initially be given a size of 3 microns. The weakness of this strategy is that transistor sizes do not influence complex gate formation and thus create complex gates may be created with many large transistors. This unnecessarily increases the layout area.

Using a second strategy, transistors are sized first and then the complex gates created. With this approach more realistic transistor sizes are used in deciding how to form complex gates. Those gates with small transistor sizes are used to build complex gates; those with large sizes are mostly left untouched. The new complex gate's transistor size is based upon the largest transistors of the gates that were merged. When this new complex gate is inserted, the gates that drive it may be driving larger transistors than they were before. Since these gates must drive a larger capacitive load, they are now undersized if the present speed is to be maintained.

The third and most complicated strategy involves forming complex gates and sizing transistors at the same time. Before a decision is made to create a complex gate, all gates that the new complex gate will drive must be processed. That is, no changes in transistor sizes or complex gates can be made along any paths that can be reached from the output of the new complex gate. Doing so could change the load that the complex gate is required to drive, resulting in the problems encountered in the second strategy. After a new complex gate is created, all gates that drive it are resized. This approach will presumably provide the best results.

5.4. Algorithm for Layout Driven Synthesis

We present an algorithm for producing high-performance CMOS custom layouts. Our algorithm consists of four phases as shown in Figure 17. The input is a set of boolean equations. These equations are first minimized and then factored by

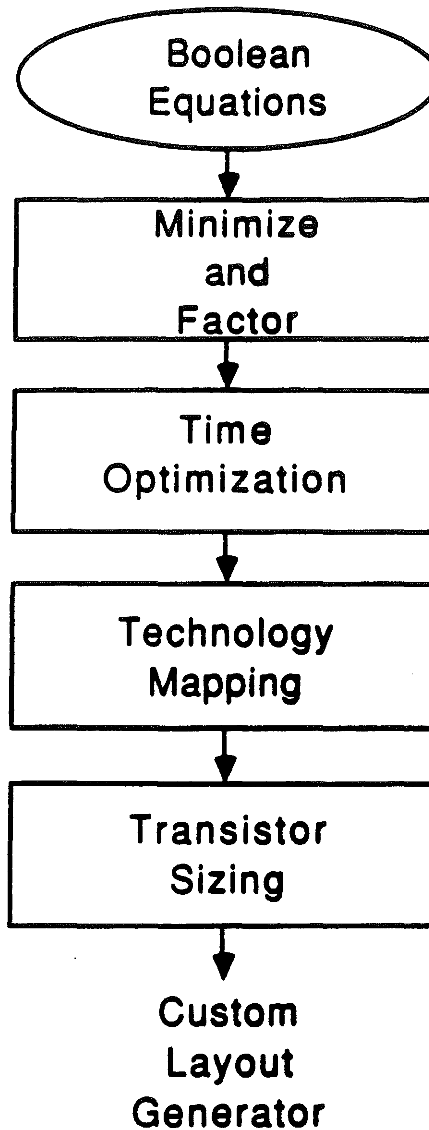


Figure 17. Algorithm for Layout Driven Synthesis

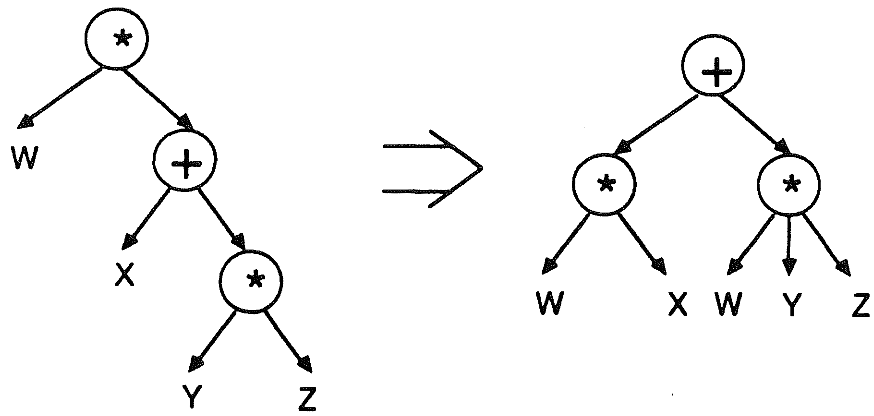
MISII to make use of common terms. Parameters for the maximum number of NFETs and the maximum number of PFETs allowed in series are provided for the next phases to place limits on the size of complex gates created. In the second

phase, the algorithm reduces the number of levels along the longest paths by performing balanced factoring. This technique attempts to partially collapse the critical path then refactor so as not to exceed the maximum number of transistors allowed per gate. The algorithm's third phase then performs technology mapping by combining gates into complex gates. The fourth phase sizes the transistors, producing a design that can be passed to a custom layout generator.

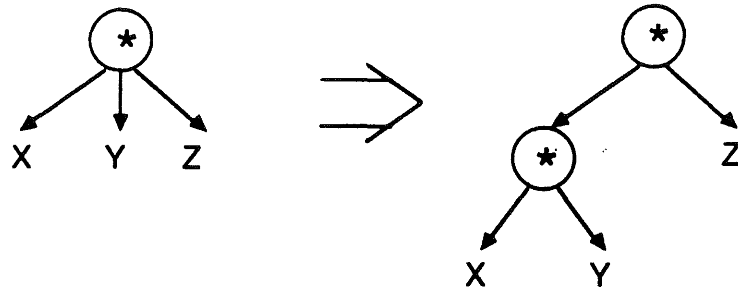
5.4.1. Time Optimization

The general idea of timing optimization is to perform a limited collapse of nodes along the critical path. This requires some duplication of logic, creating a design with greater breadth and shorter depth. Some logic along critical paths can be removed and added along non-critical paths. More than just a partial collapse is required, however, to produce high-performance designs. We employ a balanced factoring method that addresses the problem of how many transistors to place in each gate. Gates are factored to get the number of inputs per gate that achieves the best speed rather than attempting to reduce the number of transistors.

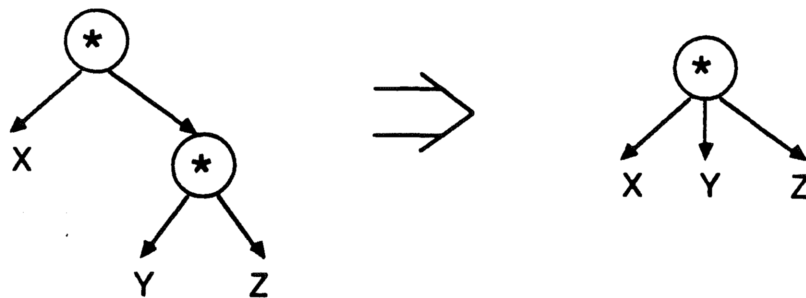
Three operations are used to reduce the number of levels: distribute, extract, and merge. These operations are shown as applied to a graph of operator nodes in Figure 18. Distribution transfers logic across other nodes, allowing some logic to be shifted to non-critical paths. Extraction removes non-critical inputs from a node, reducing the delay through the node. Merging combines two nodes having the same operator. Only two *critical* nodes should be merged in order to reduce the delay.



(a) Distribute Operation



(b) Extract Operation



(c) Merge Operation

Figure 18. Critical Path Reduction Operations

Use of the three operations is illustrated in Figure 19. Figure 19(a) is a design of five levels. Distributing node 2 over node 3 produces the design of Figure 19(b) which has one fewer level. Using the extract operation then on node 4 balances out the paths through that node (Figure 19(c)). Node 2 can be merged into node 0 (Figure 19(d)) and an extract operation applied to node 0 (Figure 19(e)) to balance the path lengths. In this manner, critical paths can be shortened by shifting logic to non-critical paths.

The input equations which have been minimized and factored are converted into a tree structure similar to that in Figure 19. Nodes may have only as many inputs as the number of transistors allowed in series. If performing the optimization at a node creates a new longest path or requires the addition of too many nodes, the optimization will not be performed. We perform the timing optimization going from inputs to outputs. Duplication of logic through distribute operations is preferred closer to input pins as there are fewer gates to be duplicated. Also transistor sizes are smaller closer to the inputs and duplicated logic can be placed in complex gates with more transistors. These factors help to contain the extra area that results.

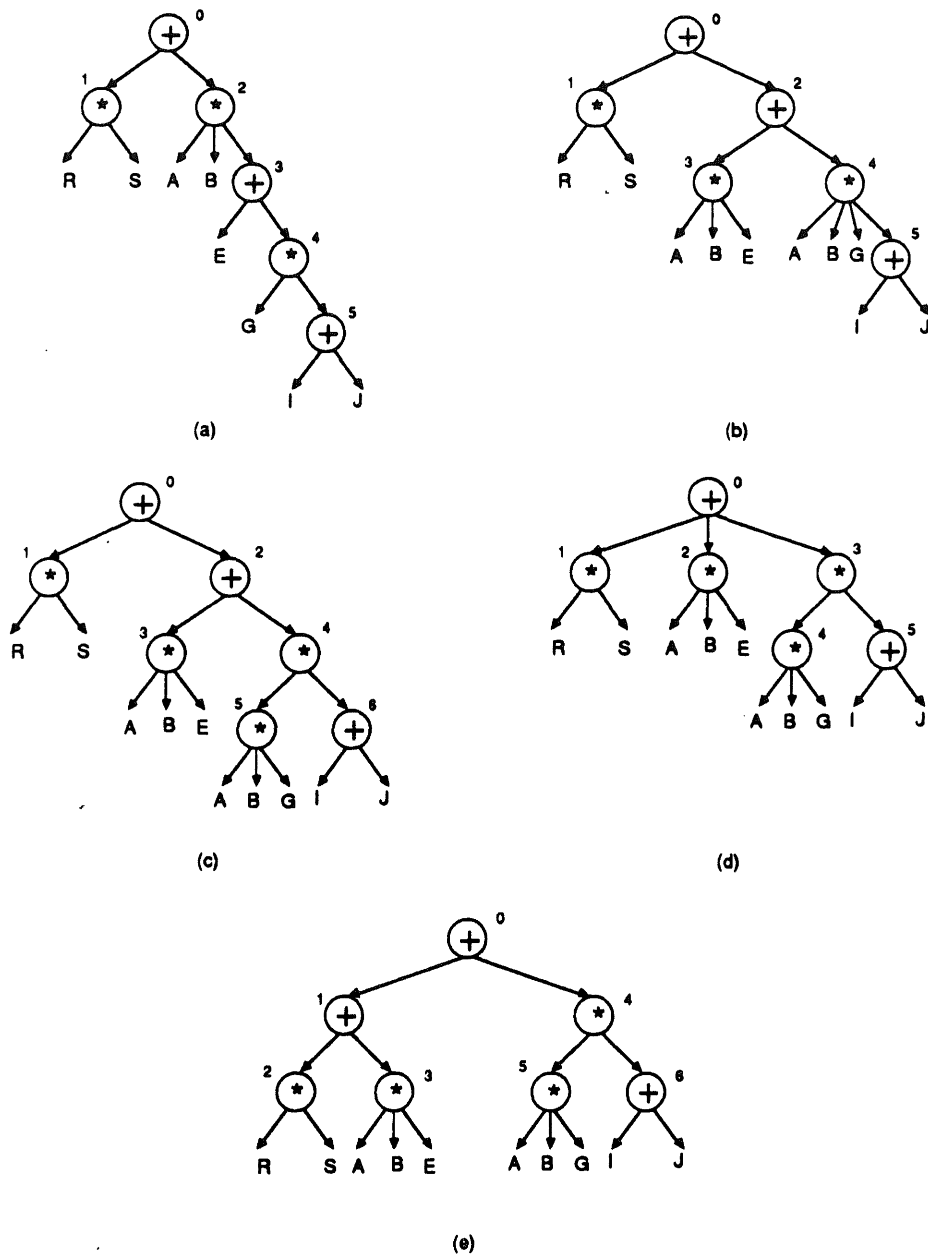


Figure 19. Example of Shortening Critical Path

Algorithm 1: Balanced Factoring

```
{restructure critical paths}
Let
  V be the set of vertices in the design;
  CP be the set of vertices along the critical path;

procedure balanced_factor(V)
begin
  CP_o = find_critical_path(V);
  can_improve = TRUE;
  while (can_improve)
    balance(V, CP_o)
    CP_n = find_critical_path(V);
    if (CP_o = CP_n) then
      can_improve = FALSE
    else
      CP_o = CP_n;
    end
  end
end


{Redistribute Logic Along the Critical Path}
Let
  D_[i,V] be the cumulative delay at node i of design V;
  D_accept be a delay reduction constraint;
  AN_[i] be the increase in number of nodes due to optimization of node i;
  A_accept be an area increase constraint;

procedure balance(V, CP)
begin
  for all i ∈ CP
    V_new = distribute(V, i, i+1);
    V_new = extract(V_new, i+1);
    V_new = merge(V_new, i+1, i-1);
    V_new = extract(V_new, i-1);
    if (D_[i,V] - D_[i-1,Vnew] > D_accept) and (AN_i < A) then
      V = V_new;
    end
  end
end
```

```
{Distribute node i over node i+1}
Let
  S_i be the set of input nodes to node i;

procedure distribute(V, i, c)
begin
  remove c from S_i
  For all j ∈ S_c
    if operator(i) = operator(j) then
      For all k ∈ S_i
        l = duplicate_tree(k);
        add l to S_j;
      end
    extract(V, j);
  else
    m = duplicate_tree(i);
    remove j from S_c;
    add m to S_c;
    add j to S_m;
    extract(V, m);
  end
 end
 add c to S_{i-1};
end
```

```
{Remove Non-Critical Inputs From Critical Nodes}
Let
  CD_i be the cumulative delay at the critical input of node i;
  S_i be the set of inputs of node i;
  T_c be the transistor limit constraint
  P be the delay percentage
```

Algorithm 1: Balanced Factoring

```
{restructure critical paths}
Let
  V be the set of vertices in the design;
  CP be the set of vertices along the critical path;

procedure balanced_factor(V)
begin
  CP_o = find_critical_path(V);
  can_improve = TRUE;
  while (can_improve)
    balance(V, CP_o)
    CP_n = find_critical_path(V);
    if (CP_o = CP_n) then
      can_improve = FALSE
    else
      CP_o = CP_n;
    end
  end
end


{Redistribute Logic Along the Critical Path}
Let
  D_[i,V] be the cumulative delay at node i of design V;
  D_accept be a delay reduction constraint;
  AN_[i] be the increase in number of nodes due to optimization of node i;
  A_accept be an area increase constraint;

procedure balance(V, CP)
begin
  for all i ∈ CP
    V_new = distribute(V, i, i+1);
    V_new = extract(V_new, i+1);
    V_new = merge(V_new, i+1, i-1);
    V_new = extract(V_new, i-1);
    if (D_[i,V] - D_[i-1,Vnew] > D_accept) and (AN_i < A) then
      V = V_new;
    end
  end
end
```

```
{Distribute node i over node i+1}
Let
  S_i be the set of input nodes to node i;

procedure distribute(V, i, c)
begin
  remove c from S_i
  For all j ∈ S_c
    if operator(i) = operator(j) then
      For all k ∈ S_i
        l = duplicate_tree(k);
        add l to S_j;
      end
    extract(V, j);
  else
    m = duplicate_tree(i);
    remove j from S_c;
    add m to S_c;
    add j to S_m;
    extract(V, m);
  end
 end
 add c to S_{i-1};
end
```

{Remove Non-Critical Inputs From Critical Nodes}
Let
  CD_i be the cumulative delay at the critical input of node i;
  S_i be the set of inputs of node i;
  T_c be the transistor limit constraint
  P be the delay percentage

```
procedure extract(V, i)
begin
  input_cnt = T_c;
  sort S_i by delay (worst delay to least delay)
  if (length(S_i) > 1) then
    k = node_duplicate(i);
  end
  if (length(S_i) > T_c) then
    For all j ∈ S_i
      if (input_cnt = T_c and length(S_i) > 1) then
        n = node_duplicate(i);
        add n to S_k;
        k = n;
        input_cnt = 1
      else
        input_cnt = input_cnt + 1;
      end
      add j to S_k;
      remove j from S_i;
    end
  end
  num_inputs = 0;
  T_i = S_i;
  k = node_duplicate(i);
  For all j ∈ S_i
    if D_j < (P * CD_i) then
      add j to S_k;
      remove j from S_i;
      num_inputs = num_inputs + 1;
    end
  end
  if (num_inputs > 1) then
    add k to S_i;
  else
    S_i = T_i;
  end
end
```

```
{Combine Critical Nodes}
procedure merge(V, N_i, N_{i-1})
begin
  if (operator(N_i) = operator(N_{i-1}))
    For all j ∈ N_i
      remove j from N_i;
      add j to N_{i-1};
    end
  end
end
```

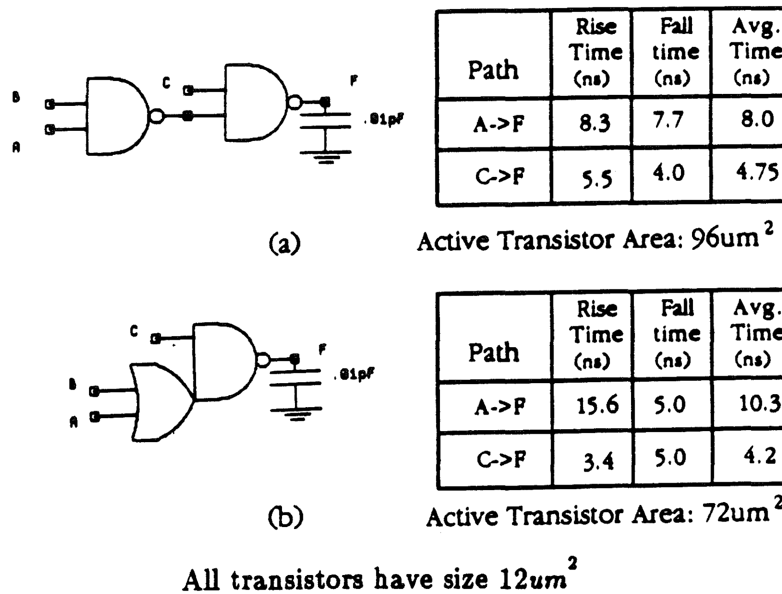
■

5.4.2. Technology Mapping

The third phase involves technology mapping. To perform the mapping, estimates of the gate delay times are made in order to identify the critical path. Each gate is converted to a NAND/NOR gate and assigned a delay based upon the number of NFETs in series, the number of PFETs in series, and the load that the gate must drive. The critical path is found using a method similar to that discussed in [YeGh88].

The next step is complex gate formation. The algorithm forms complex gates first along the critical path. The combining algorithm goes from input pins to output pins, which tends to produce large complex gates near the inputs and small complex gates at the outputs (as there are fewer gates left to combine). Complex gate formation is restricted by the user entered parameters for maximum transistors in series, the number of fanouts that the gate must drive, and the amount of slack. We have found that single gates or complex gates in which the critical path passes through only one gate level of the complex gate are best along the critical path.

Figure 20 shows an example demonstrating this with actual delay times in a 3 micron CMOS technology. The complex gate formed in Figure 20(b) has only a small effect on the delay through the critical path. However, the complex gate achieves a reduction of active transistor area (routing area is not included). Figure 20 also shows that if the critical path ran from A to F, the complex gate should not be formed. For purposes of comparison in this example, it is assumed that all inverters that must be added to some inputs of the complex gate (for the designs of Figure 20(a) and Figure 20(b) to be equivalent) are pushed back to the previous stages.



Path	Rise Time (ns)	Fall time (ns)	Avg. Time (ns)
A->F	8.3	7.7	8.0
C->F	5.5	4.0	4.75

(a) Active Transistor Area: 96um ^{2}

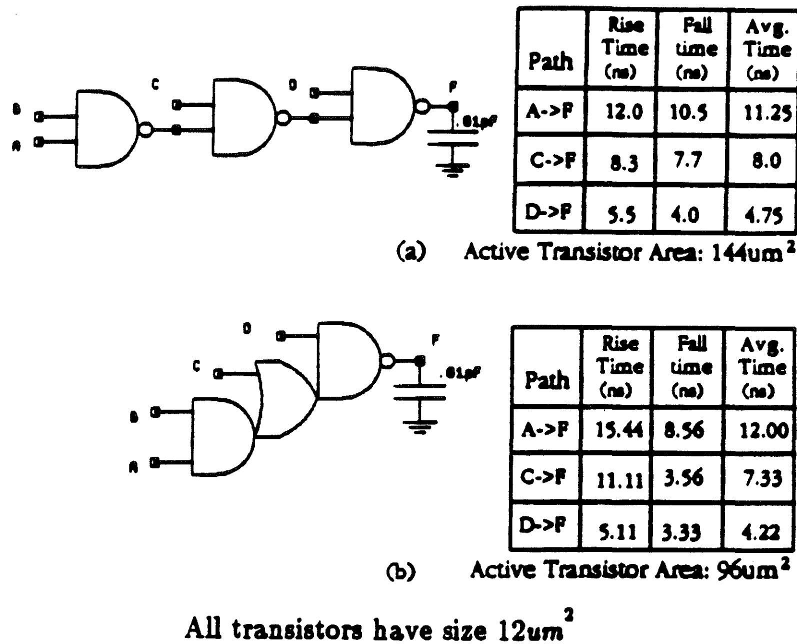
Path	Rise Time (ns)	Fall time (ns)	Avg. Time (ns)
A->F	15.6	5.0	10.3
C->F	3.4	5.0	4.2

(b) Active Transistor Area: 72um ^{2}

All transistors have size 12um^2

Figure 20. Area/Delay Considerations for 2-Level Complex Gates

Gates along the non-critical paths are combined into complex gates in the fourth phase. The worst case path to each output is processed first to prevent the initial combining from being along the short paths. Figure 21 shows the delay and area for a complex gate with more transistors than Figure 20. It demonstrates that complex gates with more transistors are slower for some paths, but yield a savings in transistor area. Note that the complex gate of Figure 21(b) could be made just as fast as its individual gate implementation. This would require larger transistor



Path	Rise Time (ns)	Fall time (ns)	Avg. Time (ns)
A->F	12.0	10.5	11.25
C->F	8.3	7.7	8.0
D->F	5.5	4.0	4.75

(a) Active Transistor Area: 144um^2

Path	Rise Time (ns)	Fall time (ns)	Avg. Time (ns)
A->F	15.44	8.56	12.00
C->F	11.11	3.56	7.33
D->F	5.11	3.33	4.22

(b) Active Transistor Area: 96um^2

All transistors have size 12um^2

Figure 21. Area/Delay Considerations for 3-Level Complex Gates

sizes, however, and hence the complex gate consumes a larger layout area in order to match the delay. Therefore, paths with much smaller delays than the critical path can have complex gates with more transistors.

Algorithm 2: Complex Gate Generation

```
{combining gates into complex gates}
Let t be sink node and s be source node in the graph;
   Ci be the input capacitance of vertex i associated with all the fanout pins;
   Si be the slack of vertex i (required delay - actual delay);
   C_high and C_low be the fanout constraints;
   S_small and S_large be the slack constraints;
   Q be a set of vertex.

PROCEDURE combining(V,Q)
BEGIN
  FOR (i = t to s in Q)
  BEGIN
   IF (mark[i]=false)
   BEGIN
    IF (Ci > C_high OR Si < S_small)
     combine gates into two level three input complex gate;
    ELSE IF (Ci > C_low OR Si < S_large)
        combine gates into two level four input complex gate;
       ELSE
         combine as many gates as possible into complex gate;
   mark[i] = true;{where i is in V}
  END;
 END;
```

■

5.4.3. Transistor Sizing

The fourth phase is transistor sizing. Transistor sizing is performed first on the critical path to ensure optimal sizes for speed. Transistors in gates along non-

critical paths can then be sized. The N and P transistor sizes are chosen to achieve equal rise and fall times for the gate. The sizing algorithm proceeds from outputs to inputs basing size upon the capacitive load that must be driven. To achieve nearly optimal sizes for speed we examine the gate to be sized and its preceding stage. The graph of a gate's transistor size plotted against total path delay is a convex curve [He87]. We examine the effect of sizing on both gates, thus considering two curves. The point where these two curves intersect is the transistor size chosen. Further details of the transistor sizing algorithm can be found in [WuVG90].

Algorithm 3: Combine-Then-Size Strategy

{This algorithm is to combine gates into complex gates, and then to size the transistors to obtain optimum speed}

```
Let V,E be the vertex and edge sets of the graph G;
   where V is the set of all gates in G;
   and E is the set of all connections between gates in G;
Let Vcritical be the vertex set of the critical path;

BEGIN
  restructure_longest_path(V,Vcritical);
  find_critical_path(V,Vcritical);

  {combine gates along critical path}
  combining(V,Vcritical);
  {combining gates along non-critical paths}
  combining(V,V);

  {size transistors along critical path}
  transistor_sizing(Vcritical,V);
  {size transistors along non-critical paths}
  transistor_sizing(V,V);
END
```

■

5.5. Results

Our synthesis tool is currently running on SUN 3 workstations under the UNIX operating system. Synthesized designs with complex gates and sized transistors are passed to LES for layout generation and then to GDT[1] [BuMa85] for simulation and comparison. We have run a number of examples and compared our results with those of MISII (OCTTOOLS release 2.0). They are shown in Figure 0. To achieve a fair comparison, the output of MISII (which does not size transistors) was run through our transistor sizing routine before passing it on to LES. The LES layout was passed on to GDT to perform the simulation. The results for our layout driven synthesis algorithm (LDS) were obtained by first minimizing and factoring the design using MISII, then applying the timing optimization, complex gate formation, and transistor sizing before passing the circuit to LES and GDT. Table 1. Table 2. and Table 3. display a number of MCNC benchmark examples with comparisons to MISII (OCTTOOLS release 3.1). Figure 2 compares custom layout results, Figure 3 was generated using the MCNC standard cell library. Because of the size of these designs, we did not obtain actual layout results but used estimates for time (which we have found to be within $\pm10\%$ of the actual value) and the active transistor area (not including the routing area). The following commands were used to perform the logic synthesis in MISII.

---

[1] GDT is a registered trademark of Silicon Compiler Systems.

Design	Complexity	Area (um²)			Time (ns)		
	# Gates	MISII	LDS	% improvement	MISII	LDS	% improvement
P147	18	72,341	62,129	14.1	11.84	11.26	4.9
P191	17	62,304	62,708	-0.8	16.55	11.33	31.5
F1	20	76,903	51,000	33.7	17.11	11.50	32.8
F2	29	128,040	81,510	36.3	21.11	12.61	40.3
z4mlc	64	330,038	270,150	18.1	26.28	12.56	52.2

Table 1. Custom Layout results for MISII and the LDS Algorithm

Design	Active Transistor Area (um^2)			Time (ns)		
	MISII	LDS	% increase	MISII	LDS	% improve-ment
9symml	16,848	16,884	0.2	45.03	41.61	7.7
z4ml	5,340	6,636	24.2	20.16	16.69	17.2
b9	7,764	14,820	91.8	25.87	20.68	20.1
f51m-hdl	8,676	12,168	40.2	33.35	26.30	21.1
f51m	10,944	14,184	29.6	29.65	25.18	15.1
att12	6,564	11,040	68.2	45.95	23.76	48.3

Table 2. Custom Layout Results Using MCNC Benchmarks

Design	Area			Time		
	MISII	LDS	% increase	MISII	LDS	% improve-ment
9symml	382	465	21.7	19.6	15.4	21.4
z4ml	119	116	-2.5	10.1	6.6	34.7
b9	237	338	42.6	8.9	8.4	5.6
f51m-hdl	216	284	31.5	10.8	10.7	0.9
f51m	269	295	9.7	11.1	11.3	-1.7
att12	218	256	17.4	14.4	11.6	19.4

Table 3. Standard Cell Layout Results Using MCNC Benchmarks

```
source script
rlib les_cus.lib
map -m.75
phase -g
speed_up -w 0
map -m.75
```

The script used is the standard script provided with OCTTOOLS release 2.0. We found that it produced superior results for timing than the standard script in OCTTOOLS release 3.1. The technology file les_cus.lib contains delay and area estimations for gates in the SCMOS 3 micron technology. Our algorithm also uses this

Design	Active Transistor Area (um^2)			Time (ns)		
	MISII	LDS	% increase	MISII	LDS	% improvement
9symml	16,848	16,884	0.2	45.03	41.61	7.7
z4ml	5,340	6,636	24.2	20.16	16.69	17.2
b9	7,764	14,820	91.8	25.87	20.68	20.1
f51m-hdl	8,676	12,168	40.2	33.35	26.30	21.1
f51m	10,944	14,184	29.6	29.65	25.18	15.1
att12	6,564	11,040	68.2	45.95	23.76	48.3

Table 2. Custom Layout Results Using MCNC Benchmarks

Design	Area			Time		
	MISII	LDS	% increase	MISII	LDS	% improve-ment
9symml	382	465	21.7	19.6	15.4	21.4
z4ml	119	116	-2.5	10.1	6.6	34.7
b9	237	338	42.6	8.9	8.4	5.6
f51m-hdl	216	284	31.5	10.8	10.7	0.9
f51m	269	295	9.7	11.1	11.3	-1.7
att12	218	256	17.4	14.4	11.6	19.4

Table 3. Standard Cell Layout Results Using MCNC Benchmarks

```
source script
rlib les_cus.lib
map -m.75
phase -g
speed_up -w 0
map -m.75
```

The script used is the standard script provided with OCTTOOLS release 2.0. We found that it produced superior results for timing than the standard script in OCTTOOLS release 3.1. The technology file les_cus.lib contains delay and area estimations for gates in the SCMOS 3 micron technology. Our algorithm also uses this

table to perform delay estimations. Generally synthesis with layout constraints produced designs up to 48 percent faster with an average of 30 percent more area. The area for our designs could be reduced by performing area optimizations along the non-critical paths. Currently no area optimizations are performed. Tables 2 and 3 demonstrate that our algorithm improves performance for both standard cell and custom layout designs.

Figure 22 displays a composite graph showing our ability to perform area/time tradeoffs. Our results are normalized against standard cells whose reference point is shown at time = 1, area = 1. Several points are noted on the graph. Point A

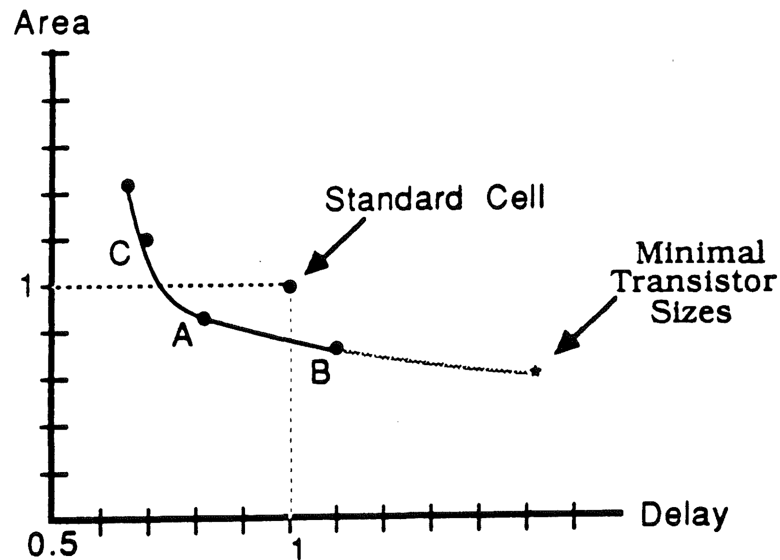


Figure 22. Composite Graph of Experimental Results

shows that synthesis considering layout produces designs with smaller area and faster speed. Point B illustrates the capability to save even more area while having a larger delay than the standard cell synthesis approach. Similarly, Point C shows that speed can be improved further at the expense of area. The dashed section of the curve ends when the minimal transistor sizes are used (PFET size = 2, NFET size = 1). This part of the curve shows expected results as we have not actually tested this. Another observation is that using custom layout, the speed can be kept speed constant while varying the output load (fanout). Of course, there is an associated increase in the area. As the fanout increases in standard cells, however, the speed is decreased.

table to perform delay estimations. Generally synthesis with layout constraints produced designs up to 48 percent faster with an average of 30 percent more area. The area for our designs could be reduced by performing area optimizations along the non-critical paths. Currently no area optimizations are performed. Tables 2 and 3 demonstrate that our algorithm improves performance for both standard cell and custom layout designs.

Figure 22 displays a composite graph showing our ability to perform area/time tradeoffs. Our results are normalized against standard cells whose reference point is shown at time = 1, area = 1. Several points are noted on the graph. Point A

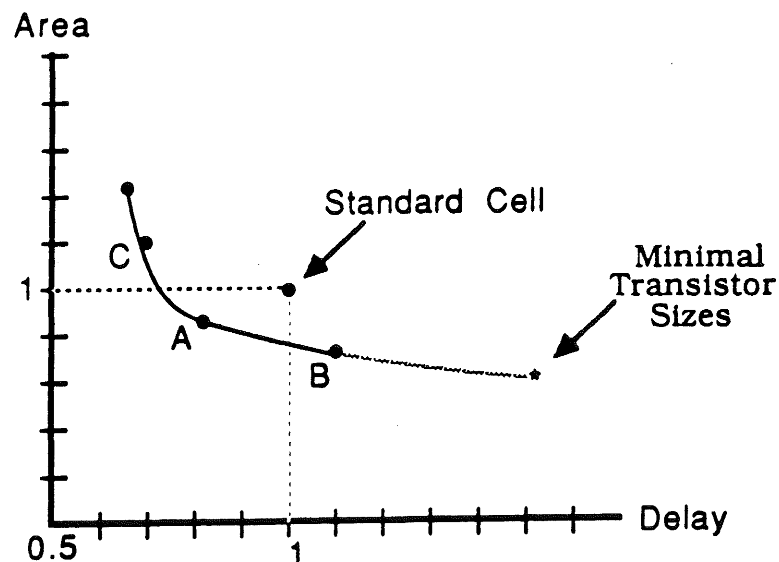


Figure 22. Composite Graph of Experimental Results

shows that synthesis considering layout produces designs with smaller area and faster speed. Point B illustrates the capability to save even more area while having a larger delay than the standard cell synthesis approach. Similarly, Point C shows that speed can be improved further at the expense of area. The dashed section of the curve ends when the minimal transistor sizes are used (PFET size = 2, NFET size = 1). This part of the curve shows expected results as we have not actually tested this. Another observation is that using custom layout, the speed can be kept speed constant while varying the output load (fanout). Of course, there is an associated increase in the area. As the fanout increases in standard cells, however, the speed is decreased.

CHAPTER 6.

MICROARCHITECTURE OPTIMIZATION

6.1. Introduction

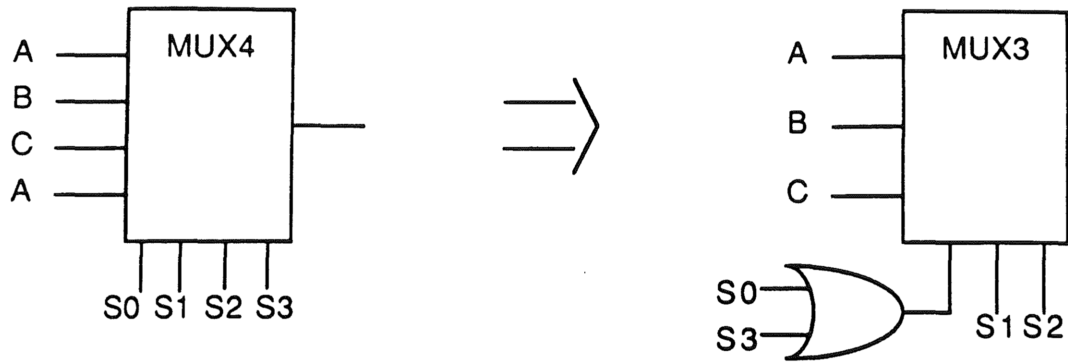
This chapter examines methods for performing microarchitecture optimization, then presents algorithms for incorporating these methods.

6.2. Types of Microarchitecture Optimization

The goal of microarchitecture optimization is to optimize the design for area/time without changing the state assignment. This section describes the types of optimizations that can be performed.

6.2.1. Minimization

This type of optimization should be one of the first to be applied. It reduces the number of components or the amount of logic in a component. Figure 23(a) and Figure 23(b) show examples of minimization rules. Figure 23(a) shows the removal of the redundant signal A as an input to the multiplexor. Figure 23(b) shows the replacement of an adder by the sum of its two constant values.



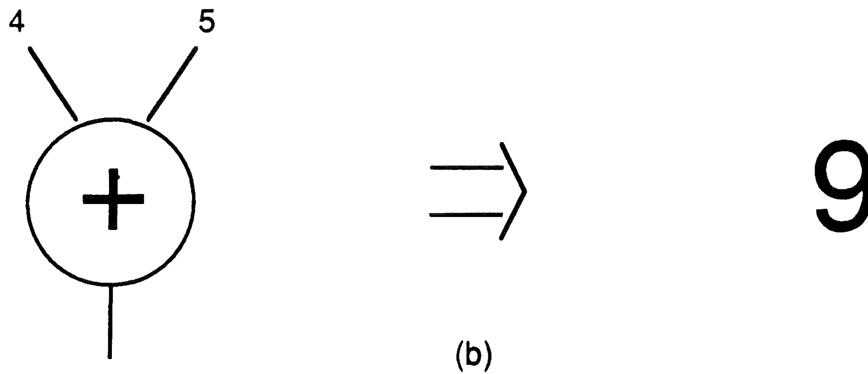


Figure 23. Minimization Rules

6.2.2. Factorization

Factorization is used to extract early arriving signals in order to speed up late arriving ones. It may also be necessary to factor components in order to meet the requirements of a layout module generator. For example, module generators may

only be able to construct 4 to 1 or 2 to 1 multiplexors. Figure 24 illustrates the factorization of a multiplexor.

Procedure 4.1 describes the factoring algorithm. The algorithm factors a single component having $|\mathbf{R}|$ inputs, $\mathbf{R}$ being the set of all required input to output delays. The procedure *Factor* is recursive and takes five parameters: 1) c, which indicates which input of the parent component the factored out inputs should be connected to, 2) the set $\mathbf{R}$, 3) n, the maximum number of inputs to be factored out of the parent component, 4) s, the size of the last component that failed to meet the constraints, and 5) C_0, the component to be factored.

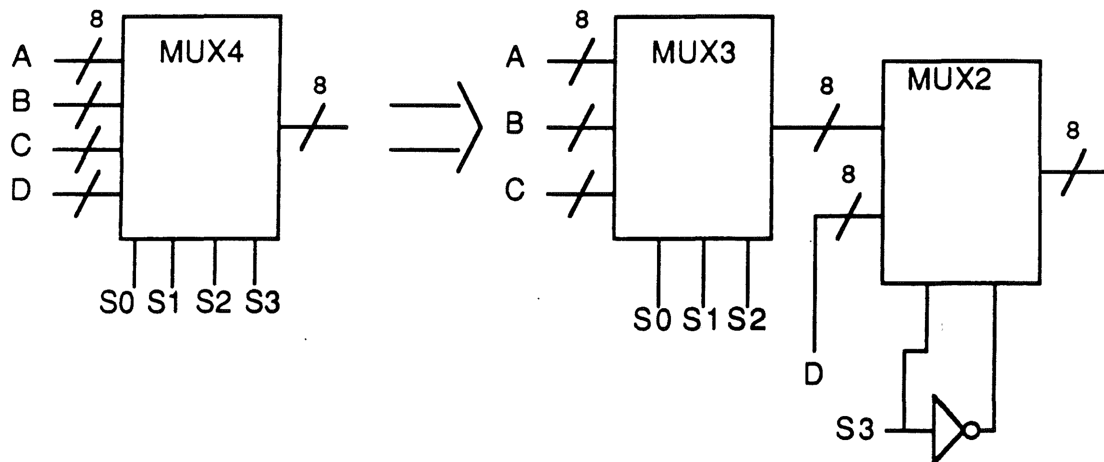


Figure 24. Factorization of Microarchitecture Components

```
Let:R={r | required delay from input to output};
     each component C_i has delay d_i and s_i inputs;
    C_0 be the component to be factored
    n=number of component inputs that still need to be assigned;
    s=last tried component size that failed to meet the constraints;
    D_s=smallest delay through any multiplexor that can be generated by the database

Function Factor(c,R,n,s,C_0)
   Begin
   start:
      n_t = n;
      R_1 = R;
      C_1 = find_new_component(R, n, s);
      if (C_1 ≠ φ )
         if (c > 0)
            assign C_1 to the cth input of C_0
         for(i=1;i≤s_1;i++)
            if(min(r-d_1)> D_s && ((n-s_1+i)> 1))
               n = n - Factor(i,R={r|r=r-d_1},n-s_1+i,s_1, C_1);
            else
               r_s = smallest r in R;
               assign r_s to i-th input of C_1;
               R = R - {r_s};
               n = n - 1;
            if(n==0) return(n_t);
         if(|R|==n)
            /*  Not able to assign all inputs, try again  */
            n = n_t;
            s = s_1;
            R = R_1;
            goto start;
   return(n_t-n);
   End
Function find_new_component(R,n,s)
   Begin
   largest_allowable_delay = min(r);
   max_number_of_inputs = min(s-1, n);
   if (there exist database components C_i such that
        s_i ≤ max_number_of_inputs && d_i ≤ largest_allowable_delay)
      select component C_1 such that s_1 ≥ all s_i
   else
      C_1 = φ ;
   return(C_1);
   End
```

Procedure 4.1

The factoring algorithm begins by sorting the set of required delays, **R**, from smallest delays to largest delays. Then the database is queried to find the same type of component but with fewer inputs. For example, consider Figure 25. In Figure 25, the database is shown to have returned three components having six or fewer inputs. The 2-input multiplexor has a delay of 2ns, the 4-input multiplexor has a delay of 5ns, and the 6-input multiplexor has a delay of 7ns. Figure 25 shows the factoring process for a six to one multiplexor. The set of required delays, **R**, is shown to be (5, 5, 6, 6, 7, 9) for inputs A through F, respectively. Since the six input multiplexor did not meet the required delays, the next smallest one is selected. In this case it is the four input multiplexor.

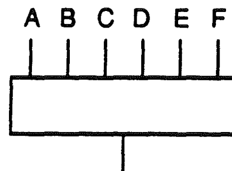
The next stage is to assign the inputs to this new component. All inputs whose required delays will not be satisfied if they are factored out (ie., the delay through the new component + the smallest possible delay through any component of the same type) are connected directly to the new component. The remaining signals represent those that can be factored out. The algorithm queries the database again to find the component with the most inputs that will still meet the timing constraints when the signals are extracted. This component will then be processed recursively in a similar manner. When a solution is found that meets the time constraints, the algorithm ends.

For the example of Figure 25, input A cannot be factored out of the 4 to 1 multiplexor or its timing constraint of 5 will not be met. That is, the delay of the four

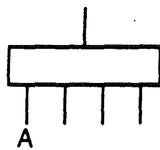
List of multiplexors returned by the component database:

(size, delay) = (2,2) (4,5) (6,7)

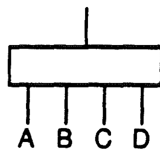
A B C D E F
Set of required delays R = (5,5,6,6,7,9)



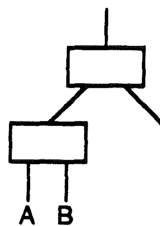
(a) Initial Implementation



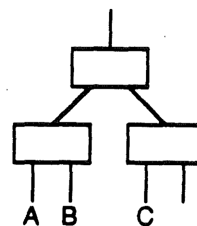
(b)



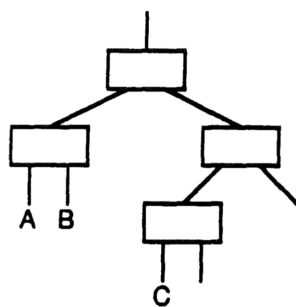
(c)



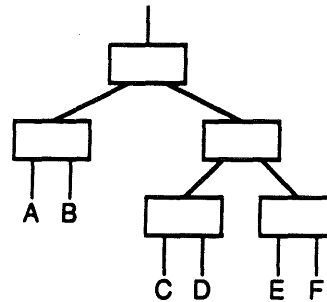
(d)



(e)



(f)



(g)

Figure 25. Example of Factorization

input multiplexor (with delay of 5) plus the delay of the smallest multiplexor (2-input MUX with delay of 2) is greater than the required delay of 5 for input A. For this reason, input A is connected directly to the 4-input multiplexor (Figure 25(b)). The set **R** then becomes (5, 6, 6, 7, 9) with only five more inputs to be assigned. For similar reasons, inputs B, C, and D are assigned directly to the 4-bit multiplexor as shown in Figure 25(c). At this point, not all inputs have been assigned and there are no unused multiplexor input ports. Therefore, using the 4 to 1 multiplexor has failed. The set **R** is reset to the original (5, 5, 6, 6, 7, 9) and another attempt is made using a smaller multiplexor. If a 2 to 1 multiplexor is used, inputs A and B can be factored out using a second 2 to 1 multiplexor (Figure 25(d)). The time constraints are still met and the new set **R** is (6, 6, 7, 9). The largest multiplexor that can be used to factor out input C is a 2 to 1 multiplexor (Figure 25(e)). In addition, input C can be factored out again using another 2 to 1 multiplexor and the time constraints are still met (Figure 25(f)). In a similar fashion, inputs D, E, and F can be assigned as in Figure 25(f).

6.2.3. Swap Equivalent Signals on the Same Component

If two signals on a component are interchangeable and one has less delay than another, the early arriving signal can be swapped with the late arriving signal. Figure 26(d) demonstrates how this can be accomplished. Swapping of component pins can be described as follows. Let $\mathbf{I}(c) = \{i_j \mid j = 1..n\}$ be a set of equivalent inputs to a component c, where $i_j = \{a_j, r_j, s_j\}$. a_j is the arrival time, r_j is the required time,

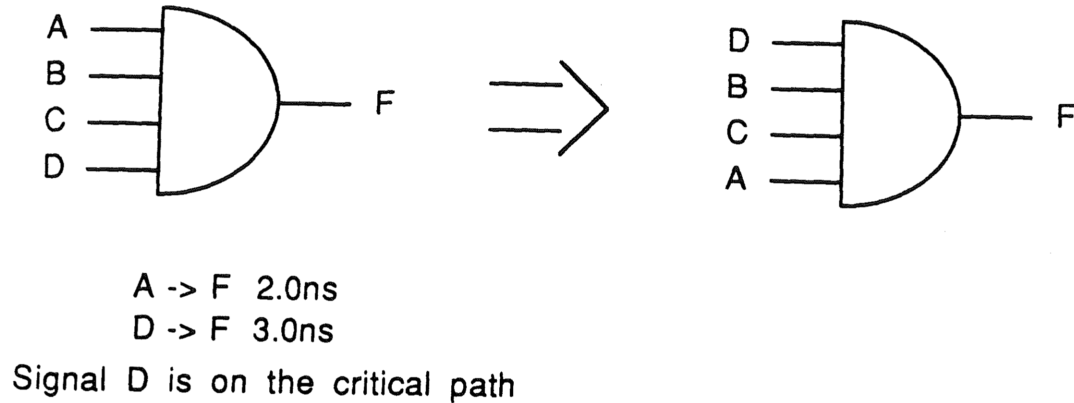


Figure 26. Signal Swapping.

and $s_j = r_j - a_j$ is the slack.

Let $\mathbf{T} = \{i_j \mid s_j < 0 \ j = 1..n\}$ be a set of critical inputs, $\mathrm{N} = \{i_j \mid s_j \geq 0 \ j = 1..n\}$ be a set of non-critical inputs. Sets T and N can then be sorted according to each pin's slack. Swapping of pins then takes place as shown in Procedure 4.2. The algorithm tests whether a pin from **T** can be swapped with a pin from N. If doing so does not create a new critical path, the pins are swapped.

6.2.4. Merge Similar Units

Two components can be merged when one of them performs a subfunction of the other. For example, in Figure 27, combining a register and shifter into a

```
Let ABS() be the absolute value function

Procedure Swap_Pins (T, N)
  Begin
  k = 0
  For j=0 to |T|
     Begin
     i_j = the jth pin of T;
     s_t = the slack of pin i_j;
     i_k = the kth pin of N;
     s_n = the slack of pin i_k;
     If ABS(s_t) ≤ ABS(s_n) then
        Begin
        swap(i_j, i_k);
        k = k + 1;
        End If
     End
  End
```

Procedure 4.2

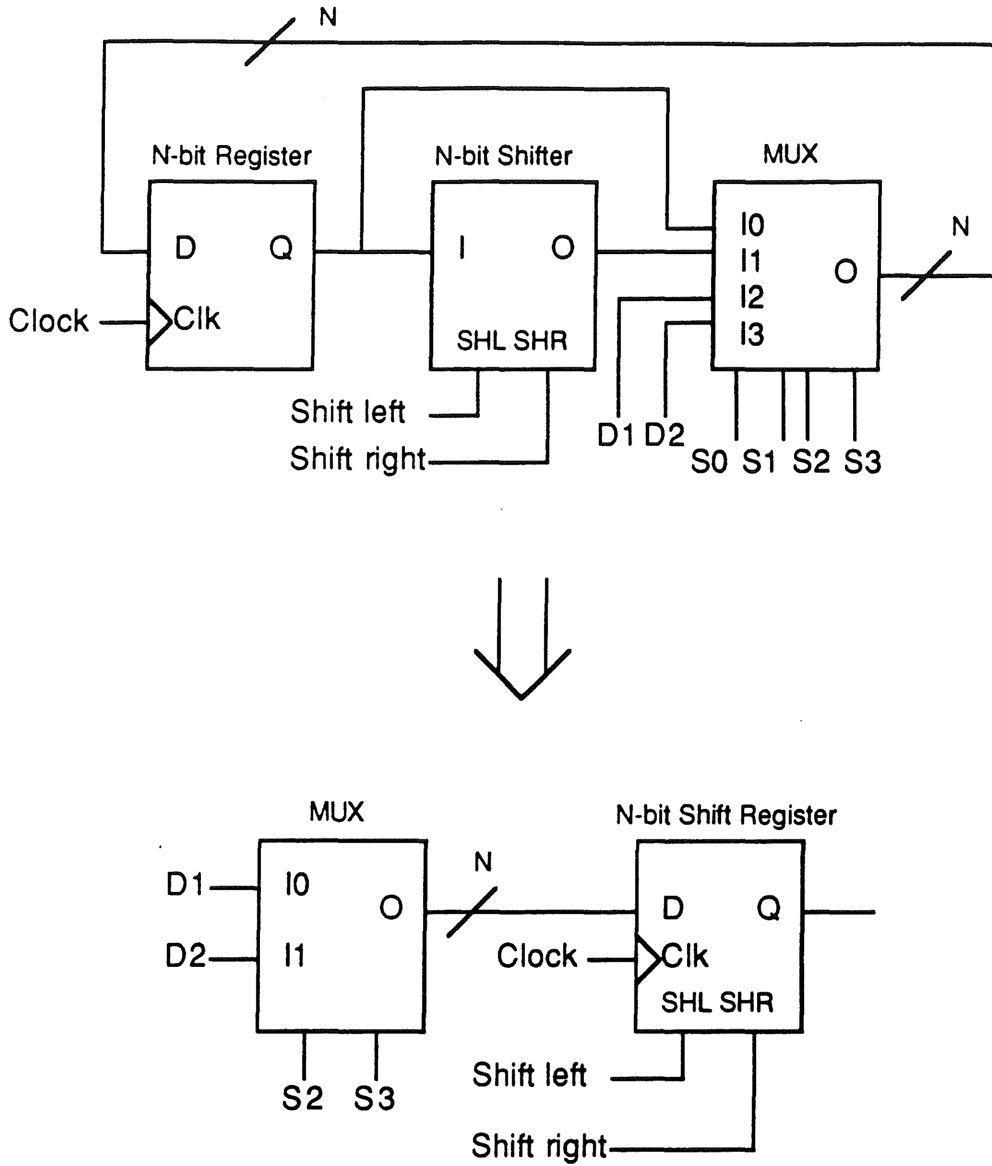


Figure 27. Merge Similar Units

register that performs a shift. Merging rules examine connectivity between two components and their functionality. Functionality of components can be found by querying the database for a list of functions that the microarchitecture component performs. The merge can be performed for two components, c_0 and c_1, when function(c_0) $\subset$ function(c_1). For example, in Figure 27, the function *shift* is a function that can also be performed as part of the register component. Thus a register that does not perform a shift and a shifter can be combined into a single shift register.

Merging of similar components is accomplished in two subphases: 1) same type component merging, and 2) different type component merging. Same type component merging is accomplished by an algorithm that proceeds from the design's input pins to the design's output pins, examining whether two components that are of the same type are connected together (eg., two multiplexors, two adders, etc). The algorithm checks a list of valid component types for merging. If a match is found, the merging procedure continues, otherwise the next set of components is examined. Then, there are three cases that occur when merging components:

(1) A component is only connected to a component of similar function to itself as in Figure 28(a).

(2) A component is connected to multiple components, some of which are of a similar function, some of which are of a different function. An example of this is in Figure 28(b).

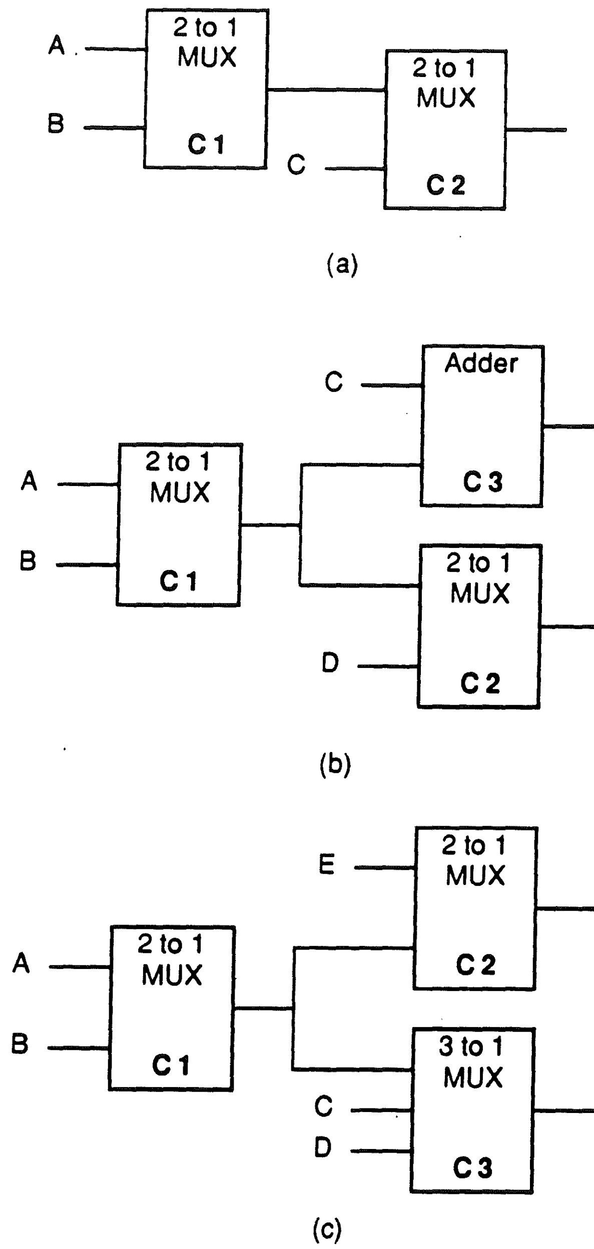


Figure 28. Three Possible Merging Cases

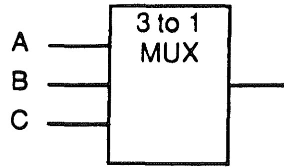
(3) A component is connected to multiple components. all of which are of the same function type.

For case 1 occurrences, the two components are merged. Thus the design of Figure 28(a) becomes the design of Figure 29(a). In a case 2 occurrence, the two components c_1 and c_2 are merged to create a new component, but c_1 must remain connected to those components which are of different types. Thus the design of Figure 28(b) becomes that of Figure 29(b). In case 3 occurrences, component c_1 is merged with all components that its output is connected to. For example, the design of Figure 28(c) becomes the design of Figure 29(c). Though the design of Figure 29(c) is more expensive than that of Figure 28(c) it is used as an intermediate step in optimization. This is discussed in further detail later.

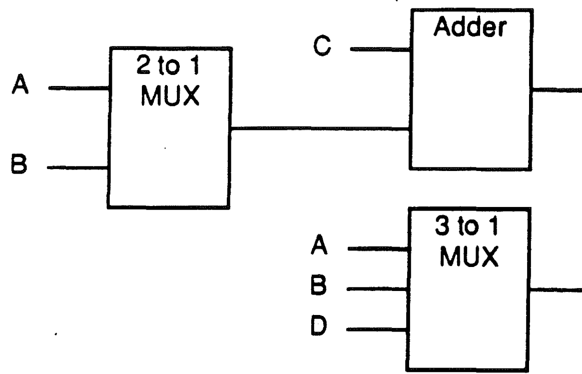
Merging of different type components is performed using rules. There is one rule for each type of merge operation. For example, a rule to perform the optimization of Figure 27 is shown in Figure 30. If the connectivity of the components is found to be similar to that of Figure 27, then the component database is queried to produce the new set of components which are substituted into the design.

6.2.5. Merge Unsimilar Units

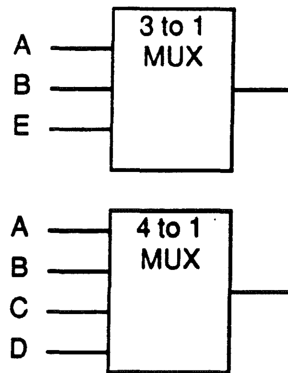
Two components can be merged into a single component that performs a different function than any of the original units. For example, combining a register and incrementer into a counter, as in Figure 31. Rules for this type of merging are



(a)



(b)



(c)

Figure 29. Results of Merging

If there is a component C1 with functionality = register
AND there is a component C2 with functionality = shifter
AND output Q of C1 is connected to input I of component C2
AND there is a component C3 with functionality = multiplexor
AND output Q of C1 is connected to input I of C3
AND output O of C3 is connected to input D of C1

Then
C4 = Query Component Database for a shift register
C5 = Query Component Database for a multiplexor with two fewer inputs than C3
Replace C1, C2, and C3 with C4 and C5

Figure 30. Rule for Merging

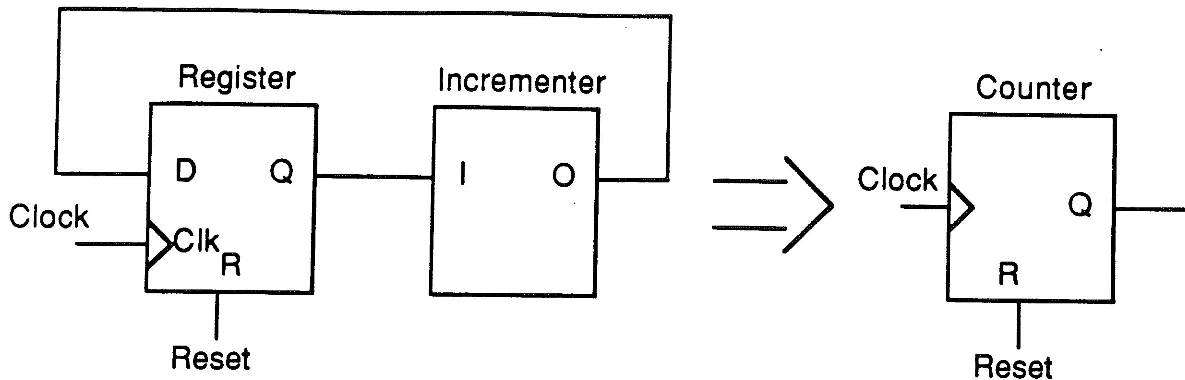


Figure 31. Merging Unsimilar Units

similar to those for merging similar functional units. In this case, however, for two components, c_0 and c_1, function(c_0) $\cup$ function(c_1) $\subseteq$ function(C_l), where C_l is a component that can be generated by the component database. For example, in Figure

31 the register and incrementer are both subfunctions of a counter component that can be generated by the database. Mergeability can be determined by querying the database with a list of functions desired in a component to determine if such a component can be created.

6.2.6. Style Change

The optimizer can query the component database to request a component that performs the same function(s) but is faster or has a smaller area. The database returns a list of components from which the optimizer can select one based on the time and area requirements. Part of the database query can include a layout style request. For components having a bit-sliceable architecture, such as ALUs, the optimizer will request a bit-sliced layout style. By placing the component in the bit-sliced datapath, routing area can be reduced. As mentioned earlier, bit-slices usually tend to be faster and smaller than their equivalent random-logic implementation. Transistor sizes in the designs produced by layout module generators are fixed, however. In some cases larger transistor sizes may be required for drive capability and speed. Buffers can usually be inserted to add greater drive capability. For greater speed, however, larger transistor sizes for gates in a design typically decrease the delay. Thus producing a random-logic design with larger transistor sizes than those used in the bit-sliced cell may result in a faster design. As standard cells have fixed transistor sizes, a transistor sizing program, such as [WuVG90], can be combined with a custom-layout generator to produce the layout

for the component. Estimators that calculate delays for bit-slice logic, based on a single slice, and for random-logic, based on gate type and transistor sizes, assist the microarchitecture optimizer in determining which design will be faster.

The database searches through its list of different architectural styles for the component to select one that it estimates will come closest to meeting the specified constraints [ChGa90]. For each style, the database maintains a range of delays and area that can be obtained. Then, depending on the layout style, the database can call tools such as logic optimizers, transistor sizing tools, etc., to generate the low level design in terms of gates or a layout. In this manner, the microarchitecture optimizer is freed from the low level details and is not concerned with which low level optimization tools should be called.

6.2.7. Duplicate Logic

Duplication of components is a technique designed to improve the speed of a path at the cost of additional area. It is the reverse of factorization. Figure 32 shows the duplication of the two-input multiplexor in order to reduce the delay along a critical path.

6.2.8. Merge Multiple Components and Optimize

This technique combines components performing different functions into a single unit and then applies logic optimization. Optimization of this type can be par-

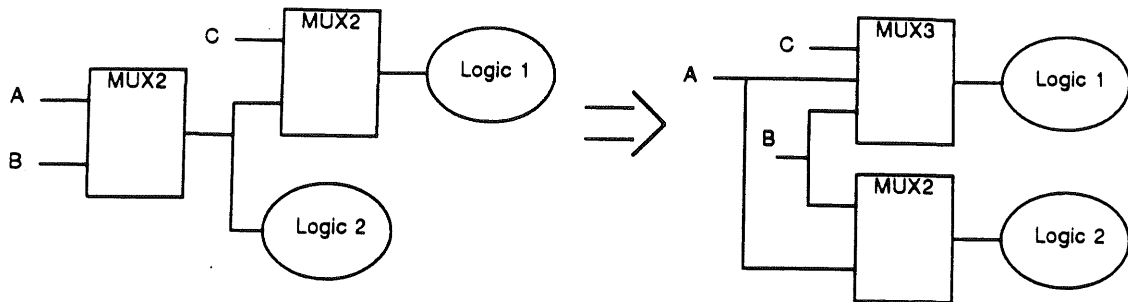


Figure 32. Component Duplication

ticularly effective when some of the inputs to the components are constants. The optimization of the constants will propagate through the logic. Thus in cases where the microarchitecture optimizer believes constant propagation in the logic will occur, it will merge even bit-sliceable components, optimize them, and treat them as random logic. Constant propagation is obvious when a number of the component's inputs are constants. Components connected to the output of such a component should also be combined into the random logic since the constants can usually be propagated through several levels of microarchitecture components.

6.2.9. Extraction of Common Subexpressions

Designs can often have the same logic duplicated in different parts of the design. Local transformations will not detect this. Therefore, global analysis is required to find and extract such common subexpressions. An example of common subexpression extraction is shown in Figure 33.

Common subexpression elimination is performed for each component type. For example, it will be performed separately for multiplexors and adders. The algorithm consists of three steps: 1) for each component which is of the selected component type, a set N is generated that contains all the inputs to that component, 2) a set L of possible subexpressions is generated, 3) a common subexpression is selected and

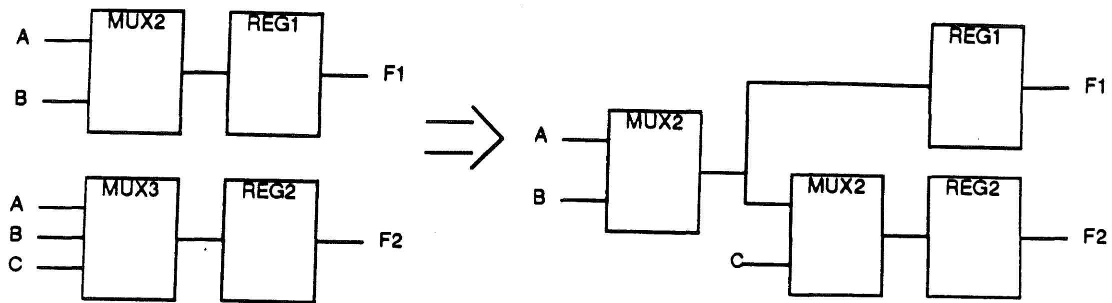


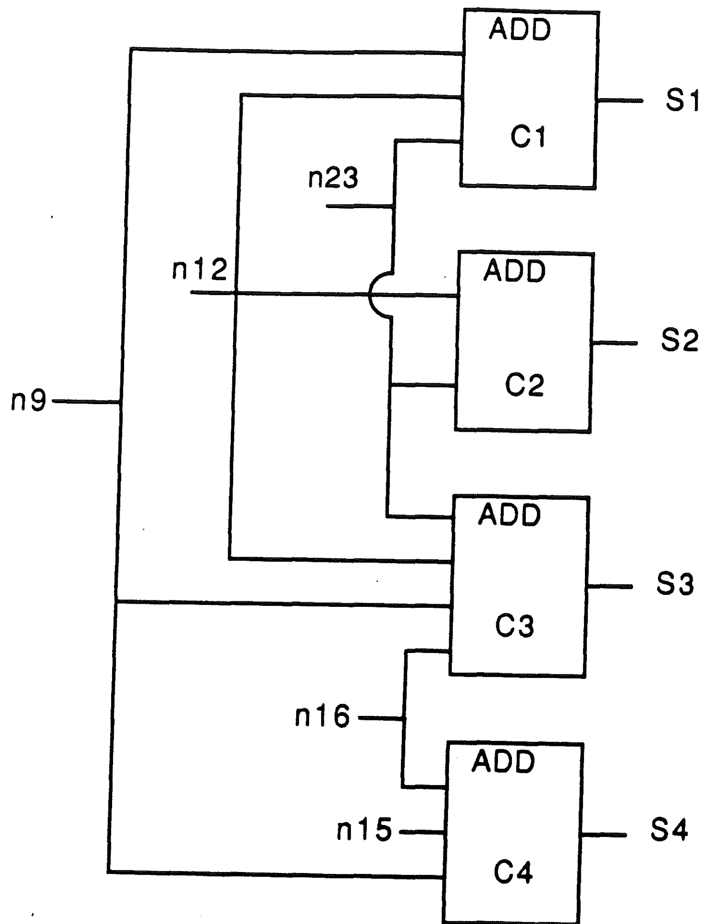
Figure 33. Common Subexpression Extraction

extracted. This process is repeated until no more subexpressions are present.

As an example, consider Figure 34. In this example, the component type is *adder*. For each adder (components c_1, c_2, c_3, and c_4), the set N_i is generated. Every net is assigned a unique id number (for example, nets n12 and n23 in Figure 34). Two components that have an input connected to the same net have that net id number in common. Each set **N** is sorted by the net id numbers. From Figure 34, the **N** sets generated are: N_1 = {n9, n12, n23}, N_2 = {n12, n23}, N_3 = {n9, n12, n16, n23}, and N_4 = {n9, n15, n16}.

The second step is to identify possible common subexpressions. A common subexpression s_e is present if the following expression is true: $|N_i \cap N_j| > 1$. A set of common subexpressions, S_{ij}, is generated for each $N_i \cap N_j$. Each set S_{ij} must have at least two elements or be the null set as only two or more inputs can be extracted. From the four **N** sets generated above, the sets S_{ij} are as follows: S_{12} = {n12, n23}, S_{13} = {n9, n12, n23}, $S_{14} = \phi$, S_{23} = {n12, n23}, $S_{24} = \phi$, and S_{34} = {n9, n16}. A set **L** is created from the **S** sets. It contains no duplicated entries but instead keeps a count of the number of occurances for each subexpression. Thus the set **L** is {{n12,n23}:2, {n9,n12,n23}:1, {n9,n16}:1}.

From **L** a subexpression for extraction is chosen using the following criteria: a) most number of occurances, and b) smallest subexpression. In the case of Figure 34, the set in **L** with the largest number of occurances is {n12,n23}. Figure 35(a) shows the new design after the common subexpression is extracted. The set {n12,n23} is



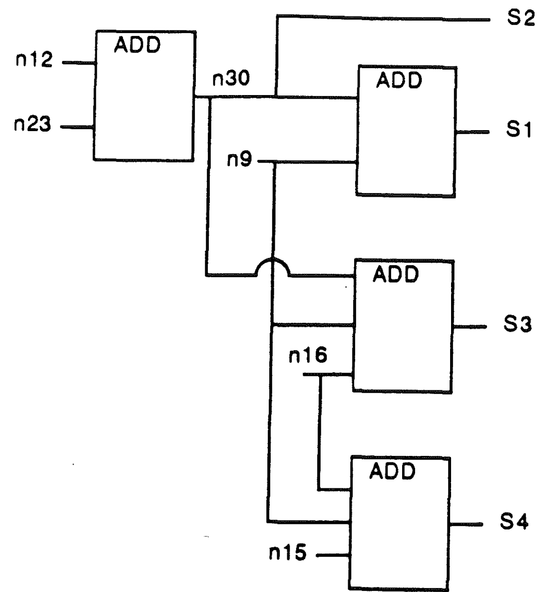
N1 = {n9, n12, n23}

N2 = {n12, n23}

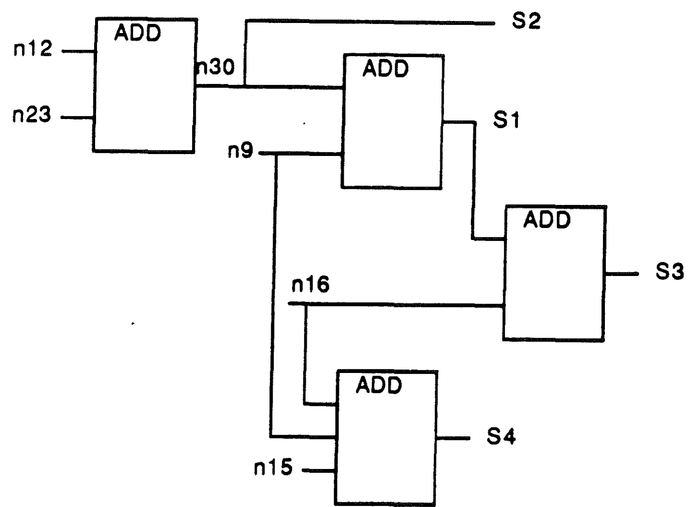
N3 = {n9, n12, n16, n23}

N4 = {n9, n15, n16}

Figure 34. Common Subexpression Elimination



(a)



(b)

Figure 35. Common Subexpression Elimination Example

removed from **L** and any set containing {n12,n23} as a subset (for example the set {n9, n12, n23}) is replaced with the net id for the extracted subexpression (in this case, n30 as shown in Figure 35). The new set **L** is {{n9, n30}:1, {n9,n16}:1}. Since both sets have the same number of occurances and the same size, the first set {n9,n30} is selected. Figure 35(b) shows the design after the extraction of this set. The new set **L** for the design of Figure 35(b) is {ϕ }. Therefore there are no more subexpressions to extract.

6.2.10. Addition of Buffers

Some components that drive large loads may require the addition of buffers at their outputs. This can reduce the delay by providing greater drive power. Methods of doing this have been discussed in [GuPa90] [SiSV90]. One solution is to partition the load by constructing a fanout tree from buffers. This tree should be constructed in a manner that does not violate the time constraints yet minimizes the amount of area increase.

[GuPa90] also mentions that in standard cell designs, components with higher drive capacity can be selected. Alternatively, some duplication of logic can be used to reduce fanout. In our case, the component database contains tools for transistor sizing and can generate a layout using a custom layout generator. This allows the design to be more finely tuned than when using standard cells. With the custom layout capability component transistor sizes are not fixed at discrete intervals. Rather, transistor sizes can be selected on a continuous basis in order to meet delay

requirements.

6.3. Strategies for Microarchitecture Optimization

Having examined types of optimization techniques, we now describe an algorithm for applying them. A block diagram of the optimization process is shown in Figure 36. It is divided into three parts: a blackboard, a control section, and a set of optimization procedures. The blackboard contains the design netlist, statistics for delay and area, a set of user constraints, a set of critical paths, and a set of non-critical paths. Critical and non-critical path sets are determined by the timing analyzer in the control section. The controller also selects which optimization procedure to use. Each optimization procedure corresponds to one phase of the optimization process. The microarchitecture optimization is carried out in four phases: general design improvement, random logic grouping, timing optimization and area optimization.

Phase 1 of the algorithm, general design improvement, attempts to reduce the number of components and to prepare the design for timing optimization should that be necessary. For example, to be able to refactor multiplexors along the critical path, all multiplexors that can be merged should be merged into a single multiplexor. Then the timing optimizer can decide how to refactor the single multiplexor. Thus optimizations in this phase set up techniques that will be performed later or employ techniques that improve both the time and area of a design.

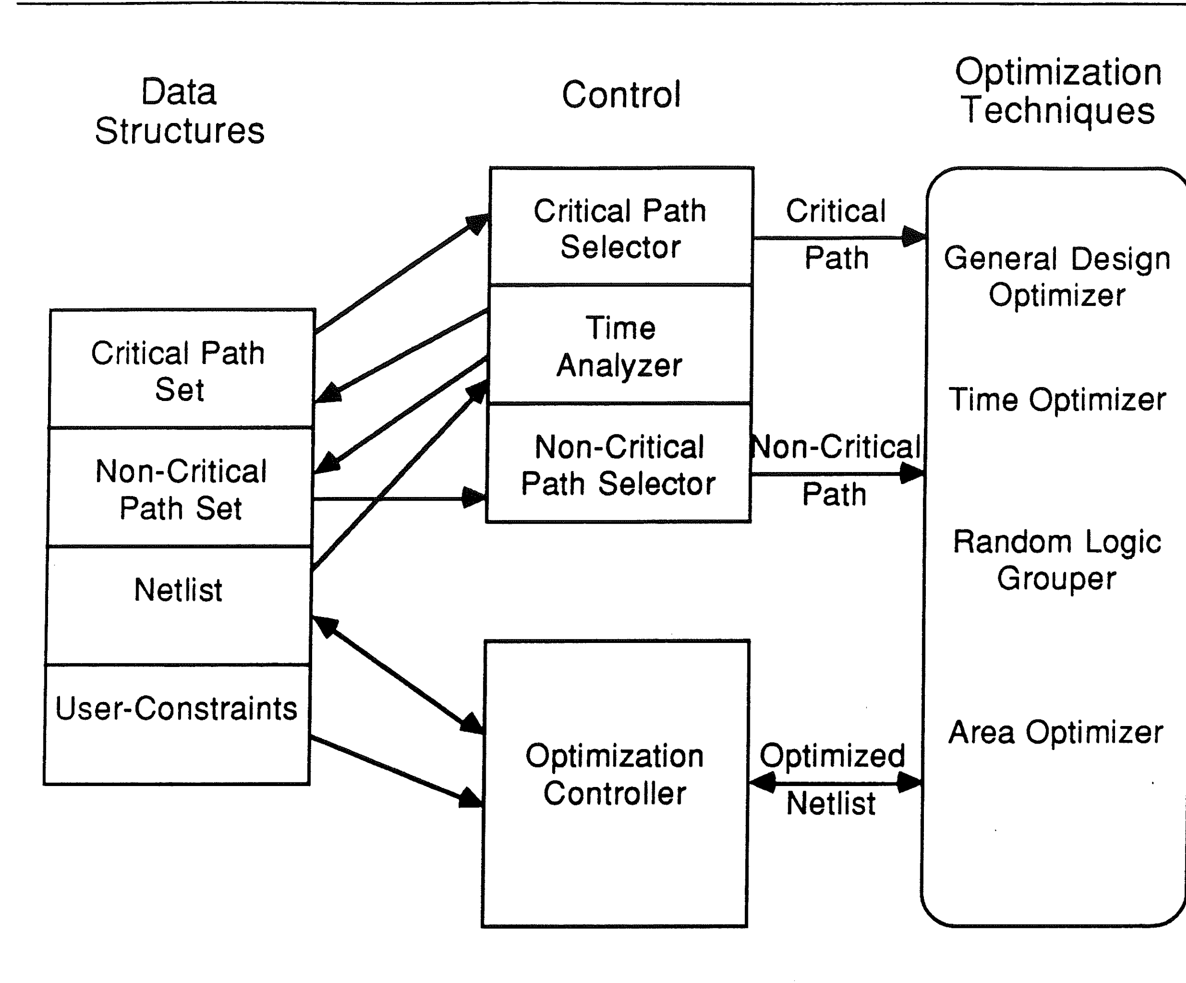


Figure 36. Overview of Microarchitecture Optimization

Phase 2 groups random logic components for logic optimization. Microarchitecture optimizations are not performed on random logic gates. Instead, they are passed to the database which has tools for restructuring the logic to meet a set of constraints passed by the microarchitecture optimizer. Thus Phase 2 prepares the design for Phases 3 and 4 by reducing the number of components that the microarchitecture optimizer must deal with. In doing so, it groups components that will be

implemented using random logic gates rather than a bit-sliced layout.

Phase 3 applies time reduction techniques. The microarchitecture design is originally tuned for area by the technology mapper. Therefore, in this phase, the microarchitecture optimizer operates on critical paths, making necessary time for area tradeoffs.

Phase 4 works on non-critical paths, attempting to reduce the design's area. During area optimization, some microarchitecture components may be merged with others for logic optimization. These types of optimizations must be performed after timing optimization because once components are merged, the microarchitecture optimizer cannot recognize the original component functionality. This information is necessary for some of the timing optimization techniques.

```
Procedure Optimize_Microarchitecture (microarchitecture design)
Begin
General_Design_Improvements(microarchitecture_design)
Random_Logic_Grouping(microarchitecture_design)
Identify critical path set
While (Critical path set is not empty)
   Begin
   critical_path = select_critical_path(critical_path_set)
   Timing_Optimization(critical_path)
   Remove critical_path from critical_path_set
   End
Identify non critical path set
While (Non critical path set is not empty)
   Begin
   non_critical_path = select_non_critical_path(non_critical_path_set)
   Area_Optimization(non_critical_path)
   Remove non_critical_path from non_critical_path_set
   End
End
```

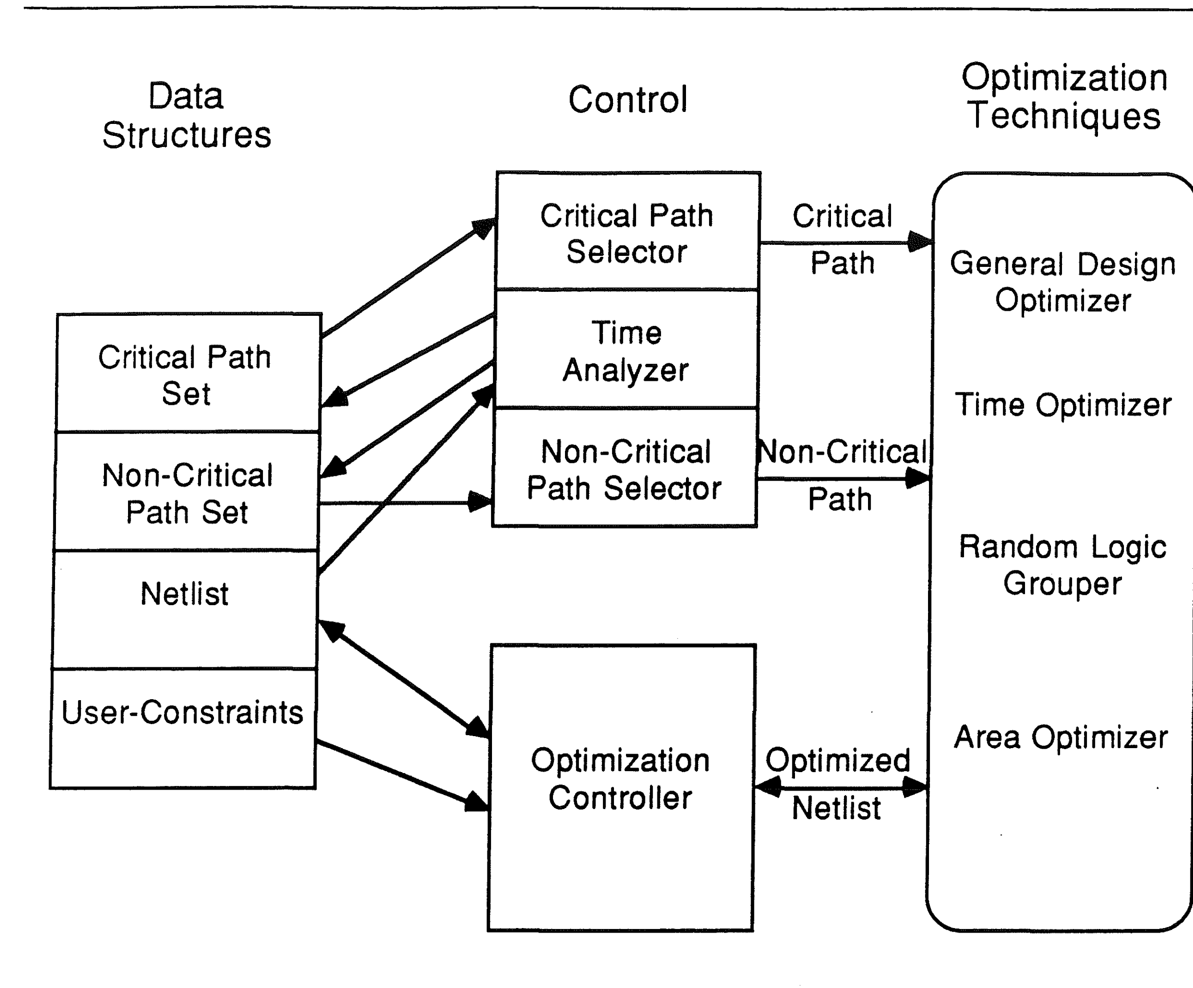


Figure 36. Overview of Microarchitecture Optimization

Phase 2 groups random logic components for logic optimization. Microarchitecture optimizations are not performed on random logic gates. Instead, they are passed to the database which has tools for restructuring the logic to meet a set of constraints passed by the microarchitecture optimizer. Thus Phase 2 prepares the design for Phases 3 and 4 by reducing the number of components that the microarchitecture optimizer must deal with. In doing so, it groups components that will be

implemented using random logic gates rather than a bit-sliced layout.

Phase 3 applies time reduction techniques. The microarchitecture design is originally tuned for area by the technology mapper. Therefore, in this phase, the microarchitecture optimizer operates on critical paths, making necessary time for area tradeoffs.

Phase 4 works on non-critical paths, attempting to reduce the design's area. During area optimization, some microarchitecture components may be merged with others for logic optimization. These types of optimizations must be performed after timing optimization because once components are merged, the microarchitecture optimizer cannot recognize the original component functionality. This information is necessary for some of the timing optimization techniques.

```
Procedure Optimize_Microarchitecture (microarchitecture design)
Begin
General_Design_Improvements(microarchitecture_design)
Random_Logic_Grouping(microarchitecture_design)
Identify critical path set
While (Critical path set is not empty)
   Begin
   critical_path = select_critical_path(critical_path_set)
   Timing_Optimization(critical_path)
   Remove critical_path from critical_path_set
   End
Identify non critical path set
While (Non critical path set is not empty)
   Begin
   non_critical_path = select_non_critical_path(non_critical_path_set)
   Area_Optimization(non_critical_path)
   Remove non_critical_path from non_critical_path_set
   End
End
```

6.3.1. General Design Improvements

To improve the overall design, Phase 1 proceeds as follows:

Procedure General_Design_Improvements (microarchitecture design)
Begin
Merge Similar Units (from inputs to outputs of the design)
Merge UnSimilar Units (from inputs to outputs of the design)
Apply Minimization Rules (from inputs to outputs of the design)
Perform Common Subexpression Elimination
End

Phase 1 begins with components of similar types being merged. As mentioned earlier, this is necessary for refactoring and common subexpression recognition. Further it reduces the number of components and hence makes minimization rules easier to apply. For example, performing the optimization of Figure 23 would be more difficult to discover if the multiplexor containing the common signal A were factored into two multiplexors, each containing the signal A.

Next unsimilar components are merged using a set of rules. These rules also reduce the number of components in the design. Once all merging is complete, a set of minimization rules can be applied to clean up redundant and unnecessary logic in the microarchitecture design. Up to this point, all optimizations reduce the number of microarchitecture components in the design. The next step, common subexpression extraction, increases the number of microarchitecture components but reduces the actual amount of logic required to implement them. It allows hardware that is

redundant in a number of components to be shared. Common subexpression elimination is not performed on random logic as this can be performed by logic optimization tools.

6.3.2. Random Logic Grouping

Phase 2 collects certain types of components to be optimized together as random logic. This is performed as follows:

Procedure Random_Logic_Grouping (microarchitecture design)
Begin
Group Logic Gates into random logic components
Group Non-Bit-Sliceable Components into random logic components
Group Components with Constant Inputs into random logic components
End

Phase 2 groups components that will then be optimized as a single random logic component. During this phase, three types of components can be grouped: 1) gates, 2) components for which bit-slicing is difficult, and 3) bit-sliceable components that have constants as inputs. Type 1 components, gates such as NAND, AND, and XOR, each have a lower-level technology specific implementation. For example, at the microarchitecture level, one could have a 12-input AND gate. Of course, a gate with this many inputs is usually not physically implementable as a single gate. Thus the technology-specific design is constructed from smaller gates that are available in the specified technology. Phase 2 groups all random logic gates at the microarchitecture level that are connected together and forms a single

component of type "random logic", as shown in Figure 37.

Type 2 components, for which bit-slicing is difficult, such as a decoder, will also be grouped with the random logic gates that they are connected with. Finally, bit-sliceable components with constant inputs will be added to the random logic set. Components connected to the outputs of the type 3 components will also be grouped into the random logic since during logic optimization, the constants will often propagate through.

For each microarchitecture component, the database has a file containing a set of boolean equations that describe the behavior of the component. As mentioned earlier, the equations can represent sequential logic as well as combinational logic.

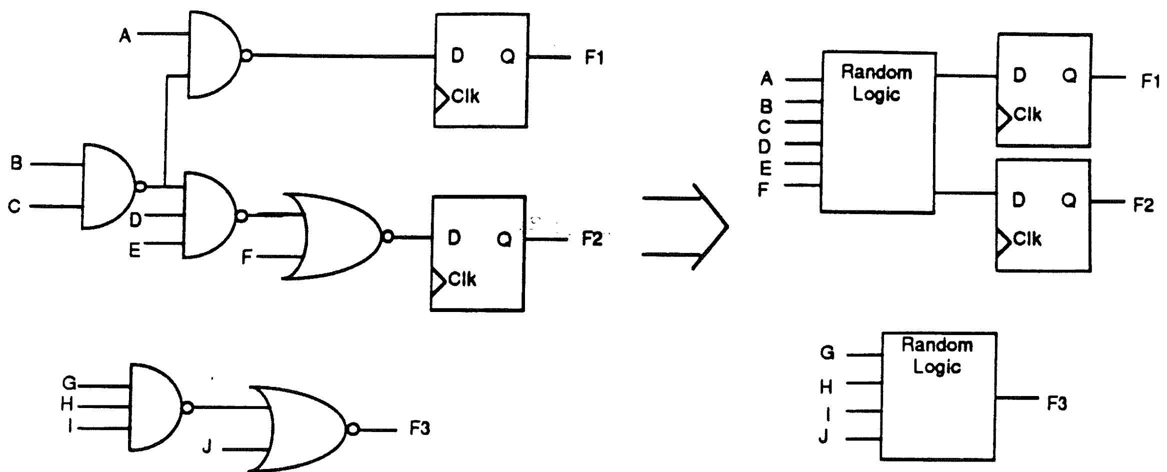


Figure 37. Random Logic Grouping

The microarchitecture optimizer can request that the database create a new component by merging two component's equation files. Logic optimization on this new component can then be performed by tools in the database.

6.3.3. Timing Optimization

Timing analysis is performed as the first stage of Phase 3. Delays and setup times for each component can be found by querying the database. The timing analyzer calculates four types of worst delay: 1) input pins to registers, 2) register to register, 3) registers to output pins, and 4) input pins to output pins. The worst delays at the design's output pins are compared with the required delays that are entered by the user. Output pins with negative slacks do not meet the delay constraints. Slack is computed as:

slack = actual delay - required delay

Required delays are also calculated at each register data input. The required delay is calculated based on the required maximum clock width, which is entered by the user:

required delay = max clock width - setup time

Actual delays are calculated based on the worst delay to the register's input, the setup time for that input, and the worst delay to the clock input of the register:

actual delay = worst delay to register input + setup time -
worst delay to register clock input

The slack is then computed from the actual and required delay values. A slack value is found for every component's output pin in a similar manner by subtracting the actual delay from the required delay.

After timing analysis, the goal of timing optimization is to make sure that no component's outputs have a negative slack value. Any component having such an output is said to be on a critical path. Ideally these negative slack values are raised to zero, with any value over zero representing over optimization (assuming area/time tradeoffs must be made).

Timing optimization is performed for each critical path. The worst critical path (ie., the one having the largest negative slack) is processed first. Timing optimization along the critical path proceeds as follows:

Procedure Timing_Optimization (Critical Path)
Begin
Swap Equivalent Signals
Factor
New Component Style Selection
Merge Multiple Components for Optimization
End

Phase 3 operates on microarchitecture components along the critical paths. It uses factoring, signal swapping, new component selection, and merging of components in order to reduce delay. The timing optimization phase ends as soon as

there are no critical paths.

Signal swapping is performed first since there is no area increase associated with it. Usually, however, improvements in delay from this type of optimization are small. Factoring is employed in the second step to produce shorter paths for critical signals. This technique usually increases the area only slightly. The set of required delays is calculated for each input to the output of the factorable component. These delays are passed to the factoring routine which then attempts to factor in a manner that will meet those delays.

In the third step of Phase 3, the optimizer selects new component styles by querying the database to find out what components are available with smaller delays. The component that comes closest to satisfying the required delays at each output is selected. Thus the optimizer tries to set each of the slacks at the output pins to zero.

Having failed to fix all critical paths with the previous three steps, the microarchitecture optimizer attempts to combine bit-sliceable components into a random logic component and query the database to apply logic optimization. In addition, the database can use a transistor sizing program to size the transistors in a fashion that will meet the time constraints. By using larger transistor sizes than those used in the bit-sliced approach (where transistor sizes are fixed), it may be possible to produce a faster component. If indeed the database returns a faster component, the microarchitecture optimizer will switch the layout style to a custom layout. Of

course, the random logic approach combined with the large transistor sizes results in larger area.

6.3.4. Area Optimization

Finally, Phase 4 performs area optimizations along non-critical paths. It mainly employs new component selection and component merging. Components that have outputs with positive slacks are examined for possible area/time tradeoffs. Area optimization operates as follows:

```
Procedure Area_Optimization (Non-Critical Path)
Begin
   New Component Style Selection
   Merge Multiple Comps for Optimization
End
```

New component selection includes choosing a bit-sliced layout style for components where doing so results in an area reduction. Some components, such as multiplexors, may need to be factored in order to use a layout module generator. For example, only 4-to-1 and 2-to-1 multiplexors may be available. An 8 to 1 multiplexor would then need to be factored. This can be achieved by the algorithm presented earlier.

In some cases, a layout module generator exists but contains more functions than are required. For example, consider an ALU. The bit-slice of the module generator may perform addition, subtraction, eight logical functions (eg., NAND,

AND), and a set of comparison functions (eg., equal, greater than, zero). If only the addition operation, the comparison functions, and a logical AND are required, the layout module generator performs more functions than are necessary. Thus a random logic implementation will probably produce the smallest area design. If there is already a random logic component connected to the ALU, the ALU can be merged into the random logic component and the logic reoptimized.

An alternative approach to generating the random logic design is to separate the groups of functions that need to be performed. For example, Figure 38 shows

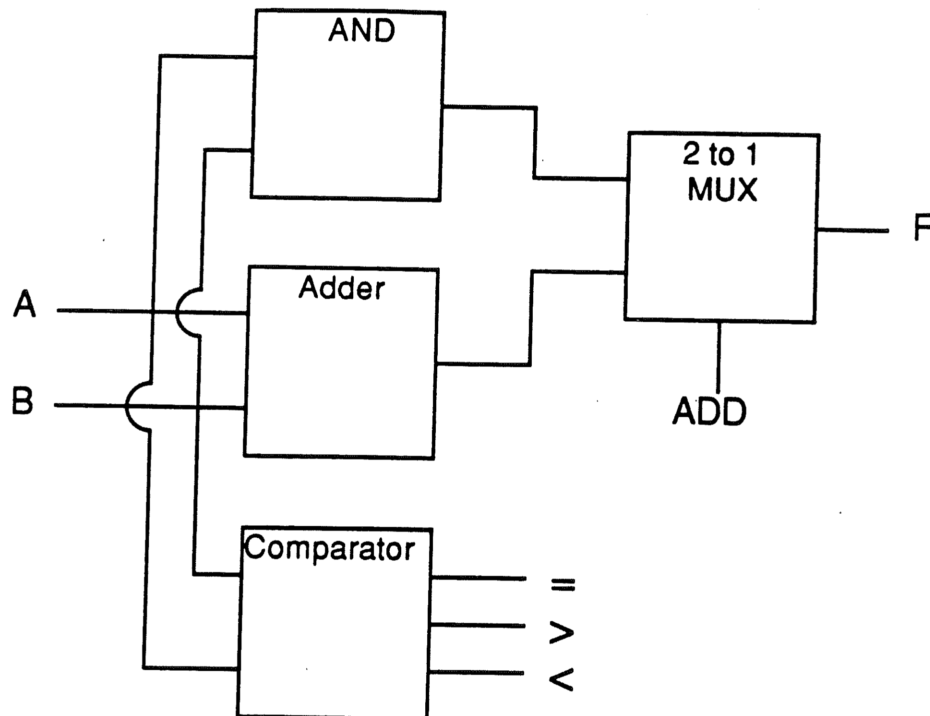


Figure 38. Option for performing ALU functions

that three groups of functionality can be generated for our example of the ALU: an arithmetic unit (adder), a comparator, and a logical AND. A multiplexor is used to choose the addition function or the logical AND. In this case all components can be implemented using the layout module generators and placed in the bit-sliced data-path during layout.

6.4. Experimental Results

This section presents experiments performed using MILO. A number of design examples were written in VHDL, then run through VSS to generate the initial microarchitecture design. The designs were then run through MILO with the following four strategies:

(1) Optimize the design for area and produce an underlying gate-level design for each microarchitecture component.

(2) Optimize the design for time and produce an underlying gate-level design for each microarchitecture component.

(3) Optimize the design for area and use the module generators for all bit-sliceable microarchitecture components, gate-level designs for all other components.

(4) Optimize the design for time and use the module generators for all bit-sliceable microarchitecture components, gate-level designs for all other components.

To get an idea of how good these optimizations were compared to a traditional straight logic optimization, the output design from VSS consisting of microarchitecture components was completely expanded into a flat gate-level design. This design was run through MISII and then through a transistor sizing program. The logic optimization was also performed for both area and time. Thus each example was run six different ways.

Five different benchmarks were run through MILO: Rockwell Counter, Armstrong Counter, and three different versions of DRACO: Draco2, Draco3, and Draco Schematic. A short description of each of these designs and their results are shown in the following sections. In the final section a comparison of all of the optimizations performed by MILO and MISII is made and conclusions are drawn.

6.4.1. Benchmark Experiments

6.4.1.1. Rockwell Counter

The Rockwell Counter benchmark was supplied by Rockwell International and is a design used in telephone switching networks. It has four inputs as shown in the block diagram of Figure 39: 1) CLK, the system clock, 2) RST, which performs a synchronous reset of the counter, 3) DTI, a 12-bit data input, and 4) LDE, a control line which loads the counter with input DTI. It has only one output, DTO, which represents the value of the count.

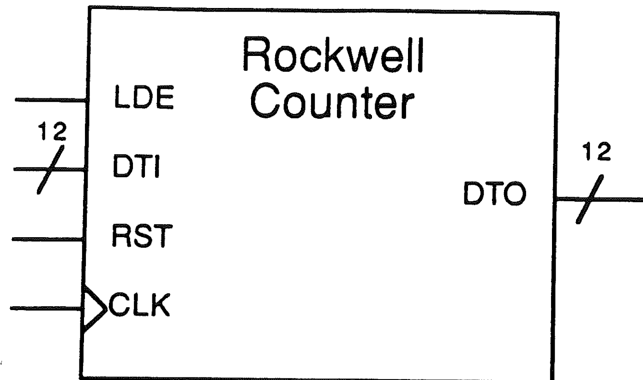


Figure 39. Block Diagram of the Rockwell Counter

The counter is a divide by 3328 counter that operates as follows:

(1) The counter has a start count of 0 and a terminal count of 3327.

(2) The counter increases by 208 on each clock edge. If the count is greater than 3327, the counter will start at the previous start count plus 26 (eg., the first time: 0 + 26). If the previous start count plus 26 is greater than 207, then the count will start at the previous start count plus 1.

(3) There are 26 sequences (ie., 26 start counts) before the counter reaches 3327 and wraps back to 0.

(4) The counter has an active high load enable which synchronously loads the counter. The state machine must adjust to the new state so as to keep the

same counting sequence.

Table 4 shows the optimization results for the Rockwell Counter when optimizing for time, while Table 5 shows the results when optimizing for area. In this example, the design with the fewest transistors is achieved using the module generators during microarchitecture optimization. The area of the designs employing only gates are roughly equal. When comparing time results, MILO's optimization with gates produced the smallest delay, followed by MILO's optimization using the module generators. The optimization by MISII produced the largest delay.

Table 6 displays the tradeoff of time for area when comparing the time optimized designs with the area optimized designs. The change in time and area is shown as a percentage. For example, MILO's optimization using only gates achieves a 37%

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	206.0	1800
MILO	Module Generators	233.5	1484
MISII	Gates	222.5	1344

Table 4. Time Optimization Results for the Rockwell Counter

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	327.0	1158
MILO	Module Generators	337.5	1056
MISII	Gates	413.0	1170

Table 5. Area Optimization Results for the Rockwell Counter

Optimization Tool	Optimization Style	time difference (%)	area difference (%)
MILO	Gates	-37.0	+55.4
MILO	Module Generators	-30.8	+40.5
MISII	Gates	-46.4	+14.8

Table 6. Time/Area Tradeoffs for Rockwell Counter

improvement in time at a cost of a 55% increase in area when comparing the time optimized design with the area optimized design. This table illustrates that fairly substantial reductions in time can be achieved at a cost of additional area. Finally,

Figure 40 compares the three optimization approaches (MILO with gates, MILO with module generators and gates, and MISII) graphically. The curve represents the potential to achieve area/time tradeoffs between the best area optimized design and the best time optimized design, although this ability has not actually been tested.

6.4.1.2. Armstrong Counter

The Armstrong Counter is a benchmark adapted from [Arms89]. As shown in the block diagram of Figure 41, it has four inputs: 1) CLK, the system clock, 2)

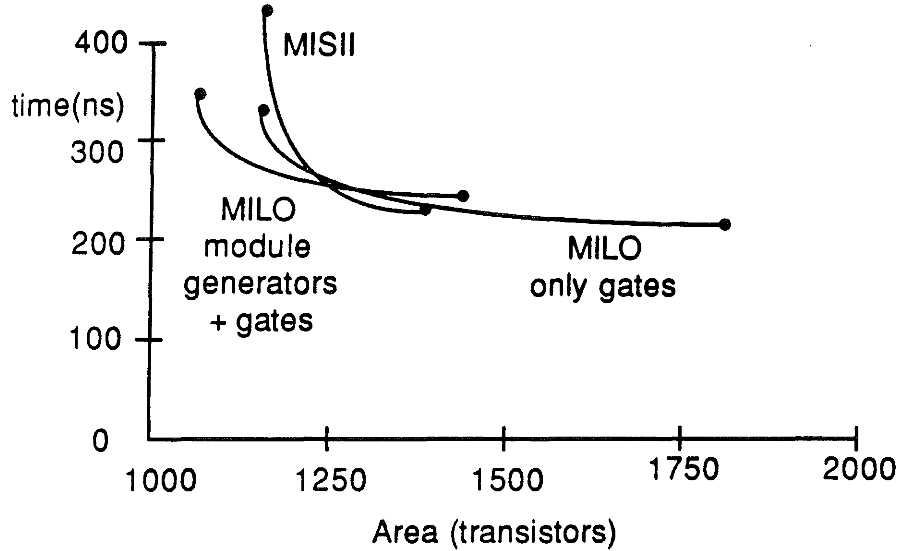


Figure 40. Three Optimization Approaches for the Rockwell Counter

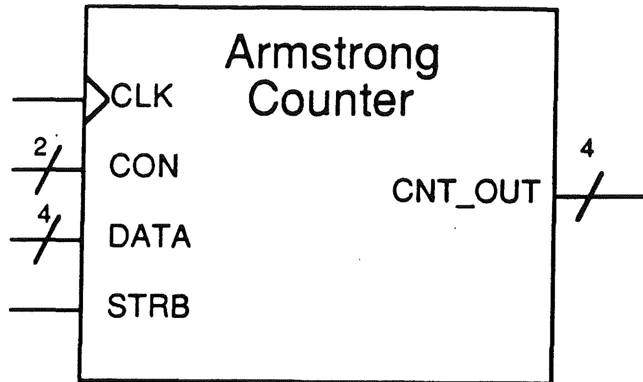


Figure 41. Block Diagram of Armstrong Counter

CON, a two-bit input that selects which function the counter will perform, 3) DATA, a four-bit input that determines the end count for the counter, and 4) STRB, an asynchronous line that loads the two values DATA and CON into registers. It has a single four-bit output, CON_OUT.

The behavior of the Armstrong Counter is as follows:

(1) On the rising edge of STRB, the values of DATA and CON will be loaded.

(2) The counter can perform four functions as specified by the value of CON: clear the counter, load a limit register, count up to a limit, or count down to a limit.

Table 7 shows the optimization results for the Armstrong Counter when optimizing for time, while Table 8 shows the results when optimizing for area. In this

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	28.0	486
MILO	Module Generators	20.0	393
MISII	Gates	43.5	484

Table 7. Time Optimization Results for the Armstrong Counter

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	38.0	486
MILO	Module Generators	20.0	395
MISII	Gates	74.5	460

Table 8. Area Optimization Results for the Armstrong Counter

example, MILO's optimization with module generators produced the smallest delay, followed by MILO's optimization using the gates. The optimization by MISII produced the largest delay. When comparing area results, the design with the fewest

transistors is again achieved using the module generators during microarchitecture optimization. The area of the MISII design is smaller that that produced by MILO using gates.

Table 9 displays the tradeoff of time for area when comparing the time optimized designs with the area optimized designs. The change in time and area is shown as a percentage. Optimization by MILO using only gates shows a 26% reduction in delay with no increase in transistor count. This indicates that the improvement in time was mainly due to changes in transistor sizing. Finally, Figure 42 compares the three optimization approaches (MILO with gates, MILO with module generators and gates, and MISII) graphically. Again, the curve represents the potential to achieve area/time tradeoffs between the best area optimized design and the best

Optimization Tool	Optimization Style	time difference (%)	area difference (%)
MILO	Gates	-26.3	+0.0
MILO	Module Generators	-0.0	-0.5
MISII	Gates	-41.6	+5.2

Table 9. Area/Time Tradeoffs for the Armstrong Counter

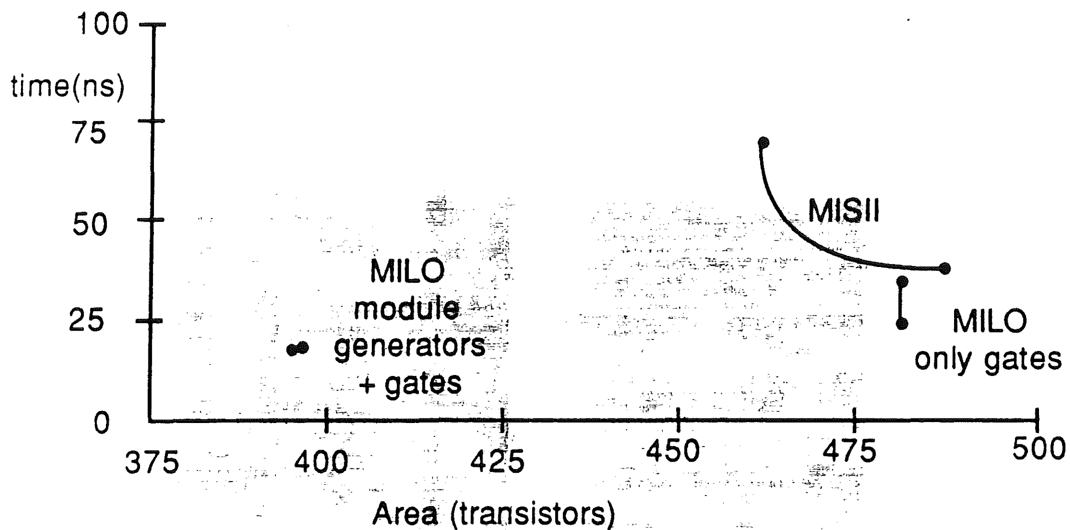


Figure 42. Three Optimization Approaches for Armstrong Counter

time optimized design. For the Armstrong Counter, MISII has the largest distance between the area and time optimized designs.

6.4.1.3. DRACO

DRACO is another benchmark obtained from Rockwell International and is the most complex of all our benchmarks. A block diagram of DRACO is shown in Figure 43, consisting of nine inputs and one output. DRACO is primarily intended to interface 16 I/O ports to a microprocessor's 8-bit multiplexed address/data bus and control signals. DRACO was developed by Rockwell as an ASIC chip.

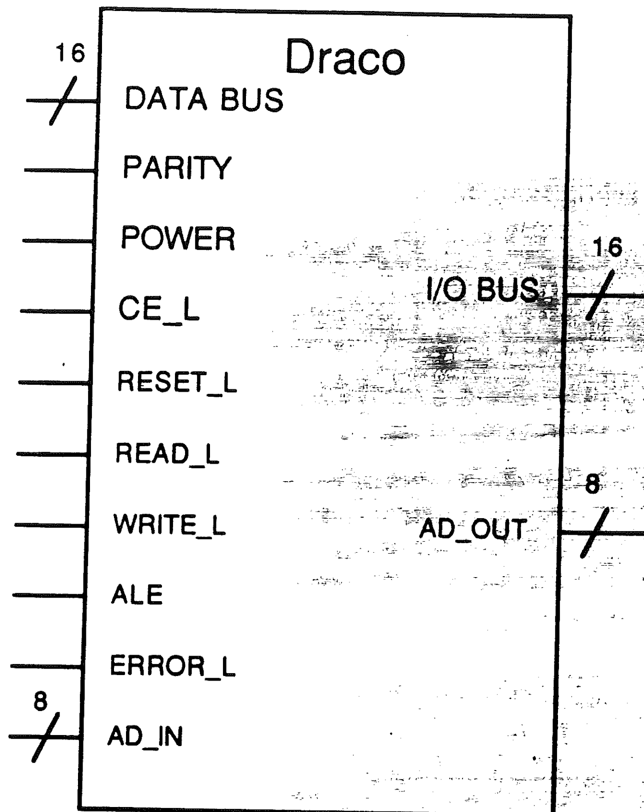


Figure 43. Block Diagram of DRACO

Three VHDL descriptions of the DRACO chip were written. Each description represented DRACO at a different level of abstraction. "Draco Schematic" was derived from the logic schematic provided by Rockwell International. "Draco2" and "Draco3" were more abstract versions and each used a different style of modeling in VHDL. Thus the designs produced by VSS from each of these descriptions are quite different.

Table 10 through Table 15 demonstrate optimization results for time and area

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	194.5	5868
MILO	Module Generators	109.0	3644
MISII	Gates	283.5	5800

Table 10. Time Optimization Results for Draco2

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	226.5	5152
MILO	Module Generators	117.5	3390
MISII	Gates	342.0	4668

Table 11. Area Optimization Results for Draco2

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	101.5	5544
MILO	Module Generators	135.0	3968
MISII	Gates	138.5	4202

Table 12. Time Optimization Results for Draco3

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	115.5	5298
MILO	Module Generators	176.5	3492
MISII	Gates	174.5	4206

Table 13. Area Optimization Results for Draco3

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	205.0	5658
MILO	Module Generators	135.5	3026
MISII	Gates	149.0	4216

Table 14. Time Optimization Results for Draco Schematic

Optimization Tool	Optimization Style	time (ns)	area (# of transistors)
MILO	Gates	206.0	4486
MILO	Module Generators	136.5	3018
MISII	Gates	258.0	3762

Table 15. Area Optimization Results for Draco Schematic

on the DRACO examples. For examples "Draco2" and "Draco3", MILO's optimizations proved to be the best in terms of delay. Optimization by MILO using module generators resulted in the best designs in terms of area. Table 16 through Table 18

Optimization Tool	Optimization Style	time difference (%)	area difference (%)
MILO	Gates	-14.2	+13.9
MILO	Module Generators	-7.2	+7.5
MISII	Gates	-17.1	+24.1

Table 16. Time/Area Tradeoffs for Draco2

Optimization Tool	Optimization Style	time difference (%)	area difference (%)
MILO	Gates	-12.1	+4.6
MILO	Module Generators	-23.5	+13.6
MISII	Gates	-20.6	-0.1

Table 17. Time/Area Tradeoffs for Draco3

Optimization Tool	Optimization Style	time difference (%)	area difference (%)
MILO	Gates	-0.5	+26.1
MILO	Module Generators	-0.7	+0.3
MISII	Gates	-42.2	+12.1

Table 18. Time/Area Tradeoffs for Draco Schematic

show the tradeoff of time for area when comparing the time optimized and area optimized designs. Figure 44 through Figure 46 compare the three optimization approaches.

In addition to comparisons of transistor counts, two layouts were generated by SLAM for Draco2 designs as an additional comparison. Figure 47 shows the layout for Draco2 that was produced from the design optimized by MILO using the module generators. The layout consists of two sections: the left hand portion is a custom layout consisting of random logic. The right hand portion of the layout is the bit-sliced datapath produced by the module generators. Figure 48 shows the layout for Draco2 that was produced from the design optimized by MISII. It consists entirely of a custom layout for random logic. As would be expected, the design with the module generators is smaller than the random logic design. The total layout area of

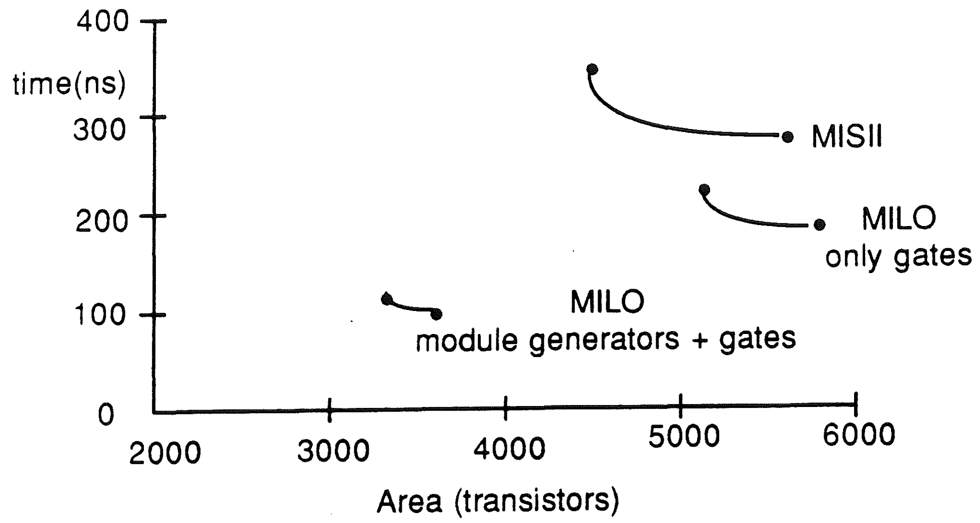


Figure 44. Three Optimization Approaches for Draco2

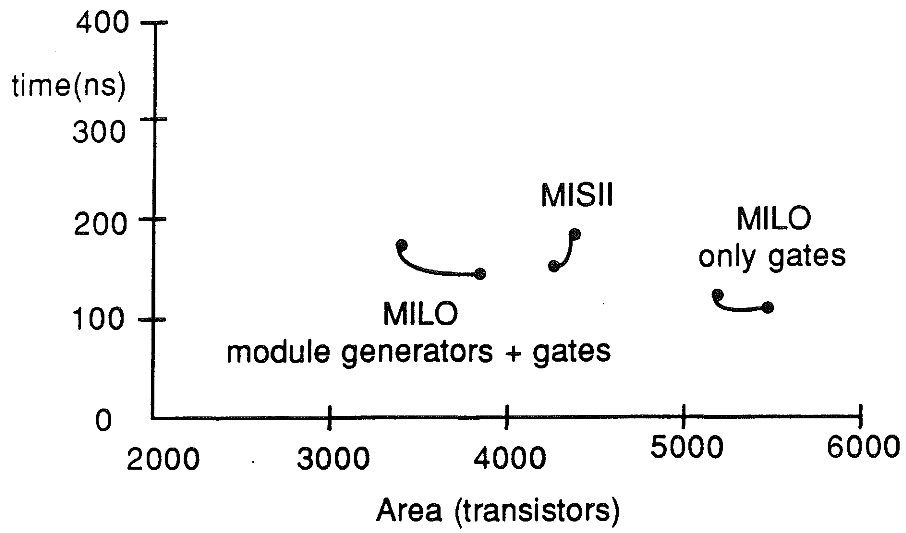


Figure 45. Comparison of Three Optimization Approaches for Draco3

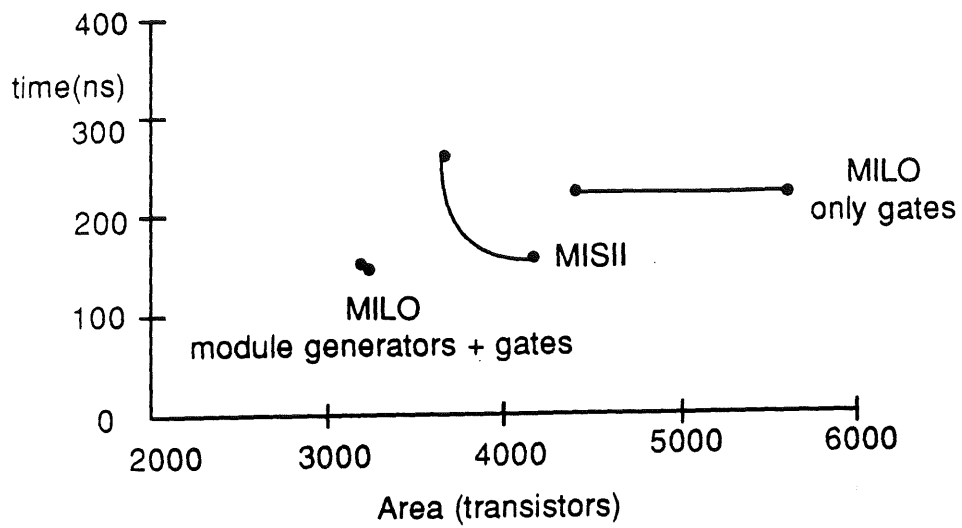


Figure 46. Three Optimization Approaches for Draco Schematic

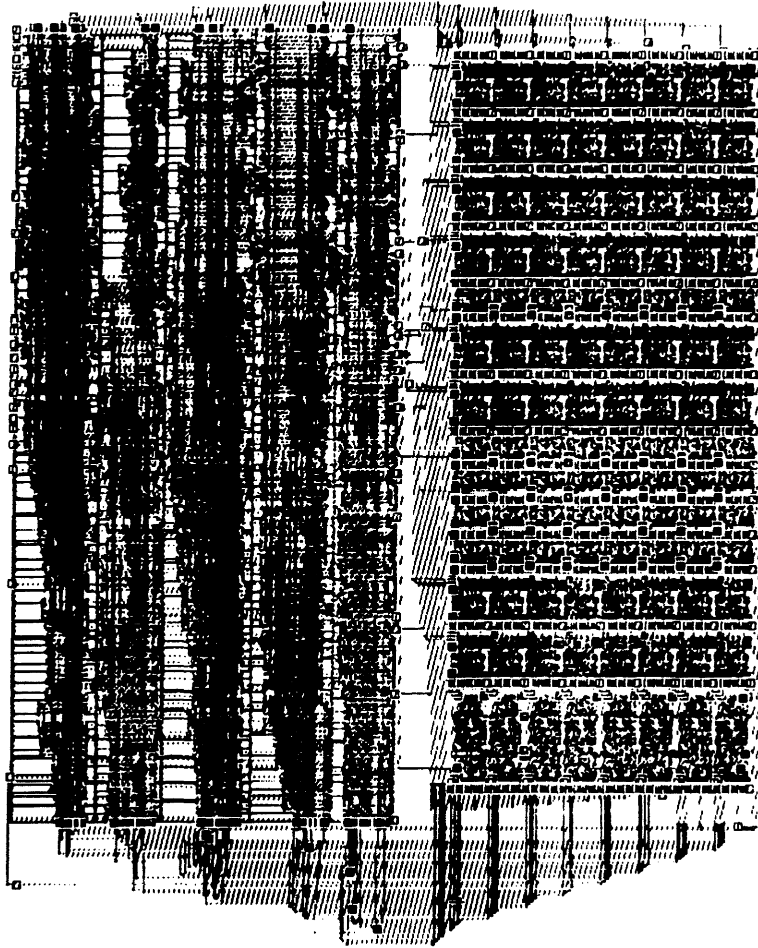


Figure 47. Layout of Module Generator Design for Draco2

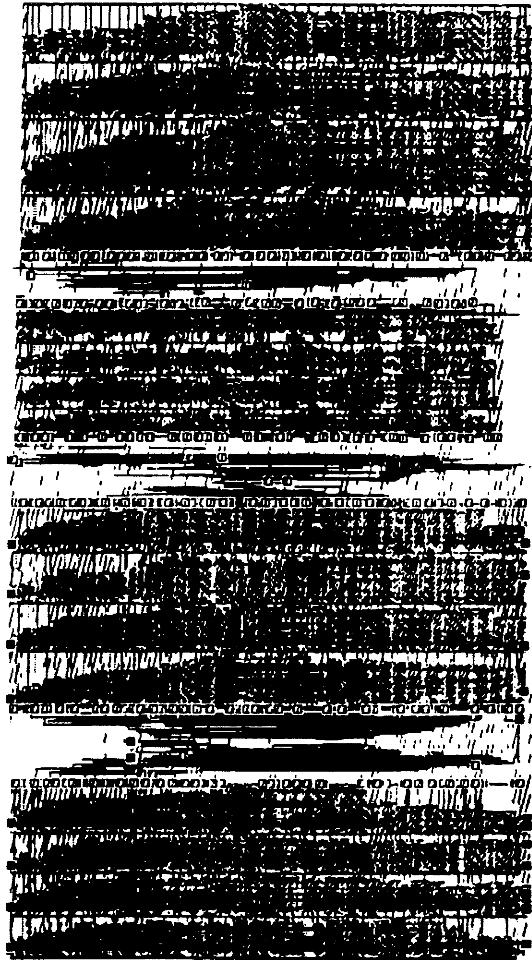


Figure 48. Layout of MISII Design for Draco2

the random logic design is 14,668,600 square micrometers compared with only 8,592,672 square micrometers in the MILO module generator design. This represents an area difference of 70%.

6.4.2. Analysis

Table 19 compares optimization by MILO using only gates and straight logic optimization by MISII. MILO when using only gates produces faster designs in four of the five cases, ranging from 7% faster to 35% faster. In one of the five cases

Benchmark	MILO (%)	MISII (%)
Draco2	69	100
Draco3	73	100
Draco Schematic	138	100
Armstrong Cntr.	64	100
Rockwell Cntr.	93	100

Table 19. Comparison of MILO and MISII Timing Optimization

MILO is slower by 37%. This demonstrates that MILO can produce faster designs on average.

Table 20 compares optimization by MILO using only gates and straight logic optimization by MISII of the examples for area. In four of the five cases, MISII produces a design with a smaller area. This is to be expected as the MILO logic optimizer is primarily geared for time optimization. However, MILO's optimization with module generators compensates by providing area efficient bit-sliced layouts. The best designs in terms of area were usually achieved when using module generators as

Benchmark	MILO (%)	MISII (%)
Draco2	111	100
Draco3	132	100
Draco Schematic	119	100
Armstrong Cntr.	106	100
Rockwell Cntr.	99	100

Table 20. Comparison of MILO and MISII Area Optimization

shown in the tables that follow.

Table 21 compares optimization using modules generators with optimization using only gates and optimizing for time. Table 22 shows the same comparison for area. The table shows that in most of the cases, optimization with the module generators produced a design with the smallest area and fastest speed. Table 23 and Table 24 make the same comparison with module generators but use the MISII results as the base for comparison.

Benchmark	MILO (module gen.) (%)	MILO (gates) (%)
Draco2	56	100
Draco3	133	100
Draco Schematic	85	100
Armstrong Cntr.	71	100
Rockwell Cntr.	113	100

Table 21. MILO gate vs. MILO module generator designs (Time)

Benchmark	MILO (module gen.) (%)	MILO (gates) (%)
Draco2	66	100
Draco3	66	100
Draco Schematic	67	100
Armstrong Cntr.	81	100
Rockwell Cntr.	91	100

Table 22. MILO gate vs. MILO module generator designs (Area)

Benchmark	MILO (module gen.) (%)	MISII (gates) (%)
Draco2	38	100
Draco3	97	100
Draco Schematic	91	100
Armstrong Cntr.	46	100
Rockwell Cntr.	105	100

Table 23. MISII vs. MILO module generator designs (Time)

Benchmark	MILO (module gen.) (%)	MISII(gates) (%)
Draco2	73	100
Draco3	83	100
Draco Schematic	80	100
Armstrong Cntr.	86	100
Rockwell Cntr.	90	100

Table 24. MISII vs. MILO module generator designs (Area)

These experiments demonstrate the effectiveness of MILO in generating efficient designs for either time or area. By optimizing the microarchitecture design instead of simply expanding the design and performing logic optimization, superior designs can be produced. Further, the results demonstrate flexibility in generating designs with different layout styles -- those using only gates and those incorporating a bit-slice capacity.

CHAPTER 7.

CONCLUSION

7.1. Summary

This thesis presented novel approaches to both the problems of logic synthesis and microarchitecture optimization.

First, a method of synthesis for custom layout was presented. Higher-quality layouts can be produced by taking layout parameters into account during the synthesis process. Traditional logic synthesis techniques work well for layouts using standard cells but achieve less than optimal results for custom layouts. They fail to consider layout parameters like transistor sizing and complex gate formation. We implemented an algorithm for high-performance CMOS designs that incorporates these parameters and compared our results with those of a traditional logic synthesis system, MISII. Our results demonstrate speed improvements for both custom and standard cell layout Further, layout driven synthesis has greater control over area/time tradeoffs.

Second, this thesis described how to incorporate logic synthesis tools into the process of microarchitecture optimization. The logic synthesis tools are used to provide feedback on individual microarchitecture components. The microarchitecture optimizer can then use this information to make modifications at the register-transfer level -- changes having much greater effect then those that can be employed at the logic level alone.

Techniques were introduced for improving the microarchitecture structure and for employing constraint driven synthesis based on the user's requirements for time and area. These techniques include the capability for mixing layout styles such as custom layout for random-logic components and bit-slicing for regularly structured components. In this manner the entire design, control logic and datapath, can be optimized at the same time. Further, a new methodology was presented for microarchitecture-level optimization that greatly reduces the amount of technology-specific knowledge necessary to perform the optimizations. Microarchitecture components are generated by a database based on a set of parameters from the microarchitecture optimization tool. Thus the microarchitecture optimizer does not need to deal with multiple logic optimization tools, layout module generators, transistor sizing tools, etc.

7.2. Future Research

A number of improvements can be made to MILO both at the logic and microarchitecture levels. First, at the logic level, more research is needed to refine

the interaction of factorization, complex gate formation, and transistor sizing to produce superior results. Improvements should include: (a) re-examining timing after the initial transistor sizing phase and complex gate formation, then redoing the transistor sizing based on the new timing information. and (b) performing area optimization along non-critical paths.

Second, at the microarchitecture level, improvements to the initial technology-mapping scheme can be made. Currently the technology-mapping program uses all-gate implementations for each of the microarchitecture components. If module generator implementations were used for bit-sliceable components, less time would be taken to generate and optimize gate-level designs that are later replaced.

Currently microarchitecture optimization only modifies the design without changing the state assignment. However, further timing improvements could be obtained by employing "clock-splitting" -- the insertion of registers at strategic points to reduce the register to register delays and allow a faster clock to be used. In addition to adding registers, the control logic equations would need to be modified.

Finally, feedback could be passed to the microarchitecture optimizer from layout tools. Additional delays are added to the design during the layout phase largely due to routing. If after layout, the timing constraints were not met, the microarchitecture optimizer could use delay estimates from layout tools to identify long paths and then make structural changes at the microarchitecture level to reduce these delays.

APPENDIX A.

LOPT Program Description

A.1. Tutorial

This section explains how to run "lopt", the logic optimization program. The input is a IIF description (described in Appendix B), the output is a gate-level VHDL file. This output VHDL file can then be passed to LES for generation of a custom layout.

The lopt program can be run interactively or in batch mode as demonstrated in the examples that follow. The examples illustrates how to use the "lopt" shellscript to generate a 5-bit up/down counter from an IIF description. The first example shows the use of "lopt" in the interactive mode. User input to program prompts are shown in < >. The user is prompted for the report file name in which the estimated area and delays of the optimized design are placed, the capacitive load that an output pin must drive, the maximum clock width, maximum setup time, the maximum allowed delay for an output pin, and a filename in which the optimized VHDL netlist is placed. If 0 is entered for the delay values, "lopt" will attempt to generate the best delay. In the interactive mode, only one load and maximum delay value can be entered. All pins are then assumed to have these values. Using the batch mode, different values can be specified for separate pins.

To run the program interactively, type:

lopt -i count5.iif

The user will be prompted as follows:

VHDL Output File? <count5.vhdl>

Enter Report File Name: <report>

Enter the load for output pin: Q_4 <15>

Enter the clock width: <0>

Enter the maximum setup time: <0>

Enter the required delay for output pin: Q_4 <0>

The generated VHDL file is placed in count5.vhdl and can then be passed on to LES for layout.

To run "lopt" in batch mode, four parameters must be included on the command line: 1) IIF file, 2) VHDL output file name, 3) lopt parameter file, and 4) lopt report file name. Parameters 2 and 4 are files created by "lopt" so filenames are required on the command line. Parameters 1, and 3 are required input files to "lopt".

The following example demonstrates the command line for batch mode.

To run the program in batch mode, type:

```
lopt -i lcnt4.iif -p lcnt4.p -r lcnt4.r -v lcnt4.vhdl
```

The user will not be prompted for any inputs. Examples of the IIF and parameter files follow. Next the file formats for the parameter and report files are described.

A.2. Input Files

File count5.iif:

```
NAME= Counter_5_Load_Enable_Updown;

INORDER= D_0 D_1 D_2 D_3
       D_4 CLK  LOAD  ENA
       DWUP ;

OUTORDER= Q_0 Q_1 Q_2 Q_3
       Q_4 MINMAX  RCLK ;
C_0=1;
CLK0=(CLK@(~l ENA));
C_1=(((C_0*Q_0)*!DWUP)+((C_0*!Q_0)*DWUP));
Q_0=((Q_0!=C_0)@((~r CLK0)~a((0/(!LOAD*!D_0)),(1/(!LOAD*D_0)))));
C_2=(((C_1*Q_1)*!DWUP)+((C_1*!Q_1)*DWUP));
Q_1=((Q_1!=C_1)@((~r CLK0)~a((0/(!LOAD*!D_1)),(1/(!LOAD*D_1)))));
C_3=(((C_2*Q_2)*!DWUP)+((C_2*!Q_2)*DWUP));
Q_2=((Q_2!=C_2)@((~r CLK0)~a((0/(!LOAD*!D_2)),(1/(!LOAD*D_2)))));
C_4=(((C_3*Q_3)*!DWUP)+((C_3*!Q_3)*DWUP));
Q_3=((Q_3!=C_3)@((~r CLK0)~a((0/(!LOAD*!D_3)),(1/(!LOAD*D_3)))));
OVFUNF=(((C_4*Q_4)*!DWUP)+((C_4*!Q_4)*DWUP));
Q_4=((Q_4!=C_4)@((~r CLK0)~a((0/(!LOAD*!D_4)),(1/(!LOAD*D_4)))));
MINMAX=(CLK*OVFUNF);
RCLK=((CLK*OVFUNF)+!OVFUNF);
```

File counter5.p:

```
rdelay Q_0 40
oload Q_0 10
rdelay Q_1 40
oload Q_1 10
rdelay Q_2 40
oload Q_2 10
rdelay Q_3 38
oload Q_3 26
rdelay Q_4 19
oload Q_4 19
rdelay MINMAX 40
oload MINMAX 10
rdelay RCLK 40
oload RCLK 10
rsetup  30
cwidth  20
```

A.3. LOPT Parameter File Format

The parameter file provides constraints to the optimizer on: worst case delay to an output pin, the load on an output pin, the worst setup time for any input pin, and the maximum clock width. Delays are entered in terms of nanoseconds. Load on output pins is entered in terms of number of unit sized transistors that need to be driven. For example, a load of 10 means the output gate must be able to drive 10 unit sized transistors.

The format of the parameter file is as follows:

```
rdelay output_name worst_delay
oload  output_name load
rsetup worst_setup_time
cwidth maximum_clock_width
```

Example parameter file (note that order of the parameters does not matter):

```
oload Q[0] 15
oload Q[3] 15
cwidth 4.0
rsetup 9.0
oload Q[1] 10
rdelay Q[3] 10.89
rdelay Q[1] 4.0
rdelay Q[2] 5.0
rdelay Q[0] 15.89
oload Q[2] 10
```

A.4. LOPT Report File Output Format

The report file contains information on path delays and area of the design. There are four information types that are reported: the combinational delay from an input to an output (CD), the setup delay for an input pin (SD), the minimum clock width (CW), the transistor area (TA), and the number of transistors (NT).

The format of the report file is as follows:

```
CW minimum_clock_width
CD input_name output_name delay
SD input_name setup_time
TA transistor_area
NT number_of_transistors
```

Example report file:

```
CW 36.566666
CD CLK Q[3] 9.800000
CD CLK Q[2] 6.750000
CD CLK Q[1] 8.100000
CD CLK Q[0] 10.684999
SD UPDOWN 28.466667
TA 722
NT 212
```

A.5. LOPT Usage

"lopt" is a shellscript for performing logic optimization and technology mapping for custom layout from an IIF input description. The shellscript can be run without providing any parameters -- the user will be prompted interactively for required information. Parameters can also be used on the command line to reduce the amount of prompting or to run the shellscript completely in batch mode.

Shellscript Usage:

```
lopt [-i IIF_input_file]
     [-o optimization_type] [-t technology_mapping_type]
     [-m mis_script_style]
     [-p parameterfile] [-r reportfile] [-v VHDL_output_file]
```

Options:

-i: IIF input file

-o: Optimzation type. Takes value "a" for area, "t" for time.
Default value is "a".

-t: Technology mapping type. Takes value "a" for area, "t" for time.
Default value is "a".

-m: MISII script type. Takes value "b" for best optimization,
"m" for memory conservation. Using the "b" option, MISII
generates all minterms and hence can run out of memory. This
will create a memory fault. Using the "m" option will avoid
this problem but will not do as good of an optimization.
Default value is "b".

-p: Input parameter file for "lesopt" technology mapping program.
The parameter file format is described in the file "parm.spec".

-r: Output report file for "lesopt" technology mapping program.
The report file format is described in the file "report.spec".

-v: VHDL output file

Examples of Use:

```
lopt

lopt -i ex/ex1.iif

lopt -i ex/lcnt4.iif -v lcnt4.vhdl

lopt -i ex/lcnt4.iif -v lcnt4.vhdl -o t -t a -m m lcnt4.vhdl
```

APPENDIX B.

IIF

B.1. Introduction

In order to describe microarchitecture level components (such as counters, shift registers, etc.), we need a format capable of describing sequential, asynchronous behavior, and I/O conversion. In this document, we define the Irvine Intermediate Form (IIF) which extends a boolean expression language with clocking and asynchronous behavior in order to describe generic components composed of logic gates, flip-flops with asynchronous set and reset, and interface components. MILO can accept an IIF description, and generate a netlist of logic gates and complex cells.

IIF is an extension of the Berkeley EQN (equation) format for describing boolean expressions. Besides providing the basic boolean operations of AND, OR, NOT, XOR, and XNOR, IIF contains operators for specifying a D flip-flop with asynchronous set and reset and operators for tristate, delay, and wire-or. Any component in the GENUS library is representable in IIF. An IIF file has the following format:

design name
input/output declarations

list of equations

An IIF description is divided into 2 parts. The first part describes the input and output signal names. The second part is the design description. All the external signals must be declared before they are used in the second part. Temporary signals are not declared. Each IIF statement is delimited by ;.

B.2. IIF declarations

An IIF declaration consists of the following elements.

NAME = name of design;
INORDER = input signal list;
OUTORDER = output signal list;

The declaration part of IIF specifies the name of the design and external inputs and outputs of the design. The keyword **NAME** is used to assign the design name. Input signals are declared in the **INORDER** signal list and output signals in the **OUTORDER** signal list. Temporary signals (ie., those that are not external inputs or outputs to the design) are not declared.

B.3. IIF expression

An IIF expression is a boolean equation with some new operators. The followings are operators of IIF.

Binary operators:

+	OR
*	AND
!=	XOR
==	XNOR

=	assignment
~t	tristate
~w	wireor
@	AT (for specify D flip flop clock)
~a	asynchronous assignment for D flip flop
/	asynchronous AT

Unary operators:

!	NOT
~f	falling edge trigger clocking (for flip-flop)
~r	rising edge trigger clocking (for flip-flop)
~h	level clocking active at high (for latch)
~l	level clocking active at low (for latch)

The +, *, !, !=, and == operators are standard Boolean operations. Tristate operations are represented as: I0 ~t Control, where *I0* is the input and *Control* determines whether the output is *I0* or a high impedance state. IIF extends boolean expressions with operators for expressing D flip-flops with asynchronous set and reset inputs (@, ~a). The ~f, ~r, ~h, and ~l operators are used to specify the clocking of the D flip flop.

Sequential logic is represented in the form:

Var = (boolean input equation)
@ (clock_operator clock_expression)
~a (asynchronous_set_reset_expression)

For example, a falling edge triggered with set and reset is described as follows:

Q=(D @ ~f clk) ~a (0/!reset,1/!set)

The expression **Q = D @ ~f clk** means that **Q** is set to D at the falling edge

(denoted by **˜f**) of the signal **clk**. The expression **˜a (0/!reset,1/!set)** indicates that **Q** is set to 0 when reset signal is 0 (denoted by **0/!reset**) and is set to 1 when set signal is 0 (denoted by **1/!set**). The asynchronous expression is a list of descriptions in the form as **value/condition**. It means that when the **condition** expression is true, the output is set to **value**.

B.4. Examples of IIF Usage

Several examples of IIF descriptions are now presented.

The first example is a register with parallel load. A schematic of the register is shown in Figure 49. The IIF description is as follows:

```
NAME = REGISTER4;
INORDER = Load I0 I1 I2 I3 CP Clear;
OUTORDER = A0 A1 A2 A3;
not_load = !Load;
Clr = ˜b Clear
A0 = ((I0*Load) + (A0*not_load)) @ (˜f CP) ˜a(0/!Clr);
A1 = ((I1*Load) + (A1*not_load)) @ (˜f CP) ˜a(0/!Clr);
A2 = ((I2*Load) + (A2*not_load)) @ (˜f CP) ˜a(0/!Clr);
A3 = ((I3*Load) + (A3*not_load)) @ (˜f CP) ˜a(0/!Clr);
```

A second example shows how the 4-bit adder/subtracter of Figure 50. can be represented in IIF.

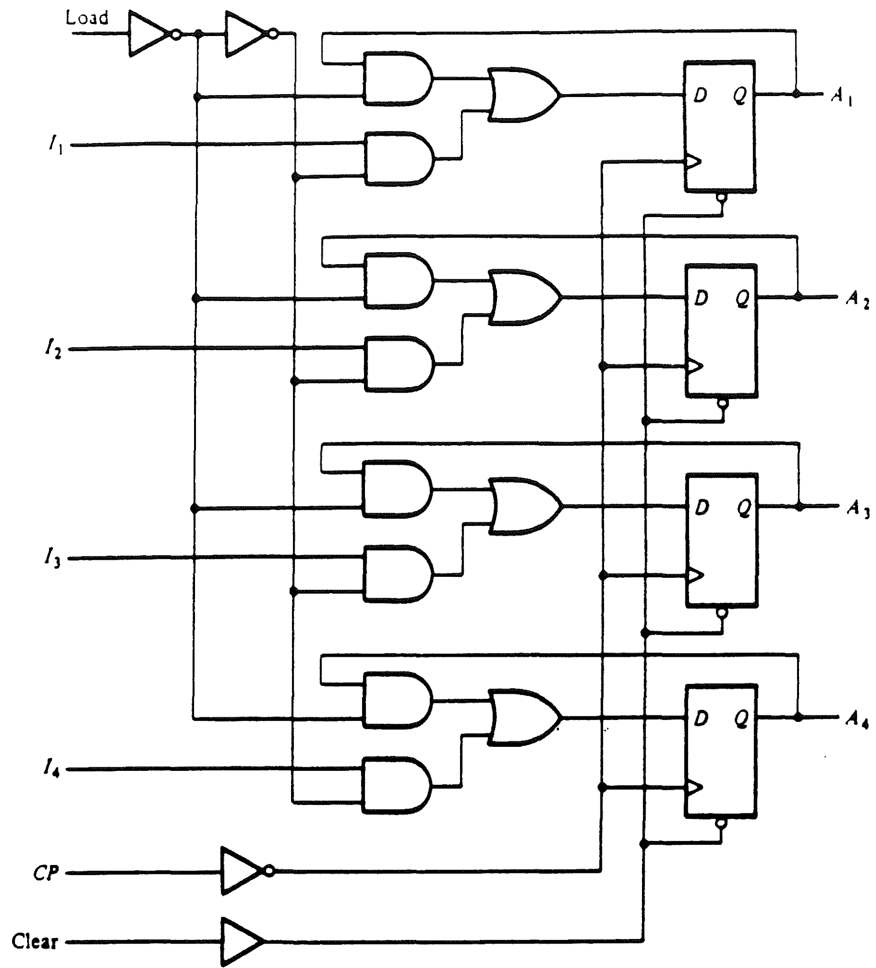


Figure 49. Register with Parallel Load

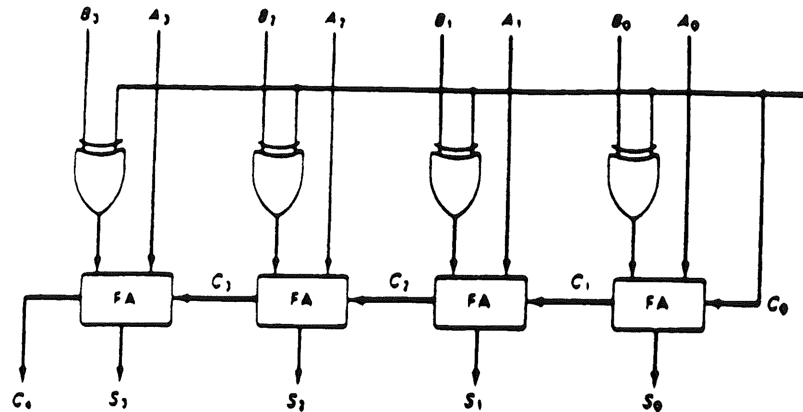


Figure 50. Four-bit Adder/Subtracter

IIF file for a 4-bit adder/subtracter:

```
NAME=adder_subtracter4
INORDER=A[0] A[1] A[2] A[3] B[0] B[1] B[2] B[3] ADDSUB;
OUTORDER=O[0] O[1] O[2] O[3] COUT;
B1[0]=ADDSUB!=B[0];
B1[1]=ADDSUB!=B[1];
B1[2]=ADDSUB!=B[2];
B1[3]=ADDSUB!=B[3];
C[0]=ADDSUB;
O[0]=A[0]!=B1[0]!=C[0];
C[1]=A[0]*B1[0]+C[0]*A[0]+C[0]*B1[0];
O[1]=A[1]!=B1[1]!=C[1];
C[2]=B1[1]*B1[1]+C[1]*B1[1]+C[1]*B1[1];
O[2]=B1[2]!=B1[2]!=C[2];
C[3]=B1[2]*B1[2]+C[2]*B1[2]+C[2]*B1[2];
O[3]=B1[3]!=B1[3]!=C[3];
C[4]=A[3]*B1[3]+C[3]*A[3]+C[3]*B1[3];
COUT=C[4];
```

Finally, a third example shows the use of tristate and wireor operators in IIF. Note that because of the priority of tristate operators, it is important to use parentheses to ensure that you get the behavior you desire. For example, "A + B ~t C" is interpreted as "(A + B) ~t C". The schematic for a tristate connection is shown in Figure 51.

IIF file for the tristate connection:

```
NAME = tristate3
INORDER = A B C E;
OUTORDER = F;
t1 = A ~t E;
```

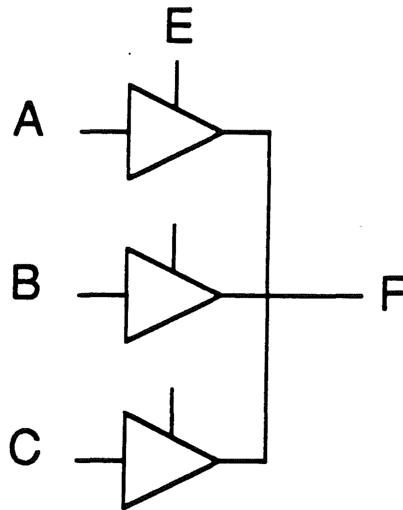


Figure 51. Tristate Connection

```
t2 = B ~t E;
t3 = C ~t E;
F = t1 ~w t2 ~w t3;
```

B.5. Operator Precedence

The precedence of the operators are ranked from highest priority to lowest priority. Operators on the same line have the same precedence.

(1) !, ~b, ~r, ~f, ~h, ~l, ~a

(2) !=, ==

(3) *

(4) +, ~w, ~t, @, /

APPENDIX C.

MILO Program Description

C.1. Introduction

This section explains how to run "milo" starting from a VHDL description. The output is a structural VHDL file consisting of microarchitecture components.

C.2. Milo Usage

"milo" is a program for performing microarchitecture optimization from a VHDL input description. The program can be run without providing any parameters -- the user will be prompted interactively for required information. Parameters can also be used on the command line to reduce the amount of prompting or to run the program completely in batch mode.

Milo Usage:

milo [-i VHDL_input_file] [-p parameter_file]
[-r report_file] [-o output_VHDL_file] [-g]

Options:

-i: VHDL input file

-p: Input parameter file for specifying required delays and capacitive loads on output pins.

-r: Output report file describing transistor count and timing of the optimized design.

-v: VHDL output file

-g: Optimize for gates only, do not use module generators for bit-sliced components.

C.3. Milo Parameter File Format

The parameter file provides constraints to the optimizer on: worst case delay to an output pin, the load on an output pin, and the maximum clock width.

The format of the parameter file is as follows:

```
rdelay output_name worst_delay
oload  output_name load
cwidth maximum_clock_width
```

Example parameter file (note that order of the parameters does not matter):

```
oload Q[0] 15
oload Q[3] 15
cwidth 4.0
oload Q[1] 10
rdelay Q[3] 10.89
rdelay Q[1] 4.0
rdelay Q[2] 5.0
rdelay Q[0] 15.89
oload Q[2] 10
```

C.4. MILO Report File Output Format

The report file contains information on path delays and area of the design. There are four information types that are reported: the worst-case combinational delay to an output (WD) -- measured in nanoseconds, the minimum clock width (CW) -- measured in nanoseconds, the transistor area (TA) -- measured in square micrometers (active transistor area only, routing area is not included), and the number of transistors (NT).

The format of the report file is as follows:

CW minimum_clock_width
CD input_name output_name delay
TA transistor_area
NT number_of_transistors

Example report file:

```
CW 36.566666
WD Q[3] 9.800000
WD Q[2] 6.750000
WD Q[1] 8.100000
WD Q[0] 10.684999
TA 722
NT 212
```

BIBLIOGRAPHY

[Arms89] Armstrong, J., "Chip Level Modeling with VHDL", Prentice Hall, 1989.

[BiBr88] Birmingham, W.P, Brennan, A., Gupta, A.P., and Siewiorek, D.P., "MICON: A Single-Board Computer Synthesis Tool", *IEEE Circuits and Devices Magazine*, January, 1988.

[BoHa87] Bostick, D., Hachtel, G., Jacoby, R., Lightner, M., Moceyunas, P., Morrison, C., Davenscroft, D., "The Boulder Optimal Logic Design System", *ICCAD*, 1987.

[BrFa85] Brownston, L., Farrell, R., Kant, E., and Martin, M., "Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming", *Addison Wesley Publishing Company*, 1985, pp. 228-239.

[Br84] Brayton, R., et al., "ESPRESSO IIC: Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishers, Netherlands, 1984.

[Br86] Brayton, R., et al., "Multiple-Level Logic Optimization System", *ICCAD*, 1986.

[BrRu87] Brayton, R., Rudell, R., Sangiovanni-Vincentell, A. and Wang, A., "MIS: A Mutiple-Level Logic Optimization System", *IEEE Transactions on Computer-Aided Design*, Vol. CAD-6, No. 6, Nov. 1987.

[BuMa85] Buric, M.R., and Matheson, T.G., "Silicon Compilation Environments", *Proceedings of the Custom Integrated Circuits Conference*, May, 1985.

[CaRo85] Camposano, R., and Rosenstiel. W., "A Design Environment for the Synthesis of Integrated Circuits", 11*th EUROMICRO Symposium on Microprocessing and Microprogramming*, Sept. 1985.

[ChGa90] Chen, G.D., and Gajski, D.D., "An Intelligent Component Database For Behavioral Synthesis", 27*th DAC*, 1990.

[CoBa85] Cohen, W., Bartlett, K., and de Geus, A., "Impact of Metarules in a Rule Based Expert System for Gate Level Optimization", *Proc. IEEE Int'l. Symp. on Circuits and Systems*, May 1985.

[Chu65] Chu, Y., "An ALGOL-like Computer Design Language", *Communications ACM*, Oct. 1965.

[Ci87] Cirit, Mehmet A., "Transistor Sizing in CMOS Circuits", 24*th DAC*, 1987.

[DaJo80] Darringer, J., Joyner, W., "A New Look at Logic Synthesis", 17*th Design Automation Conference*, 1980.

[DaJo81] Darringer, J., Joyner, W., Berman, C., and Trevillyan, L., "Logic Synthesis Through Local Transformations", *IBM J. Res. Develop.*, 25, no. 4, July 1981.

[DoLe89] Domic, A., Levitin, S., Phillips, N., Thai, C., Shiple, T., Bhavsar, D., and Bissell, C., "CLEO: a CMOS Layout Generator", *ICCAD*, 1989.

[Dutt88] Dutt, N.D., "GENUS: A Generic Component Library for High Level Synthesis", *Technical Report* 88-22, University of California, Irvine, Sept. 1988.

[EnNa85] Enomoto, K., Nakamura, S., Ogihara, T., and Murai, S., "LORES-2: A Logic Reorganization System", *IEEE Design & Test*, October 1985.

[FiDu85] Fishburn, J. P., and Dunlop, A. E., "TILOS: A Posynomial Programming Approach to Transistor Sizing", *ICCAD*, 1985.

[Fo85] Forgy, C., "OPS83 User's Manual and Report", 1985.

[GaGr84] Garrison, K., Gregory, D., Cohen, W., and de Geus, A., "Automatic Area and Performance Optimization of Combinational Logic", *Proc. IEEE Int'l. Conference on Computer-Aided Design* , 1984.

[GeCo85] de Geus, A. and Cohen, W., "A Rule-Based System for Optimizing Combinational Logic", *IEEE Design & Test*, August 1985.

[GrBa86] Gregory, D., Bartlett, K., de Geus, A., and Hachtel, G., "SOCRATES: A System for Automatically Synthesizing and Optimizing Combinational Logic", *23rd Design Automation Conference*, 1986.

[GuPa90] Gupta, G., Pastorello, D., and House, G., "Timing Optimizations in a High-Level Synthesis System", *ICCD*, 1990.

[He87] Hedlund, Kye S., "Aesop: A Tool for Automated Transistor Sizing", *24th DAC*, 1987.

[JoMc87] Johannsen, D., McElvain, K., and Tsubota, S., "Intelligent Compilation", *VLSI Systems Design*, April 1987.

[JoTr86] Joyner, W., Trevillyan, Y., Brand, D., Nix, T., and Gundersen, S., "Technology Adaptation in Logic Synthesis", *23rd Design Automation Conference*, 1986.

[Kim87] Kim, J., "Artificial Intelligence helps cut ASIC Design Time", *Electronic Design*, June 11, 1987.

[Ke87] Keutzer, K., "DAGON: Technology Binding and Local Optimization by DAG Matching", *24th Design Automation Conference*, 1987.

[KoLu88] Kollaritsch, P., Lusky, S., Prasad, S., and Potter, N., "CLAY: A Malleable-cell Transistor Matrix Approach for CMOS Layout

Synthesis", *ICCAD*, 1988.

[LiGa88] Lin, Y-L. Steve, and Gajski, D., "LES: A Layout Expert System", *IEEE Trans. on Computer-Aided Design*, August 1988.

[LiGa89] Lis, J.S., and Gajski, D.D., "VHDL Synthesis Using Structured Modeling", 26*th DAC*, 1989.

[McDe82] McDermott, J., "R1: A Rule-Based Configurer of Computer Systems", *Artificial Intelligence*, 19(1) (1982).

[McPa88] McFarland, M., Parker, A., and Camposano, R., "Tutorial on High-Level Synthesis", 25*th Design Automation Conference*, 1988.

[ObKa88] Obermeier, Fred W., and Katz, Randy H., "An Electrical Optimizer that Considers Physical Layout", 25*th DAC*, 1988.

[OrGa86] Orailoglu, A., and Gajski, D., "Flow Graph Representation", 23*rd DAC*, June, 1986.

[PaGa87] Pangrle, B.M., and Gajski, D.D., "Design Tools for Intelligent Silicon Compilation", *IEEE Transactions on Computer-Aided Design*, Vol. 6, No. 6, November 1987.

[PaKn87] Paulin, P.G., and Knight, J.P., "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Transactions on Computer-Aided Design*, Vol. 8, No. 6, June 1987.

[PaPM86] Parker, A.C., Pizarro, J., and Milnar, M., "MAHA: A Program for Datapath Synthesis", 23*rd DAC*, 1986.

[SiSV90] Singh, K.J., and Sangiovanni-Vincentelli, A., "A Heuristic Algorithm for the Fanout Problem", 27*th DAC*, 1990.

[StMu86] Stroud, C.E, Munoz, R.R., and Pierce, D.A., "CONES: A System for Automated Synthesis of VLSI and Programmable Logic From Behavioral Models", *ICCAD*, 1986.

[Tj86] Tjiang, S., "Twig Reference Manual", January 1986.

[TrDi89] Trick, M.T., and Director, S.W., "LASSIE: Structure to Layout for Behavioral Synthesis Tools", 26*th DAC*, 1989.

[TsSi86] Tseng, C.J., and Siewiorek, D.P., "Automated Synthesis of Data Paths in Digital Systems", *IEEE Transactions on Computer-Aided Design*, Vol. 5, No. 3, July 1986.

[TsWe88] Tseng, C.J., Wei, R.S., Rothweiler, S.G., Tong, M.M., and Bose, A.K., "Bridge: A Versatile Behavioral Synthesis System", 25*th DAC*, 1988.

[VaGa88] Vander Zanden, N.B., and Gajski, D.D., "MILO: A Microarchitecture and Logic Optimizer", *25th DAC*, 1988.

[WeRo88] Wei, R.S, Rothweiler, R., and Jou, J.Y., "BECOME: Behavior Level Circuit Synthesis Based On Structure Mapping", *25th DAC*, 1988.

[WuGa90] Wu, C.H., and Gajski, D.D., "Silicon Compilation from Register-Transfer Schematics", *International Symposium on Circuits and Systems*, 1990.

[WuVG90] Wu, C.H., Vander Zanden, N., and Gajski, D.D., "A New Algorithm for Transistor Sizing in CMOS Circuits", *European Design Automation Conference*, 1990.

[YeGh88] Yen, H.C., Ghanta, S., and Du, H.C., "A Path Selection Algorithm for Timing Analysis", *25th DAC*, 1988.

3 1970 00882 3970

VITA

Nels Blake Vander Zanden was born on September 8, 1965, in Columbus, Ohio. He received a B.S. in computer science from Ohio State University in 1984 and a M.S. in computer science from the University of Illinois in 1986. During his term as a doctoral candidate, he received a university fellowship in 1984 and later worked as a research assistant at UIUC. Mr. Vander Zanden also worked during the summer of 1987 at Applied Micro Circuits Corporation in San Diego, CA, and taught computer hardware design classes at the University of California, Irvine in the winter quarters of 1990 and 1991. His non-academic interests include tennis, volleyball, and music.