LOGIC SYNTHESIS FOR VLSI DESIGN

by

Richard L. Rudell

Memorandum No. UCB/ERL M89/49

26 April 1989

COVER

LOGIC SYNTHESIS FOR VLSI DESIGN

by

Richard L. Rudell

Memorandum No. UCB/ERL M89/49

26 April 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Logic Synthesis for VLSI Design

Richard L. Rudell

University of California
Berkeley, California

Department of Electrical Engineering
and Computer Science

Abstract

Very Large Scale Integration (VLSI) currently allows hundreds of thousands of transistors in a single application-specific integrated circuit. The trend of increasing levels of integration has stressed the ability of the designer to keep pace. Traditional integrated circuit design has relied on analysis tools to measure the quality and correctness of a circuit before fabrication. However, only recently have synthesis tools been used to assist in the design process. The advantages of automatic synthesis include reduced design time, reduced probability of design error, and higher quality designs because more effort is focused at higher-levels in the design. Automatic placement and routing, a form of physical design synthesis, has become widely accepted over the last five years; however, logic design has, for the most part, remained a manual task. *Logic synthesis* is the automation of the logic design phase of VLSI design; that is, choosing the specific gates and their interconnection to build a desired function. For digital integrated circuits which are partitioned into control and data-path portions, design of the control logic is often the most time-consuming. It is generally on the critical path for timing, and, because of the complexity of producing a correct description of the control, it is often on the critical path for completion of the design. Therefore, tools to assist in logic design will have a large impact on the design of integrated circuits. However, the benefits of automatic logic design are lost if the result does not meet its area, speed, or power constraints. Therefore, a critical aspect of automatic logic synthesis is the optimization problem of deriving a high-quality design from an initial specification. This thesis provides a set of logic optimization algorithms which together form a complete system for logic synthesis in a VLSI design environment. Efficient, optimal algorithms are proposed for two-level minimization, multiple-level decomposition, and technology mapping. The techniques described in this thesis have been implemented in a software program called MIS. The design of a complex digital circuit is included as part of this thesis to demonstrate the application of logic synthesis to a realistic design problem.

Alberto Sangiovanni-Vincentelli

Prof. Alberto Sangiovanni-Vincentelli
Thesis Committee Chairman

Acknowledgements

The six years I have spent in the computer-aided design research group at Berkeley have been the most challenging of my life. Professors Alberto Sangiovanni and A. Richard Newton have created an exciting environment in which to work. I would like to thank my research advisor, Alberto Sangiovanni-Vincentelli, for inviting me to study at Berkeley and for providing me with a constant challenge to keep up with his energy and enthusiasm. I am also indebted to Alberto for introducing me to the problems in automatic logic synthesis.

Working with Robert Brayton, first at IBM, and then later as a professor at Berkeley, has always been rewarding. I thank Bob for stimulating many of the ideas which led to the work in this thesis. Discussions with Kurt Keutzer of AT&T provided the ideas which developed into the technology mapping algorithm described here. Finally, working closely with Albert Wang has always been enjoyable; I look forward to working more with Albert.

Several industrial visitors to Berkeley helped bring a strong dose of practicality into this work. I thank Ewald Detjens, then with Phillips Research, and Gary Gannot of Intel Corporation for their constant interaction. I would also like to acknowledge the past and present members of the Berkeley CAD group who have contributed to its diverse personality; specifically, Jeff Burns, David Harrison, Ken Kundert, Tom Laidig, Lorraine Layer, Rick McGeer, Peter Moore, Tom Quarles, Ellen Sentovich, Greg Sorkin, and Rick Spickelmier have made working at Berkeley enjoyable.

I would like to thank IBM Corporation for their support in the form of a Graduate Research Fellowship for three years. I would also like to acknowledge the support of the Defense Advanced Research Projects Agency under contract N00039-86-R-0365, the National Science Foundation under subcontract ECS-8430435, and the California State Micro Program with supporting funds from Advanced Micro Devices, AT&T, Intel Corporation, Phillips Research, and SGS Thomson.

I thank Alberto Sangiovanni and Robert Brayton for their time spent on a critical review of this thesis.

Finally, I would like to thank my wife Robin for her patience and support during the many long hours it has taken to complete this thesis.

Contents

Acknowledgements i

Table of Contents iii

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Logic Synthesis . . . 1
- 1.2 Previous Work . . . 4
  - 1.2.1 Two-level Minimization . . . 4
  - 1.2.2 Optimum Multi-level Synthesis . . . 5
  - 1.2.3 Modern Techniques . . . 7
- 1.3 Overview . . . 8

2 Two-level Minimization 9
- 2.1 Introduction . . . 9
- 2.2 Definitions . . . 11
- 2.3 Exact Minimization Algorithms . . . 14
- 2.4 ESPRESSO-EXACT . . . 16
- 2.5 Prime Generation Algorithm . . . 17
  - 2.5.1 Prime Generation using Consensus . . . 17
  - 2.5.2 Prime Generation from the Off-Set . . . 18
  - 2.5.3 Comparison of Prime Generation Techniques . . . 19
- 2.6 Essential and Partially Redundant Primes . . . 20
- 2.7 The Reduced Prime Implicant Table . . . 20
- 2.8 Solving the Minimum Cover Problem . . . 22
  - 2.8.1 Covering Table Reduction Steps . . . 23
  - 2.8.2 Gimpel's Reduction Step . . . 24
  - 2.8.3 Maximal Independent Set . . . 26
  - 2.8.4 Choice of the Branching Column . . . 28
  - 2.8.5 Implementation Details . . . 29
- 2.9 Experimental Results . . . 31

2.9.1 Classification of the Benchmark Set 32
2.9.2 Comparison with McBoole 34
2.10 Conclusions 36
2.11 PLA Test Set Classification 37

3 Algebraic Decomposition **41**
3.1 Introduction 42
3.2 Previous Work 43
3.2.1 Dietmeyer-Su Factoring 43
3.2.2 Local Transformation 44
3.2.3 Algebraic Techniques 45
3.3 Algebraic Techniques 46
3.3.1 Basic Definitions 46
3.3.2 Kernel Decomposition Algorithm 49
3.4 Rectangles and the Rectangle Covering Problem 49
3.4.1 Basic Definitions 50
3.4.2 Rectangles and the Maximal Set-Intersection Problem 51
3.4.3 Rectangles and Kernels 51
3.4.4 Complexity of Rectangle Covering 52
3.4.5 Exact Solution for Rectangle Covering 53
3.5 Application of Rectangle Covering 54
3.5.1 Common-Cube Extraction 54
3.5.2 Kernel-Intersection Extraction 57
3.6 Effect of Overlapping Rectangles 62
3.6.1 Cube-Extraction 62
3.6.2 Kernel Extraction 63
3.7 Rectangle Algorithms 64
3.7.1 gen_rectangles: Finding All Prime Rectangles 66
3.7.2 best_rectangle: Finding a Maximum-Value Prime Rectangle 67
3.7.3 ping_pong: Finding a Maximal-Value Rectangle 70
3.7.4 greedy_extract: Greedy Selection of Rectangles 73
3.7.5 covering_extract: Simultaneous Selection of Rectangles 74
3.8 Selective Collapse 78
3.9 Representation of Two-level Functions 80
3.10 Experimental Results 82
3.10.1 ISCAS Circuit Optimization 83
3.10.2 Comparison of best_rectangle and ping_pong 84
3.10.3 Comparison of Bounding Strategies 85
3.10.4 Comparison of Single-Output and Multiple-Output Optimization . . 85
3.11 Conclusions 86

4 Technology Mapping **89**
4.1 Introduction 90
4.2 Previous Work 92
4.2.1 Rule-Based Techniques 92

4.2.2 DAGON 95
4.3 Technology Libraries 96
4.3.1 Area Model 98
4.3.2 Delay Model 98
4.3.3 Technology Library Example 99
4.4 Technology mapping using DAG-Covering 101
4.4.1 Choice of Base Functions 106
4.4.2 Creating the Subject Graph 108
4.4.3 A DAG-Covering Algorithm 111
4.5 Tree-Covering Approximation 123
4.5.1 Dynamic Programming Algorithms 123
4.5.2 Optimal Tree-Covering 123
4.5.3 Delay Optimization 125
4.5.4 Partitioning the Subject Graph 132
4.5.5 Phase-Assignment Heuristics 134
4.5.6 Extension to Non-Tree Patterns 137
4.5.7 Complexity of Tree-Covering 137
4.6 Complete Complex-Gate CMOS Libraries 141
4.6.1 CMOS Complex Gates 142
4.6.2 Counting the Number of CMOS Complex Gates 144
4.6.3 Counting the NAND-Gate Trees for a Function 149
4.7 Experimental Results 155
4.7.1 Area versus Delay; IWLS-87 Library 155
4.7.2 Area versus Delay; IWLS-89 Library 157
4.7.3 Technology Mapping Library Comparison 159
4.7.4 Inverter Optimization Heuristics 160
4.8 Extensions and Open Issues 161
4.8.1 Sequential Technology Mapping 161
4.8.2 Multiple-Output Gates 162
4.8.3 Extension of the Base-Function Set 163
4.9 Conclusions 164
4.10 IWLS-89 Benchmark Library Description 165

5 Design Example 169
5.1 OCT Synthesis Tools 169
5.2 The Data Encryption Standard (DES) 171
5.3 The DES Algorithm 172
5.4 DES Implementation Decisions 174
5.4.1 Design Alternatives 174
5.4.2 Implementation Technology 176
5.5 Fully Combinational DES Design 177
5.6 Byte-Pipelined DES Design 178
5.7 Conclusions 180
5.8 DES Design Description 180

6 Conclusions 201
6.1 Future Work 202

A MIS 205
A.1 Introduction 205
A.2 Background 205
A.3 Program Organization 207
A.3.1 Package Conventions 207
A.3.2 Generic Packages 210
A.3.3 Core Packages 211
A.3.4 Optimization packages 212
A.3.5 New packages 213
A.4 Retrospect 213

Bibliography 217

List of Figures

3.1 Example rules used by LSS. 44
3.2 Algorithm **gen_rectangles_recur**. 68
3.3 Algorithm **gen_rectangles**. 69
3.4 Algorithm **ping_pong**. 71
3.5 Algorithm **ping_pong_row**. 72
3.6 Algorithm **greedy_row**. 73
3.7 Algorithm **greedy_extract**. 74
3.8 Algorithm **covering_extract**. 75
3.9 Algorithm **rect_prime_cover**. 75
3.10 Algorithm **rect_reduce**. 77
3.11 Algorithm **eliminate**. 80

4.1 Example rules from Socrates. 93
4.2 IWLS-87 benchmark library in MIS-II format. 100
4.3 Unoptimized set of logic equations. 102
4.4 Optimized set of logic equations. 102
4.5 Subject graph for the equations of Figure 4.4. 103
4.6 Pattern graphs for the IWLS-87 library. 104
4.7 A cover for the subject graph of Figure 4.5. 105
4.8 A second cover for the subject graph of Figure 4.5. 105
4.9 Coarse-resolution base function. 107
4.10 Optimal cover for a balanced-tree decomposition. 109
4.11 Optimal cover for an unbalanced-tree decomposition. 110
4.12 Tree-levelizing transformation. 110
4.13 Example of a match and an exact match. 113
4.14 Algorithm **generate_all_matches**. 113
4.15 Algorithm **make_edge_list**. 114
4.16 Sample pattern graph for **edge_list**. 115
4.17 Edge-list for the graph of Figure 4.16. 116
4.18 Algorithm **graph_match**. 117
4.19 Example graph for DAG-covering. 120
4.20 Algorithm **optimal_area_cover**. 124
4.21 Algorithm **optimal_delay_cover**. 128

4.22 Algorithm **delay_technology_map**. 130
4.23 Algorithm **optimal_area_under_delay_constraint**. 133
4.24 Alternate covering using the inverter-pair heuristic. 135
4.25 Adding inverter-pairs at a branch-point. 136
4.26 A CMOS complex gate. 143
4.27 AND-OR tree for $ab + (c + d)(ef + gh)$. 146
4.28 Algorithm **generate_nand_trees**. 153

5.1 DES Equations. 173
5.2 MIS technology library for the MSU library. 181
5.3 BDS description for onestage. 183
5.4 BDS description for the primitive functions. 185
5.5 BDS description for the byte-pipelined implementation. 189
5.6 BDNET description for the byte-pipelined implementation. 194
5.7 BDSIM output for the byte-pipelined implementation. 196

List of Tables

2.1 ESPRESSO-EXACT results for the PLA test set. 33
2.2 Comparison of Espresso-Exact and McBoole. 35

3.1 ISCAS circuit optimization results. 83
3.2 Comparison of ping-pong and best-rectangle. 84
3.3 Comparison of bounding strategies. 85
3.4 Comparison of single-output and multiple-output optimization. 86

4.1 A partial list of matches. 121
4.2 Number of gates satisfying an (s,p)-constraint. 149
4.3 All (3,3)-gates. 150
4.4 Number of two-input NAND-gate trees for an n-input AND. 152
4.5 Number of NAND-gate patterns for (s,p)-gates. 154
4.6 Benchmark circuits for technology mapping. 156
4.7 Area versus delay for the IWLS-87 Library. 157
4.8 Area versus delay for the IWLS-89 library. 158
4.9 Technology mapping with different libraries. 159
4.10 Effect of inverter-pair and inverter-branch-point heuristics. 161

5.1 Tools used for the design of DES. 170
5.2 Pin interface for the byte-pipelined DES design. 179
5.3 Run-time for each step in the DES design. 180

A.1 Packages in MIS-II. 208

Chapter 1

Introduction

Logic synthesis addresses the problem of translating a register-transfer level description of a design into an optimal logic-level representation. This chapter reviews the process of VLSI design and describes how logic synthesis fits into this process. This is followed by a brief history of previous work in optimal logic synthesis. The chapter concludes with a description of the organization of this thesis.

1.1 Logic Synthesis

Very Large Scale Integration (VLSI) technology is in wide use in modern digital systems. VLSI technology currently allows hundreds of thousands of transistors in a single application-specific integrated circuit (ASIC), and the level of integration is increasing at a rapid rate. This trend has stressed the ability of the designer to keep pace with the advances in technology. In this thesis, VLSI design refers to the design of a single integrated circuit to perform a complex digital function. This includes generic components which are designed once and manufactured in high-volume (such as microprocessors and memories), and integrated circuits built for a specific design (i.e., ASICs).

VLSI design proceeds through a number of distinct phases. The design begins with an understanding of the purpose of the circuit behavior; i.e., the inputs and outputs of the circuit and how they are related. The design representation at this level is communicated using natural languages, timing diagrams, and block diagrams. The first design phase is called *register-transfer level design.* A register-transfer level (RTL) representation for a design describes the registers, the operations which are performed on the values stored in

the registers, and the control conditions which sequence these operations. The next phase is called *logic design*. This is the task of converting the registers, computation blocks, and controllers from the RTL description into a logic-level representation using the available building blocks. The building blocks for digital design are low-level logical operations such as AND, OR, and NOT, and storage elements. The final phase is called *physical design*. In this step, the interconnection of building blocks are translated into a set of integrated circuit masks.

In traditional integrated circuit design, a wide range of computer-aided *analysis* tools are used to measure the quality and correctness of a circuit before fabrication. This includes tools for entering and manipulating a design and tools for verifying that the design meets its functional and performance goals. Tools exist which support the analysis of a design at the register-transfer level, the logic level, and the mask-layout level, and for verifying that the behavior of the design is the same between the levels. However, only recently have effective tools for assisting the *synthesis* of an integrated circuit become available.

The advantages of automatic synthesis in VLSI design are clear. They include reduced design time, reduced probability of design error, and higher-quality designs because more effort is focused at a higher-level. However, the use of computer-aided synthesis tools provide an increase in designer productivity only if designs of acceptable quality are produced.

Successful systems now exist which automate the physical design of integrated circuits and produce designs of high-quality. These systems are particularly effective for block-oriented design styles such as *gate-array*, *standard-cell*, and *sea-of-gates* (also known as *compacted-array*). The common feature of these design styles is that cells are placed in regular rows with metal interconnection between the rows. Gate-array and sea-of-gates are design styles for implementing a complete integrated circuit. A regular pattern is placed on the silicon and only the final metal layers are used to customize the circuit. The standard-cell design style is used for complete integrated circuits, but is also in wide use as part of full-custom VLSI design. For example, a large part of the control logic for the modern microprocessors is implemented using a standard-cell design style.

Despite the success at automating physical design, the problem of translating an RTL-level description into a logic-level description has remained, for the most part, a manual task.

The term *automatic logic synthesis* (or *logic synthesis*) is used in this thesis to

describe computer-aided design programs which assist in the logic design of a digital system. Logic synthesis starts with a register-transfer level description of a design and a description of the low-level cells available in the target technology and produces an optimal logic-level representation. The subject of this thesis is the design of algorithms and techniques for automatic logic synthesis of combinational circuits with special emphasis on the problems faced in VLSI design.

The starting point for logic synthesis is a technology-independent representation of a synchronous digital design at the register-transfer level. One representation for an RTL design is as a graph of components; that is, storage elements, such as master-slave flip-flops, and combinational logic elements which implement an arbitrary Boolean logic function. The design is *synchronous* if all cycles in the graph contain at least one storage element and if all of the storage elements are clocked by a common signal. In combinational logic synthesis, the placement of the storage elements is assumed fixed; the combinational components are extracted from the RTL graph resulting in a directed-acyclic graph. The logic synthesis problem is to convert this technology independent representation of the design into an optimum multi-level net-list in a given technology.

Translating a register-transfer level representation of a design into a logic-level representation is not difficult; however, straightforward translation leads to designs which are either too large or too slow, and hence unacceptable. The benefits of automating the logic design process are lost if the result does not meet its area, speed, or power constraints. Therefore, a critical aspect of automatic logic synthesis is the optimization problem of deriving a high-quality design from the initial specification.

The accepted optimization criteria for multi-level logic are to minimize the area occupied by the logic equations while satisfying the timing constraints placed on the longest path through the logic. Another criterion which is important for some technologies is to minimize the power of the final circuit. The area, delay, and power of a design before layout are estimated using models which predict the effects of physical design based on the cells and nets in the final design.

An important part of integrated circuit design is the manufacturing test which determines if a fabricated chip works as expected. A connection is untestable (or redundant) if replacing the connection with a constant value does not affect the functionality of the circuit. Despite the observation that a smaller circuit usually results from removing a redundant connection, there is the further problem that redundancies interfere with the

production-line testing of the integrated circuit. Therefore, another goal for logic synthesis is to produce designs with no redundancies.

The design of the optimal circuit which meets all of these constraints is a difficult problem due to the tremendous number of potential solutions for even a small set of logic equations. The size of VLSI circuits makes logic synthesis for VLSI a difficult optimization problem.

A paradigm for logic synthesis has emerged in the last five years which separates the complex problem of building an optimal circuit for a set of Boolean logic functions into two steps: *technology-independent optimization* and *technology mapping*. This approach for logic synthesis is continued in this thesis.

Technology-independent optimization derives an optimal structure for the circuit independent of the gates available in a particular technology. The techniques presented in this thesis include an algorithm for exact two-level minimization of logic functions and algorithms for decomposition of a two-level circuit into an optimal multiple-level circuit.

Technology mapping is the optimization step of selecting the particular gates from the library to implement an optimized logic network. Included in this thesis is an algorithm for optimal technology mapping based on a transformation of the problem into a graph-covering problem.

In both technology-independent optimization and technology mapping, special emphasis is given to the efficiency of the algorithms. The goal is to apply the techniques to nonhierarchical designs of tens of thousands of gates; this will allow the techniques to be applied in a VLSI design environment.

1.2 Previous Work

1.2.1 Two-level Minimization

Research over the last thirty years has led to efficient methods for implementing combinational logic in an optimal two-level form. One technique for physical design of an optimal two-level form in VLSI uses a Programmable Logic Array (PLA) [30]. The first algorithms for optimal two-level design were proposed by Quine [55] and improved by McCluskey [52]. These techniques provide a minimum two-level form and hence are unable to solve many problems with more than ten inputs. Effective heuristic techniques for two-level

minimization were introduced by *mini* [42]. Several other approximate approaches followed, including *presto* [21], *pop* [70], and *espresso* [19,59]. The approximate techniques depend on iterative improvement of a set of equations and give no guarantee as to the quality of the final result. However, large functions can be minimized using this approach. For example, *espresso* has been used to optimize PLA'S with fifty inputs and fifty outputs.

The problem with two-level design is that there are many designs for which the two-level representation is inappropriate. For example, the function which converts an n-bit input string into a $\log_2 n$-bit count of the number of bits which have value one requires $2^n - 1$ product-terms in its two-level form. Even when a two-level form is reasonable for a given function, there are many cases where a multi-level representation can result in less area and a faster circuit. Especially for the gate-array and sea-of-gates design styles, the compact physical implementation provided by a PLA can not be exploited. Finally, two-level circuits are a special case of general multi-level circuits; hence, a logic synthesis system should provide tools which can select between two-level and multi-level implementations in order to trade-off the speed and area of the final design.

1.2.2 Optimum Multi-level Synthesis

Ashenhurst was the first to consider the problem of determining when a Boolean function has a nontrivial decomposition [7]. A *simple decomposition* of a function $f(x_1, \ldots, x_n)$ is a decomposition into the form $F(y_1, \ldots, y_s, \phi)$ and $\phi(z_1, \ldots, z_{n-s})$. A synthesis technique based on simple decomposition uses the heuristic that building F and ϕ will lead to an efficient implementation of f. The primary problem with this technique is that not all functions have a nontrivial simple decomposition. Even when a decomposition does exist, it is difficult to decide whether the decomposition will yield a simpler implementation of the logic function. Also, this simple approach fails to consider the important problem of determining subfunctions which are useful to realize multiple Boolean functions. Ashenhurst's techniques were later extended and generalized by Curtis [23] to handle other decomposition forms. However, the complexity of detecting decompositions, and the uncertain nature of the value of these decompositions, has limited the effectiveness of these techniques.

The first complete multi-level synthesis technique was by provided by Roth and Karp [58]. They proposed an algorithm to find the minimum solution to the multi-level logic synthesis problem. Their technique was a branch-and-bound algorithm based on a gener-

alization of Ashenhurst decomposition. Primitives in the implementation technology were considered as potential decomposition functions. The possible decompositions were ordered *alphabetically* and tried in turn at each step of the algorithm. Trivial lower bounds were used to bound the search through all possible Boolean graphs. A heuristic algorithm was proposed which would order the decompositions at each step by a measure of desirability, but no results are presented for the heuristic algorithm. A program was developed implementing their technique. The initial success of the formulation was immediately followed by the realization of the infeasibility of solving even small problems.[1]

Hellerman [40] provided a simple approach for optimum logic synthesis: enumerate all directed acyclic graphs and test each to see if it implements the desired function. His goal was to determine the optimum implementation for each function of three-variables. For each circuit graph with less than seven gates, the logic function was determined, and if the graph provided the best realization of the logic function, the solution was recorded. Twenty-five hours of computer time on an IBM 7090 were used to find the optimum NAND-gate networks for all three-variable functions. Because of the complexity of this technique (there are $O(2^{n^2})$ directed acyclic graphs of n nodes), Hellerman was unable to synthesize functions of more than seven gates.

Gimpel [32] proposed an optimum algorithm for designing three-level NAND-gate networks (also known as TANT-networks). Normal two-level minimization provides a special form of a TANT-network where the first level of gates is restricted to inverters which feed a cascade of NAND-gates to realize a sum-of-products form. A general TANT-network, in contrast, allows for arbitrary NAND-gates in the first level of gates, and arbitrary connections between the gates of the first, second, and third levels. Gimpel showed that a TANT-network can be written as a disjunction of permissible implicants, in analogy to traditional two-level minimization. A covering problem, called the *covering with closures problem*, was formulated using the permissible implicants where the minimum cover yields the optimum TANT network. Procedures to enumerate all permissible implicants and to solve the covering with closure problem are provided in his paper. A program was written to implement the synthesis technique, but no results are presented for problems of more than four variables. However, Gimpel claims to have improved on Hellerman's technique by creating the optimum TANT-network for the three-variable functions in only one minute.

[1] Karp has referenced the extreme computational complexity of these techniques as a motivation for his later work in complexity theory [47].

Davidson [27] provided an optimal NAND-gate network synthesis algorithm. His algorithm is similar to the algorithm of Roth and Karp, but uses only NAND-gates as the set of primitives. His approach was to synthesize the circuit from the output backwards. All possible partitions of the minterms between the terminals of a bounded fan-in NAND-gate at the circuit output are examined, and then the functions for each input to the NAND-gate are recursively synthesized in an optimum fashion. Bounding is possible once a best solution is known. Davidson comments that a six-function nine-variable problem was solved in three minutes, but that some single-function four-variable problems could not be solved optimally in ten minutes.

1.2.3 Modern Techniques

The techniques described in the previous section provide an optimum solution to the logic synthesis problem. However, none of these techniques have proven successful at logic optimization for designs with more than one hundred gates. The complexity of VLSI necessitates using approximate techniques to solve the optimization problem for large circuits.

One of the first modern developments is the Logic Synthesis System (LSS) from IBM [26,25]. The target technology for LSS is large gate array designs primarily in ECL. LSS focused on structuring a logic network using a rule-based approach. The technology-independent representation used was a graph of NAND-gates (or NOR-gates), and local transformations modified the graph into an optimal form.

The Yorktown-Silicon Compiler (YSC) [15] automatically synthesizes and lays out CMOS domino logic. The structuring and technology mapping phases of YSC are done with a collection of algorithms for solving a number of localized subproblems. YSC was the first system to separate technology-independent optimization from the technology-dependent operations. YSC also introduced the algebraic approximation which is extended and formalized in this thesis.

The Socrates system from GE [36] had a target design style of CMOS gate-array and standard-cell libraries. Socrates relied on work from the University of California, Berkeley for two-level minimization (ESPRESSO-MV) and work from the University of Colorado for multi-level structuring (WDIV). The contribution of Socrates was a rule-based approach for solving the technology mapping problem for CMOS gate-array and standard-cell libraries,

with a special emphasis on timing optimization.

1.3 Overview

Chapter 2 describes an exact algorithm for two-level minimization of a set of logic equations. Two-level minimization is an important step in the design of Programmable Logic Array's (PLA), but is also important as a technology-independent optimization for multiple-level logic optimization. The contribution of Chapter 2 is an exact algorithm for finding the minimum solution to the two-level minimization problem for two-valued and multiple-valued logic functions. A large percentage of the PLA optimization problems faced in the actual design of integrated circuits can be solved exactly using the techniques presented.

Chapter 3 presents new algorithms for solving the problem of finding common factors in a logic network. This is the primary technology-independent optimization step for logic synthesis. The techniques presented in Chapter 3 build on the algebraic approximation introduced by Brayton *et al.* The primary contribution is the unification of the algebraic decomposition techniques as an instance of the rectangle-covering problem. Efficient heuristics are proposed for solving the rectangle-covering problem.

Chapter 4 describes a new algorithm for solving the technology mapping problem. This is the primary technology-dependent optimization step for logic synthesis. The techniques in this chapter build on the work of Keutzer. The new techniques presented include an exact algorithm for solving the DAG-covering problem and extensions to the techniques to handle delay optimization.

In Chapter 5 a complete design is described in detail to demonstrate the use of logic synthesis in a realistic VLSI design. The design is an implementation of the Federal Government data encryption standard. It is entered at a register-transfer level, translated to a logic-level, optimized at the logic level, and then automatic placement and routing is used to finish the implementation.

Chapter 6 summarizes conclusions from this work, and suggests future areas for research in combinational logic synthesis.

The program MIS has been developed which implements the ideas presented in this thesis. In Appendix A, the software organization and goals of MIS are briefly described.

Chapter 2

Two-level Minimization

Two-level minimization remains an important problem in logic synthesis, both for optimization of Programmable Logic Arrays (PLA) and for multiple-level logic optimization. This chapter presents an exact algorithm for solving the two-level minimization problem for multiple-valued functions. Experimental results for an implementation of this algorithm show that many functions of more than twenty inputs are minimized exactly using these techniques.

2.1 Introduction

PLA's are an important design style for digital integrated circuits [30]. One important step in the automatic design of PLA's is the optimization performed at the logical level. Logic optimization of PLA's includes reducing the number of rows in the PLA (without changing the functions implemented by the PLA) as well as state-assignment, input-encoding, output-encoding, output phase assignment, and the use of multiple-bit input-decoders [59]. Each of these optimizations attempts to reduce the number of rows in the PLA, thereby improving both the area of the PLA, and the delay through the PLA. Two-level minimization is a fundamental step in each of these optimizations.

Two-level minimization is also important for multiple-level logic optimization. Local application of two-level minimization is an effective technique for reducing the complexity of a multiple-level logic network. In the multiple-level context, minimization of incompletely-specified functions is especially important – a don't-care set is constructed for a function in a multiple-level network which captures the environment of the function. The

function is simplified in two-level form with respect to this don't-care set [9].

Traditionally, research in two-level minimization concentrated on algorithms for exact solutions; that is, a cost function is defined for the algebraic representation of a logic function, and an algorithm is sought which provides a minimum-cost solution. Typically the cost function is to minimize either the total number of terms or the total number of literals required to write the set of equations. One problem with these exact algorithms is that they start from an enumeration of the minterms of the logic function, and hence are limited to relatively small problems. Even when the number of minterms (m) is manageable, the best known solutions to the covering problem have complexity $O(2^m)$. The net result was that exact two-level minimization of even simple functions, such as a four-bit multiplier with eight functions defined over eight inputs, had remained unsolved.

Recently, two-level minimization theory has been generalized to multiple-valued functions [65]. In particular, ESPRESSO-MV [59] is an extension of the *Espresso* algorithms to multiple-valued functions. The advantage of this generalization is that single-function minimization and multiple-function minimization are handled within the same framework. Multiple-valued functions also capture very naturally the minimization problems for PLA'S using input decoders [64], and the optimization problem of input-encoding for a symbolic variable [59].

An interesting out-growth of the work on ESPRESSO-MV was a new algorithm for exact minimization of multiple-valued functions [62,59]. The exact algorithm goes under the name ESPRESSO-EXACT because it borrows from the theory developed for the ESPRESSO-MV heuristic minimization program. The ESPRESSO-EXACT algorithm is similar to the Quine-McCluskey algorithm for two-level minimization, except that it has been updated to handle multiple-valued functions. However, the algorithm for each basic step is new. This set of new algorithms has greatly extended the ability of the algorithm to solve large problems. The advantages of the algorithm include a technique for detecting and eliminating from further consideration the essential prime implicants and the totally redundant prime implicants, and a fast technique for generating a reduced form of the prime implicant table. The minimum cover problem is solved with a branch and bound algorithm using the *maximal independent set heuristic* to control the selection of a branching variable and the bounding.

This chapter reviews the ESPRESSO-EXACT algorithm, and describes some new techniques which have been added to the algorithm to improve its performance. These enhancements include a faster algorithm for prime implicant generation, a sparse-matrix

representation for the prime implicant table, and the addition of a heuristic proposed by Gimpel [33] to reduce the prime implicant table without branching.

The new algorithm has been tested on the same collection of 134 minimization problems used for testing ESPRESSO-EXACT and is substantially faster than the previous version. More interestingly, the new algorithm has solved ten problems which the previous version was unable to solve. Many of the solved problems in the set have more than twenty inputs showing that the effective range of exact minimization has been extended for PLA optimization problems.

This chapter is organized as follows. First the basic definitions of multiple-valued functions are reviewed. The Quine-McCluskey minimization algorithm is then described. The details of ESPRESSO-EXACT are given next, including prime generation, prime implicant table generation, and derivation of a minimum cover of the prime implicant table. The chapter concludes with experimental results using this exact algorithm on a large collection of PLA'S.

2.2 Definitions

This section contains the basic definitions for multiple-valued functions and the two-level minimization problem. Only the most important definitions are included here. The interested reader is referred to [59] for more details.

Definition 2.2.1 *Let $p_i, i = 1, \ldots, n$ be positive integers. Define $P_i = \{0, \ldots, p_i - 1\}$ for $i = 1, \ldots, n$, and $B = \{0, 1, *\}$. A* multiple-valued input, binary-valued output function, *f, (hereafter known as a* multiple-valued function*) is a mapping*

$$f : P_1 \times P_2 \times \cdots \times P_n \rightarrow B$$

The function f has n multiple-valued inputs. Each input variable i assumes one of the p_i values in P_i.

Each element in the domain of the function is called a *minterm* of the function.

The value $* \in B$ represents a minterm for which the function value is unspecified (i.e., allowed to be either 0 or 1). Hence, functions are allowed to be incompletely specified.

An n-input, m-output switching function can be represented by a multiple-valued function of $n + 1$ variables where $p_i = 2$ for $i = 1, \ldots, n$, and $p_{n+1} = m$. This special case

is called a *multiple-output function*. It is easily proved that the minimization problem for multiple-output functions is equivalent to the minimization of a multiple-valued function of this form [65].

The *ON-set* of a function is the set of minterms for which the function value is 1. Likewise, the *OFF-set* is the set of minterms for which the function value is 0, and the *DC-set* is the set of minterms for which the function value is unspecified.

Definition 2.2.2 *Let X_i be a variable taking a value from the set P_i, and let S_i be a subset of P_i. $X_i^{S_i}$ represents the Boolean function*

$$X_i^{S_i} = \begin{cases} 0 & \text{if } X_i \notin S_i \\ 1 & \text{if } X_i \in S_i \end{cases}$$

$X_i^{S_i}$ is called a *literal* of variable X_i.

A *product term* is a Boolean product (AND) of literals. If a product term evaluates to 1 for a given minterm, the product term is said to *contain* the minterm.

A *sum-of-products* is a Boolean sum (OR) of product terms. If any product term in the sum-of-products evaluates to 1 for a given minterm, then the sum-of-products is said to contain the minterm.

A *cover* of a function is a sum-of-products which contains all of the minterms of the ON-set, and none of the minterms in the OFF-set. The cover optionally contains points of the DC-set.

The *cost* of a product term is a function mapping the set of all product terms onto the integers. The cost of a cover is the sum of the costs of the product terms in the cover.

The *two-level minimization problem* is to determine the minimum-cost cover of a multiple-valued function.

In the definitions which follow, $S = X_1^{S_1} X_2^{S_2} \cdots X_n^{S_n}$ and $T = X_1^{T_1} X_2^{T_2} \cdots X_n^{T_n}$ represent product terms.

The product term S *contains* the product term T ($T \subset S$) if $T_i \subset S_i$ for $i = 1 \ldots n$.

The *complement* of the literal $X_i^{S_i}$ (written $\overline{X_i^{S_i}}$) is the literal $X_i^{P_i - S_i}$.

The *complement* of the product term S ($\overline{S}$) is the sum-of-products $\bigcup_{i=1}^{n} \overline{X_i^{S_i}}$.

The *intersection* of product terms S and T ($S \cap T$) is the product term

$$X_1^{S_1 \cap T_1} X_2^{S_2 \cap T_2} \ldots X_n^{S_n \cap T_n}.$$

If $S_i \cap T_i = \emptyset$ for some i, then $S \cap T = \emptyset$ and S and T are said to be *disjoint*. The intersection of covers F and G is the union of $f \cap g$ for all $f \in F$ and $g \in G$.

The *distance* between S and T (*distance*(S,T)) is $|\{i|S_i \cap T_i = \emptyset\}|$.

The *consensus* of S and T (*consensus*(S,T)) is the sum-of-products

$$\bigcup_{i=1}^{n} X_1^{S_1 \cap T_1} \ldots X_i^{S_i \cup T_i} \ldots X_n^{S_n \cap T_n}.$$

If *distance*$(S,T) \geq 2$ then *consensus*$(S,T) = \emptyset$. If *distance*$(S,T) = 1$ and $S_i \cap T_i = \emptyset$, then *consensus*(S,T) is the single product term $X_1^{S_1 \cap T_1} \ldots X_i^{S_i \cup T_i} \ldots X_n^{S_n \cap T_n}$. If *distance*$(S,T) =$ 0 then *consensus*(S,T) is a cover of n terms. If the consensus of S and T is nonempty, it is the set of maximal product terms (ordered by containment) which are contained in $S \cup T$ and which contain minterms of both S and T. The consensus of two covers F and G is the union of *consensus*(f,g) for all $f \in F$ and $g \in G$.

The *cofactor* (or *cube restriction*) of S with respect to T (S_T) is empty if S and T are disjoint. Otherwise, the cofactor is the product term

$$X_1^{S_1 \cup \overline{T_1}} \ldots X_2^{S_2 \cup \overline{T_2}} \ldots X_n^{S_n \cup \overline{T_n}}.$$

The cofactor of a cover F with respect to a product term S is the union of f_S for all $f \in F$.

An *implicant* of a function is a product term which does not contain any minterm in the OFF-set of the function.

A *prime implicant* of a function is an implicant which is not contained by any other implicant of the function.

An *essential prime implicant* is a prime implicant which contains a minterm which is not covered by any other prime implicant.

The product term S can be represented in *positional cube notation* (also known as a *cube*) as a binary vector in the following form:

$$c_1^0 c_1^1 \ldots c_1^{p_1-1} - c_2^0 c_2^1 \ldots c_2^{p_2-1} - c_n^0 c_n^1 \ldots c_n^{p_n-1}$$

where $c_i^j = 0$ if $j \notin S_i$, and $c_j^i = 1$ if $j \in S_i$. The terms cube and product term are used interchangeably. For example, a prime cube is a cube which represents a prime implicant.

2.3 Exact Minimization Algorithms

In order to simplify the optimization problem, it is customary to place constraints on the form of the cost function for an implicant. Solving the two-level minimization problem for an arbitrary cost function is potentially difficult and not of practical interest.

Assumption 2.3.1 *A product term costs no more than any product term which it contains.*

If this assumption is satisfied, then it is easy to prove that a minimum solution exists which consists only of prime implicants. (Take any minimum solution, and replace each implicant with a prime implicant which contains each implicant; the resulting cover costs no more than the original cover, and hence is also a minimum solution.) Hence, the prime implicants can be used to form a minimum solution, rather than having to consider all implicants for the minimum solution.

Many useful cost functions satisfy this condition. For example, PLA optimization attempts to minimize the size of the PLA, which is reflected by assigning the same cost to each implicant. Another example is the cost function for single-output, binary-valued minimization used in multiple-level logic optimization. Here the goal is to minimize the number of literals needed in the Boolean equation; this is reflected by a cost which is the number of literals in the implicant which are not always 1. In both of these cases, a product term costs no more than any product term which it contains.

However, there are reasonable cost functions which do not satisfy this assumption. In a PLA, the process of adding a transistor to the output plane creates a product term which contains the original implicant when viewed as a multiple-valued minimization problem. This product term costs more than the implicant which it contains. Hence, the cost function for minimizing the total number of transistors in a PLA violates Assumption 2.3.1. Specifically, it may be possible to remove a transistor from the output part of a prime implicant without further expanding the implicant in its input part; all such nonprime implicants for all combinations of output transistors must be considered as candidates for a minimum solution to the minimum-literal optimization problem. Fortunately, the number of rows in the PLA is the most important optimization criteria for a PLA; reducing the number of transistors in the PLA is only a secondary optimization goal. Hereafter, we assume that Assumption 2.3.1 is satisfied.

The Quine-McCluskey algorithm to derive a minimum cover for a function consists of the following steps:

1. Generate all of the *prime implicants.*
2. Form the *prime implicant table.*
3. Derive a minimum cover of this table.

Many different algorithms have been proposed for solving each of these steps. The algorithm presented by McCluskey generates the prime implicants starting from a list of minterms using consensus. The prime implicant table is constructed with a single row for each minterm and a single column for each prime implicant. For each minterm row, a 1 is placed in a column if the corresponding prime implicant contains the minterm. The problem of selecting a minimum subset of primes is thus mapped into the problem of selecting a minimum cover of this matrix. A cover of this matrix is a row vector of 0's and 1's such that each row of the matrix shares a 1 in some column with the row vector, and a minimum cover is one with the fewest number of 1's. Petrick's technique for solving the covering problem converts a product-of-sums representation of the covering problem into a sum-of-products expression. Each resulting product term represents a possible solution, and each corresponding cover is evaluated against the cost function to find the minimum cost cover.

Better algorithms for each of these steps have been proposed. There exist many techniques for generating all of the prime implicants of a function without starting from an enumeration of minterms of the function. However, generating the prime implicant table remains a problem because an enumeration of minterms is required. Algorithms exist which solve the minimization problem without creating the prime implicant table [24]; however, these algorithms have difficulty developing heuristics to guide the selection of a minimum set of prime implicants. Algorithms for the minimum-cover problem rely on a branch and bound search among the feasible solutions, which is more efficient than Petrick's technique. Techniques are used in these algorithms to guide the search toward good solutions quickly and to trim the search space by deriving lower bounds on the remaining subproblem.

There are many potential problems with an exact minimization algorithm. If the algorithm requires an explicit enumeration of the minterms at any step, then minimization of large functions (e.g., more than thirty variables) is not feasible. Even if the prime implicants are derived without enumerating minterms, there exist functions with an exponential

number of prime implicants as a function of the number of implicants in a minimum cover [53]. Hence, there will always be functions for which the enumeration of all of the prime implicants is infeasible. Finally, the minimum-cover problem is NP-complete [31] implying that no efficient algorithm is known to solve this optimization problem. The size of the minimum-cover problem is related to the number of prime implicants; hence, even when it is feasible to enumerate all prime implicants, it may not be feasible to derive a minimum cover for the prime implicant table.

The hope for exact minimization algorithms is that the problems faced in practice do not exhibit the worst case behavior. An exact minimization algorithm should not *a priori* disallow functions of thirty variables just because *some* thirty variable functions cannot be minimized. Interestingly, as is shown in Section 2.9, experimental results indicate that a large percentage of PLA optimization problems taken from integrated circuit designs do not exhibit an exponential worst-case behavior. The fact that many large PLA optimization problems can be solved exactly indicates that PLA'S, as designed for actual circuits, are quite special – they have characteristics much different from random functions of the same number of variables. Often they do not have a large number of prime implicants, and they generate prime implicant tables which are sparse and easy to solve.

Given the existence of effective heuristic minimizers and the fact that there will always be practical minimization problems which cannot be solved exactly, a good question is, "Why is exact minimization of interest ?". First, there is the theoretical interest of determining how far exact algorithms can be pushed while solving fundamentally difficult problems. Second, and more important, is that the result from an exact minimization is the best indicator of the quality of a heuristic minimization algorithm. Comparisons of modern heuristic algorithms against exact solutions have shown that minimization algorithms such as ESPRESSO-MV provide solutions which average within one percent of the minimum solution [59]. Of course, the performance of the heuristic minimizer is known only for those problems which can be solved exactly.

2.4 ESPRESSO-EXACT

ESPRESSO-EXACT starts with a cover for the ON-set F and the don't-care set D of a multiple-valued function. The algorithm proceeds as follows:

1. Generate all prime implicants P of the function $F \cup D$.

2. Partition P into the essential primes (E), the totally redundant primes (R_t), and the partially redundant primes (R_p).
3. Create a reduced prime implicant table (A) from R_p.
4. Find a minimum cover for A.
5. Select the primes in the cover for the solution.

These steps are covered in the subsequent sections.

2.5 Prime Generation Algorithm

Two techniques are presented here for prime generation for multiple-valued functions.

The first is based on the unate recursive paradigm and the operation of consensus. The unate recursive paradigm was introduced in [19] and extended to multiple-valued functions in [59]. This algorithm starts with the ON-set and DC-set of the logic function to generate the prime implicants.

The second algorithm is related to the *blocking-matrix* used in Espresso to guide the selection of a prime implicant during the EXPAND operation [19]. This algorithm starts from the OFF-set of the logic function and forms a logic function describing the characteristics of a prime for the logic function.

2.5.1 Prime Generation using Consensus

A function f can be decomposed according to its Generalized Shannon Expansion [65] as:

$$f = lf_l + rf_r$$

where $l \cap r \neq 0$ and $l \cup r = 1$.

A prime which contains minterms in both lf_l and rf_r must be formed from the consensus of a cube $c_1 \in lf_l$ and a cube $c_2 \in rf_r$. Therefore, the set of primes for f is contained in the union of the primes of lf_l, the primes of rf_r, and the product terms resulting from the consensus of the primes of lf_l and the primes of rf_r. Not all of these product terms are prime, so it is necessary to perform single-cube containment on this set to derive the set of primes for f (that is, delete any cube contained in another cube in the cover). The primes of lf_l (rf_r) are the primes of f_l (f_r) intersected with l (r).

This leads to a recursive algorithm for generating the primes of a function. The set of primes for each of the cofactors f_l and f_r is computed recursively, and then the results are merged to generate the primes for f. The recursion ends when the function is a single product term, for which the set of primes is merely the product term.

This is the basic structure for the unate recursive paradigm [19]. Rather than cofactoring the function until only a single cube remains, a stronger condition can be used to end the recursion. If a cover of a function f is *strongly-unate* [59] then the set of prime implicants for the function can be derived by performing single-cube containment on the cover. Hence, in this case, it is possible to identify all prime implicants by inspection and terminate the recursion immediately.

2.5.2 Prime Generation from the Off-Set

Let R be a cover for the off-set of the function. If the OFF-set of the function is not available, it may be computed using a fast multiple-valued complementation algorithm [63,59] starting with the function $F \cup D$.

In order for a cube c to be an implicant of F, c must not intersect each cube $r^i \in R$. This can be expressed by writing a Boolean expression. Let c_j^k be a Boolean variable representing the condition that part k of variable j of cube c be set to 1. Let $(r^i)_j^k$ have the value of 1 if part k of variable j of the cube r^i is a 1. The following Boolean expression asserts that c does not intersect the off-set of the function:

$$I = \bigcap_{i=1}^{|R|} \bigcup_{j=1}^{n} \bigcap_{k=0}^{p_{j-1}} \left(\overline{(r^i)_j^k} + \overline{c_j^k} \right)$$

Note that the values for each cube r^i (written as $(r^i)_j^k$) are known values of either 0 or 1, and that the variables in the above equation are c_j^k.

To form a sum-of-products representation of I requires that the product-of-sums-of-products expression be *multiplied-out*; that is, repeated intersection of sums-of-products covers. However, using DeMorgan's law, it is possible to directly write an expression for $\overline{I}$:

$$\overline{I} = \bigcup_{i=1}^{|R|} \bigcap_{j=1}^{n} \bigcup_{k=0}^{p_{j-1}} \left((r^i)_j^k c_j^k \right)$$

An implicant of the function I corresponds to an assignment of $\{0, 1\}$ to the variables c_j^k which results in an implicant of f. Further, a prime implicant of I corresponds

to an assignment of {0, 1} to the variables c_j^k which is maximal in the sense that no other variable which is 0 can be made a 1; therefore, a prime implicant of I corresponds to a prime implicant of f. By construction, the logic function I is a two-valued unate logic function. Hence, any prime cover for I consists of all of the prime implicants for I [19, Prop 3.3.7]

This construction proposes two techniques for generating all of the prime implicants of a function: one which involves repeated intersection of sum-of-products forms and one which involves the complementation of a sum-of-products form. The first formulation is equivalent to the technique outlined by Roth [57, Chapter 1] for generating all of the prime implicants of a multiple-output logic function.

2.5.3 Comparison of Prime Generation Techniques

These prime generation techniques have been implemented as part of ESPRESSO and compared for efficiency. The OFF-set algorithm uses repeated intersection to generate the primes. The efficiency of the two implementations is similar so that the results are directly comparable.

The comparison was performed with the 134 functions from the Berkeley PLA Test Suite (see Section 2.9 for more information on this test set). Each algorithm was given ten hours on a DEC MicroVax-II to compute all prime implicants for each function.

Using the OFF-set prime generation, the prime implicants were found for 113 examples. Using the unate-recursive paradigm, the prime implicants were found for 118 examples. In no case was the OFF-set algorithm able to complete for an example where the unate-recursive paradigm failed. For the problems both were able to solve, the total time for the OFF-set algorithm was 31.3 hours, and the total time for the unate recursive paradigm was 14.5 hours. There was a wide range in the run-time between the two algorithms; the ratio of the OFF-set algorithm run-time to the unate-recursive algorithm run-time ranged from .65 to 121. For 8 examples the OFF-set algorithm was faster. The unate recursive paradigm was substantially faster for those problems which had a small ON-set and a large OFF-set.

For this benchmark set, the unate-recursive paradigm technique for prime generation is favored both in total run-time, and the ability to complete more examples. Therefore, this is the algorithm used by ESPRESSO-EXACT.

2.6 Essential and Partially Redundant Primes

The primes P are partitioned into the essential set E, the totally redundant set R_t, and the partially redundant set R_p according to the following rules:

$$E = \{c \in P | c \not\subseteq (P - c)\}$$
$$R_t = \{c \in (P - E) | c \subseteq (E \cup D)\}$$
$$R_p = P - (E \cup R_t)$$

The cubes of E must belong to any cover of the function because they cover some minterm not covered by any other prime. E is the set of essential prime implicants. No cube of R_t can belong to a minimum cover of F because it is contained by the set of essential prime implicants. R_t is the set of prime implicants dominated by the essential prime implicants. The cubes of R_p are partially redundant because, although any single cube of R_p can be removed, it is not possible to simultaneously remove all of the cubes of R_p while maintaining a cover of F. R_p causes the most difficulty in trying to extract a minimum subset of P.

The separation of P into the covers E, R_t, and R_p is accomplished with a fast multiple-valued tautology algorithm [66,59]. The basic test $c \subseteq H$ is done by forming the cofactor H_c, and testing if H_c is a tautology (i.e., if the function evaluates to 1 for all inputs). The fast tautology algorithms use the Generalized Shannon Cofactor to successively decompose the tautology question for a function into a tautology question on each of its cofactors. The recursion ends when the function is such that the tautology question can be answered by inspection. In particular, when the function becomes *weakly-unate* [59], it is possible to answer the tautology question by inspection.

2.7 The Reduced Prime Implicant Table

The technique for forming the reduced prime implicant table is now described. The key to the algorithm is a simple modification of the multiple-valued tautology algorithm. Rather than testing whether the function is a tautology, the subsets of cubes which would have to be removed to prevent the function from becoming a tautology are enumerated.

For each cube $c \in R_p$, form $H = E \cup R_p - c$ and use a multiple-valued tautology algorithm to determine if H_c is a tautology. By definition of E and R_p, H_c must be a

tautology because every cube of R_p is covered by the union of E and the other cubes of R_p. At each leaf in the tautology algorithm where the cover is weakly-unate, it is trivial to determine which cubes are required to make the function a tautology [59]. A cube in the partial function contributes to make the function a tautology if, and only if, it covers all of the minterms in this subspace.

Therefore, if a cube from E or D covers all of the minterms in this subspace, then no cubes of R_p are needed to cover this part of the function. That is, this leaf is a tautology independent of the cubes of R_p which are discarded. Otherwise, all of the cubes of R_p which cover all of the minterms in this subspace must be removed in order to avoid $H_c = 1$ in this leaf. This is equivalent to saying that H will fail to cover c if and only if all of the cubes of R_p which are the universe in this leaf are discarded. In this way, all of the subsets of R_p which *fail* to cover the original function are enumerated.

A {0,1} matrix is formed where each cube of R_p is associated with a column. At each leaf in the tautology algorithm where no cube from E is the universal cube, a row is added to the matrix with a 1 in each column j where a universal cube in this leaf came from $(R_p)^j$. If any prime in this row is retained in the cover, then H_c will be tautology in this leaf; if no primes in this row are selected, then H_c will not be a tautology in this leaf. Hence the selected set of primes will fail to cover some minterm of the original function if no prime in this row is selected. A minimal cover of this matrix corresponds to a minimal subset of the primes of R_p which must be retained in the cover for F.

The algorithm proceeds by forming H_c for each $c \in R_p$, and calling a modified version of the TAUTOLOGY procedure called FIND_TAUTOLOGY. Note that after determining how c can be covered, c can be moved to D because it is then known how all of the minterms of c can be covered by selecting primes from R_p. This leads to an improvement in the performance of the algorithm.

The matrix formed in this way is related to the prime implicant table of the Quine-McCluskey algorithm. This matrix is a reduced form of the prime implicant table; rather than each row of the matrix corresponding to a minterm of the function, each row corresponds to a collection of minterms (i.e., a larger subspace) all of which are covered by the same set of prime implicants. In the worst case, the tautology algorithm will terminate at each of the minterms of the function, thus producing exactly the prime implicant table. However, in practice, the algorithm is terminated much more quickly, leading to efficient creation of a reduced prime implicant table. A key to terminating the recursion as quickly

as possible makes use of *weakly-unate* functions.

2.8 Solving the Minimum Cover Problem

The minimum cover problem is stated as follows:

> **Minimum Covering Problem:** Given a binary matrix A, and a cost c_j for each column of the matrix, find a binary row vector x such that $A \cdot \mathbf{x}^T \geq (1, 1, \ldots 1)^T$ and $\sum_{j=1}^{m} x_j c_j$ is minimum.

The constraint $A \cdot \mathbf{x}^T \geq (1, 1, \ldots 1)^T$ is understood as saying that each row of the matrix must have at least one 1 in some column where x has a 1. In this case, the row is said to be *covered* by the particular column of x, and the goal is to cover all rows with a vector of minimum weight.

A minimum cover problem can also be represented as a *covering expression*. This is a Boolean expression written in conjunctive normal form. Each column of the covering matrix A has an associated Boolean variable a_i. Each row represents a clause corresponding to the disjunction of the variables for the nonzero elements in the row. A satisfying assignment (i.e., an assignment of 0 or 1 to the variables a_i for which the expression evaluates to 1) is a cover for the matrix, and the problem is to find the cover with lowest total cost.

Example 2.8.1 *Consider the matrix:*

a_1	a_2	a_3	a_4	a_5	a_6
1	1	0	0	0	0
0	1	1	0	1	0
1	0	1	0	1	1
0	0	1	1	0	1

The corresponding covering expression is

$$(a_1 + a_2)(a_2 + a_3 + a_5)(a_1 + a_3 + a_5 + a_6)(a_3 + a_4 + a_6)$$

A cover for this matrix is x = 1 1 0 1 0 0 *which corresponds to the satisfying assignment* $a_1 = 1$, $a_2 = 1$, $a_4 = 1$ *with all other variables 0. If the cost for each column is* 1, *then a minimum cover is* x = 1 0 1 0 0 0 *which corresponds to the satisfying assignment* $a_1 = 1$, $a_3 = 1$ *with all other variables 0.*

The decision problem for minimum cover is NP-complete [31, page 222] so that any algorithm which solves this problem can be expected to have a bad worst-case complexity.

The standard branch-and-bound solution to the minimum cover problem involves the following steps:

1. Apply reduction algorithms to reduce the size of the matrix.
2. If the size of the current solution equals or exceeds a bound (e.g., the size of the best solution seen so far) return from this level of the recursion. If there are no elements left to be covered, declare the current solution as the best solution recorded so far.
3. Select a branching column.
4. Add the branching column to the selected set and solve the covering problem for the submatrix resulting from deleting this column and all rows which are covered by this column. Then, solve the covering problem for the submatrix resulting from deleting this column without adding it to the selected set.

2.8.1 Covering Table Reduction Steps

There are some well-known results which are of interest in reducing the size of a given covering problem:

Partitioning: If the rows and columns of matrix A can be permuted to yield a block structure of the form:

A_1	0
0	A_2

where 0 represents an appropriately sized block of all zeros, then a minimum cover for A can be written as the union of a minimum cover for A_1, and a minimum cover for A_2. Detecting a block partition of this form is easily done by choosing an initial row and placing it in the partition for A_1. Then all rows which are connected to this row (i.e., all rows which have a 1 in some column of A_1) are added to the partition for A_1. This continues until no rows remain, in which case no partition exists, or until all of the remaining rows are disjoint from the set of rows in A_1.

Essential columns: Any row of the matrix A which has only a single 1 identifies an essential column. The solution vector x must have a 1 in the essential column in order to cover the row singleton. After placing a 1 in the essential column, any other rows which become covered are removed from consideration.

Row dominance: If row i of A contains row j of A (i.e., row i contains a 1 for all columns in which row j has a 1), then row i can be removed from the matrix A without changing the minimum solution. Clearly, once row j has been covered, then row i will

automatically also be covered, and hence row i is providing redundant information in the covering problem.

Column dominance: If column i of A contains column j of A (i.e., column i contains a 1 for all rows in which column j contains a 1), then column j can be removed from the matrix A without changing the minimum solution. Clearly, there could be no advantage to choosing column j because choosing column i instead would cover the same set of rows, and perhaps more. Hence, column j is not needed for a minimum solution. When a cost is associated with a column, this reduction can be performed only if column j costs the same or more than column i.

Note that, by construction, the reduced prime implicant table does not have any essential elements. This is because the essential primes are detected before the prime implicant table is created. However, during the branch and bound procedure, essential elements can be created. The operations of row and column dominance, which possibly delete rows and/or columns, also have the possibility of introducing essential columns.

Therefore, the strategy to reduce the size of the matrix is:

1. Look for a block partition. Recur for each subproblem if a block partition exists.
2. Apply row dominance and column dominance to the matrix.
3. Identify essential columns and add them to the covering set; delete rows which are covered by these essential columns.
4. Repeat steps 2 and 3 until no new essential columns are detected.

2.8.2 Gimpel's Reduction Step

Another heuristic for solving the minimum cover problem which has proven effective is one suggested by Gimpel [33]. Gimpel proposed a reduction step which simplifies the covering matrix when it has a special form. This simplification is possible without further branching, and hence is useful at each step of the branch and bound algorithm. In practice, Gimpel's reduction step is applied after reducing the covering matrix to minimal form, that is, after applying the reduction steps of removing essential columns, and deleting dominated rows and columns.

Gimpel's reduction is best described in terms of the covering expression for a covering table. The covering expression is examined to see if any clause has only two

literals of the same cost. For example, assume the expression has the form:

$$p = R(c_1 + c_2)(c_1 + S_1)\ldots(c_1 + S_n)(c_2 + T_1)\ldots(c_2 + T_m)$$

where c_1 and c_2 are single variables with a cost $c*$, $S_i, i = 1\ldots n$ and $T_j, j = 1\ldots m$ are sums of variables not containing c_1 or c_2, and R is a product of sums of variables not containing c_1 or c_2. Because the covering table is assumed minimal, if there is a clause $(c_1 + c_2)$, then $m \geq 1$, $n \geq 1$, and none of S_i or T_j is identically zero.

Note that with the expression written in this form, each parenthesized expression corresponds directly to a single row in the covering table. By applying Boolean algebra manipulations to this expression, it can be re-written as:

$$p = R(c_1c_2 + c_1T + c_2S)$$

where $S = \prod_{i=1}^{n} S_i$, and $T = \prod_{i=1}^{m} T_i$.

A second covering problem is derived from the original covering problem with the following form:

$$\begin{aligned} p_1 &= R(c_2 + S + T) \\ &= R\prod_{i=1}^{n}\prod_{j=1}^{m}(c_2 + S_i + T_j) \end{aligned}$$

The main theorem of Gimpel is:

Theorem 2.8.1 *Let M_1 be a minimum cover for p_1. A cover for p can be derived from M_1 according to the rule:* if S is covered by M_1 then add c_2 to M_1 to derive a cover of p; otherwise, add c_1 to M_1 to derive a cover of p. *The resulting cover is a minimum cover for p.*

Proof. Let $|p|$ represent the cost of a minimum cover for p. The rule stated in the theorem derives a cover for p by adding either c_1 or c_2 to the cover for p_1. It is easy to see that the set of variables created by this rule generates a cover for p, and hence $|p| \leq |p_1| + c*$. Next, let M be a minimum cover of p. If c_1 is in M, remove it to create $\tilde{M_1}$; otherwise, remove c_2 to create $\tilde{M_1}$. (Either c_1 or c_2 must be in M.) The variable set $\tilde{M_1}$ is a cover for p_1 as can be verified by examining the covering expressions for p and p_1; hence, $|p| - c* \geq |p_1|$. Therefore, $|p| = |p_1| + c*$ and the cover is a minimum cover for p. □

Note that with this reduction the expression $c_2 + S + T$ is not a simple sum of variables; it is first expanded into a product of sums of products as shown above. The resulting covering expression has $nm - n - m - 1$ more clauses, but depends on one less variable. When $n < 2$ or $m < 2$ or $n = m = 2$, this leads to a covering problem with fewer clauses. The reduction technique is clearly beneficial when both the number of clauses and variables in the covering matrix is reduced. However, it is potentially advantageous to allow the number of clauses to increase by applying the reduction step. This is because the number of variables (and hence the number of potential branching columns) is reduced by one. Also, the covering table formed by this reduction step may be further reduced by applying the reduction steps described in Section 2.8.1.

An important special case is worth noting. If n is 1 and the cost of c_2 is greater than or equal to the cost of any element of S_1, then the covering expression simplifies to

$$\begin{aligned} p_1 &= R \prod_{j=1}^{m} (c_2 + S_1 + T_j) \\ &= R \prod_{j=1}^{m} (S_1 + T_j). \end{aligned}$$

The second equality follows from the observation that c_2 is dominated by every column of S_1. Therefore, variable c_2 is also deleted from the covering expression. The resulting expression has one fewer clause than the original covering expression and it depends on two fewer variables.

Gimpel refers to the general reduction step as identifying a *reducing column of the second kind*, and the special case as identifying a *reducing column of the first kind.* Gimpel's reduction step was originally stated for covering problems where each column had cost 1. Robinson and House [44] showed that the reduction remains valid even for weighted covering problems if the cost of the column c_1 equals the cost of the column c_2. This is the form presented here.

2.8.3 Maximal Independent Set

An important feature of the proposed covering algorithm is the use of the maximal independent set. This routine finds a maximal set of rows of A all of which are pairwise disjoint (i.e., they do not have 1's in the same column). It is clear that the number of rows in this independent set is a lower bound on the solution to the covering problem, because

a different element must be selected from each of the independent rows in order to cover these rows. Hence, this lower bound can be used to terminate the search if the size of the current solution plus the size of the independent set is greater than or equal to the best solution seen so far. Additionally, the size of the independent set at the first level of the recursion is a lower bound for the final minimum cover. Hence, by recording this value, the search can be terminated if a solution is found which meets this lower bound,

The major drawback of this technique is that the problem of finding a maximum independent set of rows is itself an NP-complete problem. But this is not a limitation – finding a maximal independent set of rows can be solved heuristically while still providing a correct lower bound on the size of the final solution. In general, finding the maximum independent set provides the best bound; other minimal solutions provide less precise, but, nonetheless, correct lower bounds. Hence, even though this problem is itself difficult, a fast heuristic algorithm for finding a maximal independent set of rows is sufficient for this application.

To find a large independent set of rows, a graph is constructed where the nodes correspond to rows in the matrix, and an edge is placed between two nodes if the two rows are disjoint. The problem is now equivalent to finding a maximal clique (a maximal, completely connected subgraph) of this graph. To solve this problem, a greedy algorithm is used:

1. Initialize the clique to be empty.
2. Pick the node of largest degree (and not already in the current clique), and add this node to the clique. Break ties by choosing the node which is connected to the most other nodes of maximum degree.
3. Remove all nodes and their edges from the graph which are not connected to the current clique.
4. Repeat if there are nodes in the graph not in the current clique.

The node of largest degree in step 2 corresponds to the row which is disjoint with the maximum number of other rows of the matrix. The tie-breaker attempts to preserve as many of the remaining nodes of maximum degree as possible.

Thus, the bounding in the branch and bound algorithm is modified by bounding the search if the size of the maximal-independent set plus the size of the current partial solution equals or exceeds the best known solution. The goal is to terminate unprofitable searches as early as possible.

Beside the fact that the problem of finding a maximum independent set of rows is NP-complete, there is the further difficulty that the bound provided by the maximum independent set may not be sharp. For example, consider the matrix:

$$\begin{matrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{matrix}$$

A maximum independent set of rows for this matrix contains only a single row, but a minimum cover requires at least two columns. The size of the maximum independent set remains a lower bound on the size of a minimum cover; however, the search may not be terminated as early as possible.

2.8.4 Choice of the Branching Column

Good heuristics for choosing the branching column are important to the speed of the branch and bound algorithm. The goal is to find a good solution quickly, so that inferior parts of the search space may be discarded as early as possible.

To choose a branching column, a weight W_j is computed for a each column j as:

$$W_j = \sum_{i=1}^{m} A_{ij} w_i$$

where

$$w_i = \left[\left(\sum_{j=1}^{n} A_{ij} \right) - 1 \right]^{-1}.$$

It is assumed each row has two or more elements, hence w_i is well-defined. Note that if w_i were 1, then W_j would be the cardinality of each column. Choosing an element which intersects a large number of rows is reasonable since these rows are removed when this element is selected.

To understand the effect of w_i as defined here, assume that a row has only two elements. Then each element contributes $w_i = 1$ to the respective column weights. On the other hand, if a row has nine elements, then each element in the row contributes $w_i = .125$ to each column where it has a 1. This has the tendency to favor columns with a large number of 1's, but also favors columns with a large number of 1's in rows with a few elements. The larger rows are thought of as *easier* to cover while the smaller rows are *harder* to cover. The heuristic tries to force a selection from one of the harder to cover rows. Choosing an

element from a small set also creates more essential elements in subsequent levels of the recursion.

A unique element from each set of the independent set of rows must be in the minimum solution. This suggests limiting the selection of a branching column to the elements in these rows. Hence, as a final refinement, the column of maximum weight W_j which is also in some element of the maximal independent set is chosen as the branching column.

2.8.5 Implementation Details

It is known that given an arbitrary binary matrix, there is a Boolean function which creates that matrix as its prime implicant table [33]. This is the basis of the proof that the two-level minimization problem is NP-complete when starting from all prime implicants for the function. However, in practice, the prime implicant table tends to be very sparse. This influences the choice of data structure for the prime implicant table. Two common data structures for representing a binary matrix are the *bit-matrix* and the *sparse-matrix*.

A bit-matrix uses one bit to store each element of the matrix. Thirty-two adjacent columns in the matrix are packed into a single thirty-two bit machine word. This allows for a constant-factor improvement (when operating on the elements of a row in the matrix) over the more straightforward approach which uses one machine word for each element in the matrix. For example, comparing two rows for containment is thirty-two times faster because a few machine instructions suffice to compare thirty-two adjacent columns for containment.

A sparse-matrix, on the other hand, only stores the nonzero elements in the matrix. A structure is used for each nonzero element, and it is linked to both the next and previous row in the same column, and the next and previous column in the same row. This allows for efficient traversal of only the nonzero elements in the matrix. For example, in an n by m matrix with at most d elements per row, only d elements need to be examined to determine if one row contains another. This is in contrast to the $O(n)$ comparisons needed when using a bit-matrix.

An important factor in deciding which data structure is best is the choice of the most efficient algorithm for each data structure. Consider the problem of detecting row dominance in the matrix and deleting all rows which contain another row. Assume that the matrix is square and of size n by n. Further, assume that there are at most d nonzero elements in any row or column.

On the bit-matrix, a straightforward row dominance algorithm is the most efficient. This algorithm compares each row against all other rows and deletes the row if it contains another row. This algorithm requires $O(n^2)$ row comparisons, each of which has complexity $O(n)$. Using the bit-matrix data structure allows for an efficient check to see if one row contains another – only four machine instructions are needed to compare two 32-bit words to see if one contains the other (including loop overhead). Therefore, the machine instruction complexity for the bit-matrix implementation of the straightforward algorithm is estimated as $\frac{1}{8}n^3$.

However, if the matrix is sparse and a sparse-matrix is used as the data structure, there is a more efficient algorithm for detecting row dominance. For each row of the matrix, consider the columns which have a 1, and select the column with the fewest total number of 1's. The 1's in this column identify the set of rows which can possibly contain the original row; a row outside this set cannot contain the original row because it fails to contain at least this column. This algorithm requires only $O(nd)$ row comparisons rather than $O(n^2)$. Each row comparison requires examining the $O(d)$ elements in the two rows. However, the basic operation of comparing two rows in a sparse matrix to see if one contains another is more complex in terms of machine instructions. Examining compiler-generated code indicates that approximately ten machine instructions (including loop overhead) are needed for each element in the row. As a result, the machine instruction complexity of the sparse matrix implementation is $10nd^2$.

Therefore, the bit-matrix implementation is superior when $d/n > .11$; that is, when the matrix is more than 11% dense. If the matrix is less than 11% dense, the sparse-matrix implementation is superior. As the matrix becomes more sparse, the sparse matrix implementation begins to look much better. For example, the modified row dominance algorithm on a sparse-matrix with one percent nonzero elements (i.e., $d/n = .01$) is eighty times faster than the straightforward algorithm on a bit-matrix. In the limit of a constant number of nonzero elements per row, the complexity has been reduced from $O(n^3)$ to $O(n)$.

Therefore, a sparse-matrix implementation of the basic operations of row dominance and column dominance is expected to be superior to the same operations implemented using a bit-matrix if the prime implicant table is sufficiently sparse. Other operations, such as finding a block partition if one exists or finding a maximal independent set of rows, also benefit in a similar manner from the sparsity of the matrix.

Density of the Prime Implicant Table

The density of the prime implicant table is defined as the number of nonzero elements divided by the product of the number of rows and columns in the table.

A test was performed to measure the density of the prime implicant table over a collection of PLA minimization problems. The comparison was performed with the 134 functions from the Berkeley PLA Test Suite (see Section 2.9 for more information on this test set). The prime implicant table was generated for 117 of the 134 examples. The average density over all of the examples was .37%. Considering only tables with more than 100 rows, the density of the table ranges from .07% to 8.35%. Hence, this supports the conjecture that the prime implicant tables are sparse.

As the size of the table increases, the maximum density tends to decrease significantly. For this reason, the sparse-matrix implementation for the covering algorithm provides a significant advantage for larger problems. For example, the largest prime implicant table was 4,640 rows by 5,202 columns, but only .07% dense. Performing the operations of row and column reduction for this matrix required only 92 seconds on a DEC MicroVax-II when a sparse-matrix data structure was used. The same operations did not complete in ten hours using the bit-matrix implementation of ESPRESSO Version 2.2.

2.9 Experimental Results

The techniques outlined in this chapter for exact minimization of multiple-valued functions have been implemented as an option to the program ESPRESSO. The algorithm ESPRESSO-EXACT is the *exact* option to Version 2.3 of the ESPRESSO program. Version 2.3 improves upon Version 2.2 by the new techniques described in this chapter; namely, prime generation based on the unate recursive paradigm, and the use of sparse matrices and Gimpel's reduction of the first kind when solving the minimum-cover problem,

ESPRESSO-EXACT has been tested on a large set of multiple-output minimization problems. Each minimization problem is first classified as to its degree of difficulty. Then results are presented on the performance of ESPRESSO-EXACT when attempting to solve these problems and the results from ESPRESSO-EXACT are compared to the previous version of ESPRESSO-EXACT and the exact minimization program MCBOOLE [24].

The results and run-times in this chapter were collected on a DEC MicroVax-II.

This machine is roughly equivalent to a DEC VAX 11/780 for integer applications such as ESPRESSO. The DEC VAX 11/780 is often called a one MIP machine. The machine was equipped with 8 megabytes of memory.

2.9.1 Classification of the Benchmark Set

In order to compare minimization algorithms, a set of 134 PLA'S have been collected at Berkeley. Most of these PLA'S (111) come from industry and University chip designs. The remaining 23 PLA'S are mathematical functions which have commonly been used as standard minimization problems. The characteristics of the PLA'S, including the number of inputs, outputs, product terms and presence of a don't-care set, are listed in a table at the end of the chapter.

One easily determined measure of the complexity of two-level minimization for a function is the number of minterms in the function. A single-output n-input function has 2^n minterms and an n-input and m-output function has $m2^n$ minterms. Hence, it is reasonable to consider minimization of an n-input m-output function equal in complexity to the minimization of a $n + \log_2(m)$ single-output problem. This provides a simple measure of the complexity of the different multiple-output minimization problems.

By this measure, 14 (10%) of the examples have more than thirty inputs, 31 (23%) have more than twenty inputs, and 95 (71%) have more than ten inputs. Hence, a large percentage of the examples would be considered unsolvable by exact methods according to the rule of thumb given in [19][page 8]:

> Since the number of elements in the covering problem may be proportional to the exponential of the number of input variables of the logic function, the use of these techniques is totally impractical even for medium sized problems (10-15 variables).

However, this simple metric, or other measures such as the number of product terms or prime implicants (when known) can provide a misleading measure of the complexity of two-level minimization for a specific example. In order to circumvent this problem, each function has been classified as either *trivial*, *noncyclic*, *cyclic-solved*, *cyclic-unsolved*, or *too many primes*. These classifications were determined by allowing ESPRESSO-EXACT to run for up to ten hours for each example. The result of the minimization is examined to determine the classification, as follows:

Class	Total	Solved
trivial	9	9
noncyclic	56	56
cyclic-solved	49	49
cyclic-unsolved	3	0
too many primes	17	0
Totals	134	114

Table 2.1: ESPRESSO-EXACT results for the PLA test set.

trivial A solved problem is *trivial* if all primes in the minimum cover are essential. These are the easiest problems to solve because any minimal solution is the minimum solution.

noncyclic A solved problem is *noncyclic* if the prime implicant table has no rows in its reduced form. For these problems, the covering problem can be solved in polynomial time even though generating the prime implicants or forming the prime implicant table is still potentially difficult.

cyclic A solved problem is *cyclic* if the prime implicant table has more than one row in its reduced form. These problems require a branch and bound algorithm to derive a minimum cover for the prime implicant table.

cyclic-unsolved An example is called *cyclic-unsolved* if ESPRESSO-EXACT was able to generate all prime implicants and the prime implicant table, but was unable to complete the minimum covering problem.

too many primes An example is classified as *too many primes* if ESPRESSO-EXACT was unable to enumerate the set of prime implicants or if ESPRESSO-EXACT was unable to generate the prime implicant table.

Table 2.1 summarizes the results of ESPRESSO-EXACT for each problem class. Recall that the computer time was restricted to ten hours on a one MIP machine. ESPRESSO-EXACT was able to solve 114 of the 134 examples. 16 examples failed during prime implicant generation, 1 failed during the prime implicant table generation, and 3 failed trying to find the minimum cover for the prime implicant table. Most of the problems that failed, did so because of an excessive number of prime implicants.

It is interesting to examine the results of ESPRESSO-EXACT as compared to the size of each problem in terms of equivalent inputs. ESPRESSO-EXACT solved only 1 of the 14 problems with more than thirty inputs, 14 of the 17 problems with between twenty and

thirty inputs, and 61 of the 64 problems with between ten and twenty inputs. All of the problems with less than ten inputs were solved.

This success for industrial functions should be contrasted with the results from two randomly generated functions of ten inputs and ten outputs. These examples, (*ex* and *ex1010*), which are not included in the benchmark set, remain unsolved by ESPRESSO-EXACT. These functions have a large number of don't-care points and have large, dense covering tables. Hence, minimization of some functions with only thirteen equivalent inputs remains intractable.

The previous version of ESPRESSO-EXACT (Version 2.2 of ESPRESSO) was able to solve only 104 of the 134 examples given the same constraint of ten hours of computer time. For the 104 examples which both programs could solve, Version 2.2 required 36.4 hours and Version 2.3 required 17.5 hours. However, the difference on some problems is much greater. For example, on *mlp4*, Version 2.2 required 4,700 seconds while Version 2.3 required only 490 seconds. Note that the minimum solution for *mlp4* is 121, and not 119 as given in [62].

The prime generation of Version 2.2 uses the OFF-set algorithm. The prime generation technique of Version 2.3 uses the unate recursive paradigm. As mentioned earlier, Version 2.2 required 31.3 hours to generate the prime implicants for 113 examples, and Version 2.3 required only 14.5 hours for the same set of examples. For the 104 examples solved by Version 2.2, 18.9 hours were spent in prime generation by Version 2.2, and 5.8 hours were spent in prime generation by Version 2.3. Hence, 13.1 hours of the 18.9 hour difference between the two algorithms is accounted for by the change in the prime generation algorithm. However, the remainder of the performance improvement, and the ability of Version 2.3 to solve ten more problems, is due to the use of a sparse-matrix data structure for the covering table and the inclusion of Gimpel's reduction step.

2.9.2 Comparison with McBoole

MCBOOLE is an exact minimization algorithm developed at the University of McGill [24]. MCBOOLE is also based on the Quine-McCluskey algorithm. MCBOOLE uses a recursive algorithm for prime generation based on the binary-valued Shannon Cofactor and the consensus operation. During the prime generation, a tree structure is maintained showing where a cube is generated; this improves the prime generation algorithm by reducing the number of pairwise consensus operations. MCBOOLE does not generate the prime impli-

Class	Count	Espresso	McBoole
trivial	9	9	9
noncyclic	56	56	56
cyclic-solved	49	49	21
cyclic-unsolved	3	0	0
too many primes	17	0	0
Totals	134	114	86

Table 2.2: Comparison of Espresso-Exact and McBoole.

cant table; instead, it uses a directed graph, constructed during the prime generation, to represent the covering problem. The minimum cover is extracted directly from this graph.

The MCBOOLE prime generation algorithm is similar to the unate recursive paradigm prime generation algorithm given in Section 2.5.1. However, it is not clear whether MCBOOLE uses unate functions in order to terminate the recursion early. Also, it is not described in [24] how MCBOOLE generates multiple-output prime implicants.

In this section, the results of a comparison between MCBOOLE and ESPRESSO-EXACT are presented. Each program was allowed ten hours of computer time to solve each of the 134 benchmark examples. The number of problems solved by each program is shown in Table 2.2.

MCBOOLE was able to solve 86 of the problems as compared to 114 solved by ESPRESSO-EXACT. For the 86 problems which both programs solved, the run-time for MCBOOLE was 27.9 hours, and the run-time for ESPRESSO-EXACT was 20.1 hours. Both programs solved all of the trivial and noncyclic examples. MCBOOLE solved 21 of the cyclic-solved examples versus the 49 solved by ESPRESSO-EXACT. ESPRESSO-EXACT holds an advantage for these difficult problems due to the generation of the prime implicant table and the techniques used to solve the covering problem.

MCBOOLE generated the prime implicants for 117 of the examples and ESPRESSO-EXACT generated the prime implicants for 118 of the examples. For a subset of the problems solved by both algorithms, the prime generation time for MCBOOLE was 36.5 hours and the prime generation time for ESPRESSO-EXACT was 9.7 hours. The reasons for this difference are not clear, as the algorithms are similar. MCBOOLE uses a technique to reduce the number of pairwise consensus operations which is not present in ESPRESSO-EXACT. However, ESPRESSO-EXACT terminates the recursion at weakly-unate functions and handles the

multiple-output nature of the problem uniformly using multiple-valued functions. Also, differences in implementation between the two programs cannot be discounted as a reason for this difference.

2.10 Conclusions

Two-level minimization is an important step for both PLA optimization and multiple-level logic synthesis. Effective heuristic techniques have been developed which provide solutions for large minimization problems in a reasonable amount of time. However, these heuristic algorithms provide no measure of assurance as to the solution quality. Exact algorithms for two-level minimization can always be expected to fail in some situations; however, it is interesting to explore where the boundary between those problems which can be solved and those which cannot lies. As a side-benefit, an exact algorithm provides a measure of the solution quality for the heuristic algorithms, for those problems which can be solved exactly.

This chapter has presented an exact algorithm for two-level minimization of multiple-valued functions. This algorithm is based on extensions to the ESPRESSO-MV algorithm and is called ESPRESSO-EXACT. Experimental results for this program show that, although ESPRESSO-EXACT is unable to solve some problems in a reasonable amount of time, it is able to solve a large percentage of the PLA minimization problems which appear on integrated circuits. Hence, the effective range of two-level minimization for these functions has been greatly extended.

Two interesting questions arise from this work.

First, the functions which are built in PLA form are are quite special. Almost all functions of n variables have $O(2^n)$ product terms in their minimum representation and yet PLA'S are routinely built with more than fifty inputs and only several hundred product terms. In what way can we understand the characteristics of these functions and use these characteristics to improve heuristic algorithms for minimization ?

Second, many of the PLA'S in the Berkeley benchmark set have noncyclic covering problems. This means that the selection of a minimum number of prime implicants does not involve any choice. An interesting question is whether an efficient algorithm can be devised to solve a noncyclic minimization problem which does not require enumerating all prime implicants or forming the prime implicant table. Techniques are known to derive

the essential prime implicants without generating all prime implicants; the difficult part is devising an algorithm to find the secondary essential prime implicants efficiently (secondary essential primes are prime implicants which become essential once the totally dominated prime implicants are removed). This algorithm should also detect when an exact minimum is not reached.

2.11 PLA Test Set Classification

The following table summarizes the results for the classification of the 134 PLA examples in the Berkeley PLA test set. The number of inputs, outputs, and initial product terms are given for each example. The initial number of product terms is marked with an asterisk if the PLA contains a don't-care set. Each PLA is identified as either *indust* or *math*. The origin for an *indust* example is an industrial or University integrated circuit design. The *math* examples are arithmetic functions (e.g., adder, multiplier). The classification for each example (*trivial*, *noncyclic*, *cyclic-s*, *cyclic-us*, *primes*) is shown in the table. Then the number of prime implicants, the number of essential prime implicants, and the minimum solution are given. For the unsolved examples, upper and lower bounds are given on the minimum solution.

name	in/out	terms	type	class	primes	essen	solution
alu1	12/8	19	indust	trivial	780	19	19
bcd.div3	4/4	* 9	math	trivial	13	9	9
clpl	11/5	20	indust	trivial	143	20	20
co14	14/1	14	math	trivial	14	14	14
max46	9/1	46	indust	trivial	49	46	46
newapla2	6/7	7	indust	trivial	7	7	7
newbyte	5/8	8	indust	trivial	8	8	8
newtag	8/1	8	indust	trivial	8	8	8
ryy6	16/1	112	indust	trivial	112	112	112
add6	12/7	1092	math	noncyclic	8568	153	355
adr4	8/5	255	math	noncyclic	397	35	75
al2	16/47	103	indust	noncyclic	9179	16	66
alcom	15/38	47	indust	noncyclic	4657	16	40
alu2	10/8	* 87	indust	noncyclic	434	36	68
alu3	10/8	* 68	indust	noncyclic	540	27	64
apla	10/12	* 112	indust	noncyclic	201	0	25
b4	33/23	* 54	indust	noncyclic	6455	40	54
b11	8/31	* 74	indust	noncyclic	44	22	27
b2	16/17	110	indust	noncyclic	928	54	104
b7	8/31	* 74	indust	noncyclic	44	22	27
b9	16/5	123	indust	noncyclic	3002	48	119
bc0	26/11	419	indust	noncyclic	6596	37	177
bca	26/46	* 301	indust	noncyclic	305	144	180
bcb	26/39	* 299	indust	noncyclic	255	137	155
bcd	26/38	* 243	indust	noncyclic	172	100	117
br1	12/8	34	indust	noncyclic	29	17	19
br2	12/8	35	indust	noncyclic	27	9	13
dc1	4/7	15	indust	noncyclic	22	3	9
dc2	8/7	58	indust	noncyclic	173	18	39
dk17	10/11	* 57	indust	noncyclic	111	0	18
ex7	16/5	123	indust	noncyclic	3002	48	119
exep	30/63	* 149	indust	noncyclic	558	82	108
exp	8/18	* 89	indust	noncyclic	238	30	56
in1	16/17	110	indust	noncyclic	928	54	104
in3	35/29	75	indust	noncyclic	1114	44	74
in5	24/14	62	indust	noncyclic	1067	53	62
in6	33/23	54	indust	noncyclic	6174	40	54
in7	26/10	84	indust	noncyclic	2112	31	54
life	9/1	140	math	noncyclic	224	56	84
luc	8/27	27	indust	noncyclic	190	14	26
m1	6/12	32	indust	noncyclic	59	6	19
newapla	12/10	17	indust	noncyclic	113	9	17
newapla1	12/7	10	indust	noncyclic	31	9	10
newcond	11/2	31	indust	noncyclic	72	18	31
newcpla2	7/10	19	indust	noncyclic	38	14	19

name	in/out	terms	type	class	primes	essen	solution
newcwp	4/5	11	indust	noncyclic	23	7	11
newtpla	15/5	23	indust	noncyclic	40	16	23
newtpla1	10/2	4	indust	noncyclic	6	3	4
newtpla2	10/4	9	indust	noncyclic	23	4	9
newxcpla1	9/23	40	indust	noncyclic	191	18	39
p82	5/14	24	indust	noncyclic	48	16	21
prom1	9/40	502	indust	noncyclic	9326	182	472
radd	8/5	120	math	noncyclic	397	35	75
rckl	32/7	96	math	noncyclic	302	6	32
rd53	5/3	31	math	noncyclic	51	21	31
rd73	7/3	147	math	noncyclic	211	106	127
risc	8/31	74	indust	noncyclic	46	22	28
sex	9/14	23	indust	noncyclic	99	13	21
sqn	7/3	84	indust	noncyclic	75	23	38
t2	17/16	* 128	indust	noncyclic	233	25	52
t3	12/8	148	indust	noncyclic	42	30	33
t4	12/8	* 38	indust	noncyclic	174	0	16
vg2	25/8	110	indust	noncyclic	1188	100	110
vtx1	27/6	110	indust	noncyclic	1220	100	110
x1dn	27/6	112	indust	noncyclic	1220	100	110
x9dn	27/7	120	indust	noncyclic	1272	110	120
z4	7/4	127	math	noncyclic	167	35	59
Z5xp1	7/10	128	math	cyclic-s	390	8	63
Z9sym	9/1	420	math	cyclic-s	1680	0	84
addm4	9/8	480	math	cyclic-s	1122	24	189
amd	14/24	171	indust	cyclic-s	457	32	66
b10	15/11	* 135	indust	cyclic-s	938	51	100
b12	15/9	431	indust	cyclic-s	1490	2	41
b3	32/20	* 234	indust	cyclic-s	3056	123	210
bcc	26/45	* 245	indust	cyclic-s	237	119	137
chkn	29/7	153	indust	cyclic-s	671	86	140
cps	24/109	654	indust	cyclic-s	2487	57	157
dekoder	4/7	* 10	indust	cyclic-s	26	3	9
dist	8/5	255	math	cyclic-s	401	23	120
dk27	9/9	* 20	indust	cyclic-s	82	0	10
dk48	15/17	* 42	indust	cyclic-s	157	0	21
exps	8/38	* 196	indust	cyclic-s	852	56	132
f51m	8/8	255	math	cyclic-s	561	13	76
gary	15/11	214	indust	cyclic-s	706	60	107
in0	15/11	135	indust	cyclic-s	706	60	107
in2	19/10	137	indust	cyclic-s	666	85	134
in4	32/20	234	indust	cyclic-s	3076	118	211
inc	7/9	* 34	indust	cyclic-s	124	12	29
intb	15/7	664	indust	cyclic-s	6522	186	629
l8err	8/8	* 253	math	cyclic-s	142	15	50

name	in/out	terms	type	class	primes	essen	solution
lin.rom	7/36	128	indust	cyclic-s	1087	8	128
log8mod	8/5	46	math	cyclic-s	105	13	38
m181	15/9	430	math	cyclic-s	1636	2	41
m2	8/16	96	indust	cyclic-s	243	7	47
m3	8/16	128	indust	cyclic-s	344	4	62
m4	8/16	256	indust	cyclic-s	670	11	101
mark1	20/31	* 23	indust	cyclic-s	208	1	19
max128	7/24	128	indust	cyclic-s	469	6	78
max512	9/6	512	indust	cyclic-s	535	20	133
mlp4	8/8	225	math	cyclic-s	606	12	121
mp2d	14/14	123	indust	cyclic-s	469	13	30
newcpla1	9/16	38	indust	cyclic-s	170	22	38
newill	8/1	8	indust	cyclic-s	11	5	8
opa	17/69	342	indust	cyclic-s	477	22	77
pope.rom	6/48	64	indust	cyclic-s	593	12	59
root	8/5	255	math	cyclic-s	152	9	57
spla	16/46	* 2296	indust	cyclic-s	4972	33	248
sqr6	6/12	63	math	cyclic-s	205	3	47
sym10	10/1	837	math	cyclic-s	3150	0	210
t1	21/23	796	indust	cyclic-s	15135	7	100
tial	14/8	640	math	cyclic-s	7145	220	575
tms	8/16	30	indust	cyclic-s	162	13	30
wim	4/7	* 10	indust	cyclic-s	25	3	9
x6dn	39/5	121	indust	cyclic-s	916	60	81
ex5	8/63	256	indust	cyclic-us	2532	28	62/67
max1024	10/6	1024	indust	cyclic-us	1278	14	249/261
prom2	9/21	287	indust	cyclic-us	2635	9	276/287
accpla	50/69	183	indust	primes	?	97	97/175
ex4	128/28	620	indust	primes	?	138	138/279
ibm	48/17	173	indust	primes	?	172	173/173
jbp	36/57	166	indust	primes	?	0	0/122
mainpla	27/54	181	indust	primes	?	29	29/172
misg	56/23	75	indust	primes	?	3	3/69
mish	94/43	91	indust	primes	?	3	3/82
misj	35/14	48	indust	primes	?	13	13/35
pdc	16/40	* 2406	indust	primes	?	2	2/100
shift	19/16	100	indust	primes	?	100	100/100
signet	39/8	124	indust	primes	?	104	104/119
soar.pla	83/94	529	indust	primes	?	2	2/352
ti	47/72	241	indust	primes	?	46	46/213
ts10	22/16	128	indust	primes	?	128	128/128
x2dn	82/56	112	indust	primes	?	2	2/104
x7dn	66/15	622	indust	primes	?	378	378/538
xparc	41/73	551	indust	primes	15039	140	140/254

Chapter 3

Algebraic Decomposition

This chapter addresses the multiple-level logic synthesis problem of structuring a logic network into an optimal form. The primary synthesis part of this procedure is the identification of new functions to introduce into a network in order to reduce the complexity of the logic network. This is an extremely difficult optimization problem because of the staggering number of solutions for even a small set of logic equations.

The techniques proposed in this chapter build on the work of Brayton and McMullen and the algebraic approximation they formally introduced [17]. Algebraic decomposition uses the concepts of *kernel intersections* and *common cubes* to find optimal factors to introduce into a network. The extension provided in this thesis is the unification of these algorithms in a single framework based on the *rectangle-covering problem.* This leads to efficient implementations of the same algorithms proposed earlier, and new algorithms to provide even faster approximate solutions. Results are also improved as the new approach allows for solutions not previously considered.

This chapter is organized as follows. First, the general problem of logic structuring is described and previous techniques are examined. The algebraic approximation of Brayton *et al.* is then reviewed in detail. Next, the concepts of rectangles and rectangle covers are defined. The application of rectangles and rectangle covering to the problem of identifying common divisors in a collection of logic equations is then presented. Specifically, the problems of *cube extraction* and *kernel extraction* are treated in detail. Algorithms for finding rectangles in a matrix and for finding optimal rectangle covers are presented. The chapter concludes with experimental results based on an implementation of these algorithms.

3.1 Introduction

Logic structuring starts from an initial representation of a multiple-level logic circuit called a *Boolean network.*

Definition 3.1.1 *A* Boolean network *is a three-tuple (V, E, F) consisting of a directed acyclic graph $G = (V, E)$ and a collection of logic functions F. The source nodes of the DAG are the* primary inputs, *the sink nodes of the DAG are the* primary outputs *and the remaining nodes are the* internal nodes. *Associated with each internal node $v_i \in V$ is a representation of a completely-specified logic function $F_i \in F$.*

There is a one-to-one mapping between a multiple-level set of Boolean equations and a Boolean network. Likewise, any net-list interconnection of gates can be viewed as a Boolean network where each gate is replaced by the logic function which it represents.

The logic functions in the Boolean network are represented in both *sum-of-products* form and *factored-form.* The sum-of-products is the standard representation for a two-level logic function; typically a minimal two-level representation is used. A factored form is a tree representation of a logic function using the operators AND, OR, and NOT. More precisely, a factored form is either a literal, or a sum or product of factored forms.

Example 3.1.1

$$a,$$
$$ab,$$
$$abc + ad,$$
$$((a + b)(cd + e) + f)g.$$

All of these expressions are factored forms; the first three are also sum-of-products forms.

The *literals-in-sum-of-products-form* cost function for a Boolean network is the sum over all nodes of the number of literals in the sum-of-products representation for the function at the node.

The *literals-in-factored-form* cost function for a network is the number of literals in an optimal factored form for each expression in the network. The optimization problem of deriving an optimal factored form for a Boolean function is called *factoring* and is treated elsewhere [76,14]. For our purposes, it is assumed that there is a efficient way to find an optimal factored form for an expression.

The optimization problem considered in this chapter is the problem of finding a set of new nodes and edges to introduce into the Boolean network, and the new logic functions for these nodes, so as to reduce a technology-independent measure of the complexity of the Boolean network. The technology-independent measure for the size of a Boolean network is the total number of literals in sum-of-products form for the network.

3.2 Previous Work

There is a long history in the area of automatic logic synthesis for the problem of deriving an optimal Boolean network from a starting network. However, the number of proposed techniques which are practical for large networks (e.g., more than 1,000 gates) are limited. The techniques reviewed here are the optimal NAND-gate synthesis of Dietmeyer and Su [28], the local transformation approach of LSS [26,25], and the algebraic approach of YLE [17,18].

3.2.1 Dietmeyer-Su Factoring

One of the first techniques which was practical for large circuits is the factoring technique of Dietmeyer and Su [28]. Their technique starts with a two-level minimized representation of a single-output function. The common factors considered are single-cube factors (e.g., factors of the form $x_1x_2x_3$). The single-cube factor is identified from the cube representation of the logic function by choosing a *common factor subarray* and *common factor* which maximizes the *figure-of-merit*. The figure-of-merit is the width of the cube factor times the height of the common factor subarray. Three techniques are given for implementing the common factor and the common factor subarray using NAND-gates; all three are evaluated for the common factor which maximizes the figure-of-merit and the one requiring the fewest gates is chosen. The evaluation function for each implementation counts the number of inverters and bounded-fanin gates needed to realize the circuit assuming this common factor is chosen. Two algorithms for finding the common factor and common factor subarray are presented; one which finds the common factor with a maximum figure-of-merit, and a heuristic algorithm which rapidly finds a factor with a good figure-of-merit.

The primary limitation of Dietmeyer-Su factoring is that common factors which consist of more than one cube are not considered. While it is possible to find multiple-cube

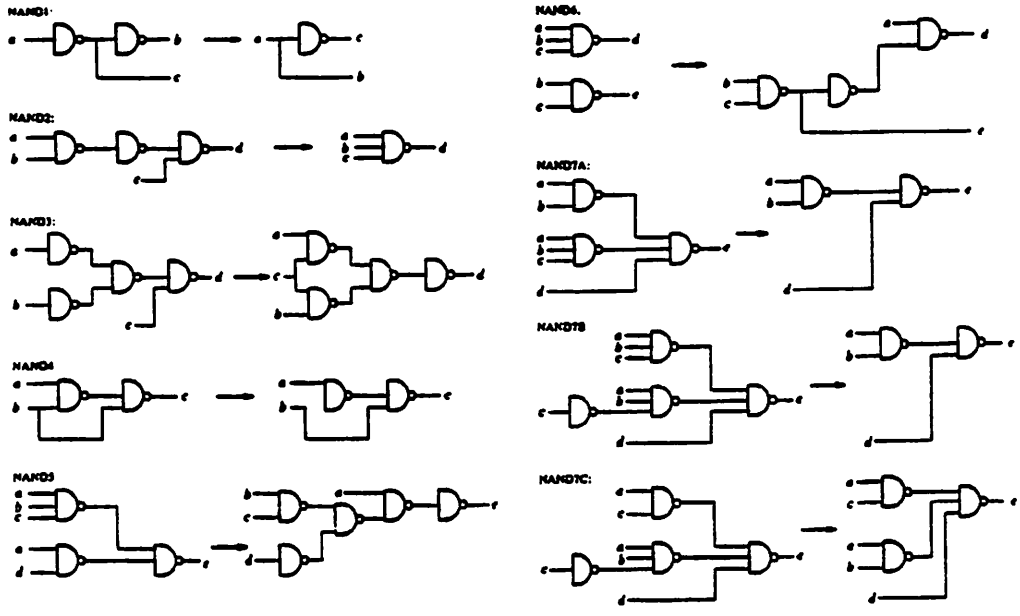

Figure 3.1: Example rules used by LSS.

factors during common-cube extraction, nothing in the heuristic cost function for a common factor guides the selection towards these factors.

3.2.2 Local Transformation

Structuring in LSS is performed on a special form of Boolean network. The technology-independent representation of LSS uses a single functional form for each node in the network (NAND-gates or NOR-gates depending on the target technology). Optimization is done by applying transformations which locally replace a subgraph of the network with a functionally equivalent subgraph. The transformations attempt to improve the area measure of the circuit, which is a function of the number of nodes and connections (equivalently, the number of nodes and literals), and the delay measure of the circuit, which is the number of levels of logic along the critical path.

Some sample local transformations used by LSS are shown in Figure 3.1 [26]. The local transformations include factoring (e.g., rules NAND3 and NAND5 represent $ab + ac \leftrightarrow$

$a(b + c)$) and single-cube extraction (e.g., rule NAND6 represents $(d = abc, e = bc) \rightarrow (d = ae, e = bc)$). The transformations also include Boolean identities such as single-cube containment (e.g., rule NAND7A represents $abc + ab \rightarrow ab$), consensus (e.g., rule NAND7B represents $ab + a\bar{b} \rightarrow a$), and absorption (e.g., rule NAND 7C represents $a + \bar{a}b \rightarrow a + b$).

Little information is given on how to solve the problem of choosing where to apply a given transformation in the network. The implication is that the transformations are ordered and the nodes of the network are ordered. The first transformation which applies at a node is examined, and if the cost decreases with the application of the transformation, it is accepted. The optimization terminates when no transformations are able to improve the cost of the circuit.

The advantage of this approach is that the technology-specific effect of each transformation is easy to predict. Hence, the cost function at each step accurately represents the cost of the actual circuit. Also, technology-specific characteristics such as allowing wired-or connections, using a dual-rail technology family, or constraining maximum fan-in or maximum fan-out, are easily considered at the same time the transformations are applied. The primary disadvantage is the lack of global information when the decision is made to apply a particular transformation.

3.2.3 Algebraic Techniques

Algebraic techniques begin with a fixed sum-of-products form for each logic function in the Boolean network. The starting form is typically the sum-of-products representation with the fewest terms or the fewest literals. Traditional two-level minimization techniques are used to find the initial representation. The algebraic approximation views the logic equations as multilinear monomials with unit coefficients over the variables $\{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$. Each literal (i.e., x_1 and $\bar{x}_1$) is treated as an independent variable and hence the Boolean identity $x_1\bar{x}_1 = \emptyset$ is not recognized. Further, multiplication is not defined when two functions share a common variable. Hence, the Boolean identity $x_1x_1 = x_1^2 = x_1$ is not recognized. A logic function viewed with the algebraic approximation is called an expression.

The motivation for this approximation is the development of efficient algorithms to decompose expressions into a better representation. The primary drawback of the algebraic approximation is a potential lack of optimality in the decomposition because the Boolean

properties of the logic functions are not considered. For this reason, improvement techniques which use the Boolean properties of the logic equations, such as don't-care set minimization [20], multiple-level minimization [11], and global-flow [10] are typically used after algebraic structuring techniques to improve optimization quality.

Brayton and McMullen [17] introduced the notion of a *kernel* of an algebraic expression and showed how to use kernels to find multiple-cube factors which are common to two or more expressions. They also formally introduced the notion of the algebraic approximation as described above. Because of the importance of their technique to the work described here, their notions of algebraic expressions and algebraic decomposition will be covered in detail in the next section.

3.3 Algebraic Techniques

3.3.1 Basic Definitions

In this section, the basic definitions for algebraic decomposition are presented.

A *variable* is a symbol representing a single coordinate of a Boolean space (e.g., a).

A *literal* is a variable or its negation (e.g., a or $\overline{a}$).

A *cube* is a set C of literals such that $x \in C$ implies $\overline{x} \notin C$. For example, $\{a, \overline{b}, c\}$ is a cube, while $\{a, \overline{a}\}$ is not. A cube represents the Boolean function which is the conjunction of its literals.

An *expression* is a set of cubes, and a *nonredundant expression* is an expression where no cube properly contains another. For example, $\{\{a\}, \{b, \overline{c}\}\}$ is a nonredundant expression consisting of the cubes $\{a\}$ and $\{b, \overline{c}\}$. $\{\{a, b\}, \{a\}\}$ is an expression, but is redundant because the cube $\{a, b\}$ contains the cube $\{a\}$. An expression represents the Boolean function which is the disjunction of its cubes.

The expression $\emptyset$ represents the Boolean function 0, and the expression $\{\emptyset\}$ represents the Boolean function 1.

Cubes and expressions are normally written using conventional algebraic notation. For example, the cube $\{a, \overline{b}, c\}$ is normally written $a\overline{b}c$, and the expression $\{\{a\}, \{b, \overline{c}\}\}$ is normally written as $a + b\overline{c}$. However, care should be taken when interpreting set operations applied to this definition of cubes and expressions. For example, $abc \cup d$ equals $abcd$

($\{a,b,c\} \cup \{d\} = \{a,b,c,d\}$), which is not equal to $abc + d$. Likewise, the set $\{a,b\}$ contains the set $\{a\}$, but the logic function ab is contained in the logic function a. Throughout this chapter, set operations are to be interpreted according the definitions above, and not in the sense of Boolean functions.

The *union* of two expressions f and g is the set $f \cup g$ made into a nonredundant expression by deleting any set which contains another set in the union.

Proposition 3.3.1 *The Boolean function corresponding to the union of two expressions equals the Boolean sum of the Boolean functions implied by each expression.*

The *support* of an expression, written $\sup(f)$, is the set of variables x such that either $x \in C$ for some $C \in f$, or $\overline{x} \in C$ for some $C \in f$. Informally, it is the set of variables that f is defined over.

Two expressions f and g have *disjoint support* if $\sup(f) \cap \sup(g) = \emptyset$.

The *product* of two expressions f and g, written fg, is the set $\{c_i \cup d_j | c_i \in f, d_j \in g\}$ made into a nonredundant expression by deleting any set which contains both x and $\overline{x}$ for any variable x and deleting any set which contains another set in the expression.

Proposition 3.3.2 *The Boolean function corresponding to the product of two expressions equals the Boolean product of the Boolean functions implied by each expression.*

If f and g are expressions with disjoint support, then the set $\{c_i \cup d_j | c_i \in f, d_j \in g\}$ is directly a nonredundant expression. In this case, the product is called an *algebraic product.*

The *algebraic quotient* of an expression f by an expression g, written f/g, is the largest set of cubes q such that $f = qg + r$ where q and g have disjoint support.

Given two expressions f and g, an expression d is called a *common subexpression*, *common divisor* or *common factor* of f and g if f and g can be written as $f = q_1 d + r_1$ and $g = q_2 d + r_2$ where both q_1 and q_2 are nonzero.

An expression is *cube-free* if no cube divides the expression evenly (i.e., without a remainder).

The *primary divisors* of an expression f are the expressions f/c where c is a cube. The set of all primary divisors of f is written $D(f)$.

The *kernels* of an expression f are the cube-free primary divisors of the expression. The set of all kernels of f is written $K(f)$. The cube c used to obtain the kernel $k = f/c$ is called the *co-kernel* of f.

Example 3.3.1 *Given the expression*

$$X = abcdg + abcdh + abce + abcf + abi$$

then

$$X/a = bcdg + bcdh + bce + bcf + bi$$

is a primary divisor, but is not a kernel because the expression is not cube-free (the cube b divides each term). However,

$$X/ab = cdg + cdh + ce + cf + i$$

and

$$X/abcd = g + h$$

are both kernels with associated co-kernels ab and abcd.

Since no single cube is cube-free, a kernel must contain two or more cubes. Also, because **1** is a cube, if f is cube-free, then f is considered one of its own kernels.

The *level* of a kernel is defined to provide easily identifiable subsets of the set of all kernels. Recall that kernels are expressions and hence it makes sense to refer to the kernels of a kernel. A kernel is called a *level-0 kernel* of f if it does not have any kernels except itself. In a level-0 kernel no literal appears twice. A kernel is called a *level-n kernel* if it contains a kernel of level $n-1$, but does not contain any kernels of level-n except itself.

The motivation for this definition of the kernels of a logic expression comes from the following theorem:

Theorem 3.3.1 *[17] f and g have a common multiple-cube divisor if and only if there exists $k_f \in K(f)$, and $k_g \in K(g)$ such that $|k_f \cap k_g| \geq 2$.*

That is, two functions have a common multiple-cube divisor if and only if the intersection of a kernel from f and a kernel from g has more than one cube. It is important to remember that an expression is a set of cubes and the intersection of kernels refers to the set intersection of the expressions, and not the Boolean intersection of the logic functions implied by the expressions.

The *cube-literal matrix* of a logic expression $f = \{c_i\}$ is a $\{0,1\}$ matrix B where each row corresponds to a cube of the expression and each column corresponds to each literal in $\cup_i c_i$. The position B_{ij} is set to a 1 if cube c_i contains literal j.

Example 3.3.2 *The expression from Example 3.3.1 has the following cube-literal matrix:*

		a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
		1	2	3	4	5	6	7	8	9
abcdg	1	1	1	1	1	0	0	1	0	0
abcdh	2	1	1	1	1	0	0	0	1	0
abceh	3	1	1	1	0	1	0	0	1	0
abcfh	4	1	1	1	0	0	1	0	1	0
abh	5	1	1	0	0	0	0	0	1	0

3.3.2 Kernel Decomposition Algorithm

The decomposition algorithm presented in [17] is:

1. Distill Algorithm
   (a) Enumerate all kernels for each logic expression.
   (b) Find a pair of kernels which intersect in two or more cubes.
   (c) Substitute the new expression into the network.
   (d) Repeat (a)-(d) while useful intersections are found in step (b).
2. Condense Algorithm
   (a) Select two cubes which intersect in two or more literals.
   (b) Substitute the new factor into the network.
   (c) Repeat (a)-(c) while useful intersections are found in step (a).

In the first phase (*distillation*), multiple-cube common divisors are extracted from the network until no multiple-cube common divisors remain. At this point, the only common divisors are single cubes. In the second phase (*condensation*), single cube divisors are extracted until no common divisors remain. At this point, the only common divisors in the network are single literals.

In [17], no algorithm for generating the single-cube divisors or kernel intersections is presented, and no algorithm for choosing among the possible single-cube and multiple-cube divisors is outlined. The remainder of this chapter will present how to unify these techniques in terms of rectangles and rectangle covers of a $\{0,1\}$ matrix.

3.4 Rectangles and the Rectangle Covering Problem

In this section, rectangles and rectangle covers are defined. In the sections which follow, the applications of these ideas to the detection of common single-cube and multiple-cube divisors are presented.

3.4.1 Basic Definitions

A *rectangle* (R, C) of a matrix B, $B_{ij} \in \{0, 1, *\}$, is a subset of rows R and a subset of columns C such that $B_{ij} \in \{1, *\}$ for all $i \in R, j \in C$.

A rectangle (R_1, C_1) is said to *strictly contain* rectangle (R_2, C_2) if $R_2 \subseteq R_1$ and $C_2 \subset C_1$ or $R_2 \subset R_1$ and $C_2 \subseteq C_1$.

A *trivial rectangle* is a rectangle (R, C) with $|R| \leq 1$ or $|C| \leq 1$.

A *prime rectangle* (R, C) of B is a rectangle which is not strictly contained in any other rectangle of B.

Example 3.4.1 *In the following matrix,*

	1	2	3	4	5
1	1	1	1	0	0
2	1	*	1	0	*
3	0	1	1	0	1
4	1	0	1	1	1

$(\{1,2\},\{2,3\})$ *is a rectangle, but is not a prime rectangle because it is contained in the rectangle* $(\{1,2\},\{1,2,3\})$ *which is a prime rectangle.* $(\{2,3\},\{1,2\})$ *is not a rectangle because* $B_{31} = 0$. *Note that the rows and columns of a rectangle need not be adjacent;* $(\{2,4\},\{1,3,5\})$ *is a rectangle.*

In the examples which follow, a "." is used to represent a 0 in a matrix.

The *co-rectangle* of a rectangle (R, C) is the pair (R, C') where C' is the set of columns not in C. For example, the co-rectangle of $(\{1,2\},\{2,3\})$ is $(\{1,2\},\{1,4,5\})$.

A set of rectangles $\{(R^k, C^k)\}$ form a *rectangle cover* of a matrix B if $B_{ij} = 1$ implies $i \in R^k$, $j \in C^k$ for some k. A covering need not be disjoint so that a 1 in B may be covered by more than one rectangle. The points of B which are labeled $*$ are not required to be covered by any rectangle in the cover. These points represent *don't-care* points in the matrix.

Each rectangle (R^k, C^k) has an associated weight (or cost) defined by a *weight function* $w(R^k, C^k)$.

The weight of a rectangle cover $\{(R^k, C^k)\}$ is defined as the sum

$$\sum_k w(R^k, C^k).$$

The *minimum-weighted rectangle-covering problem* is to find a rectangle cover of a matrix with minimum total weight.

3.4.2 Rectangles and the Maximal Set-Intersection Problem

The maximal set intersection problem is defined as follows. Given a collection of sets over a fixed base set, find all maximal intersections taken over all subsets of the collection. An intersection is maximal if the intersection set is not contained by an intersection formed over a larger subset of the collection.

Operationally, the definition is interpreted as follows. Form all pair-wise intersections of sets keeping only the largest sets ordered by containment. Then form all three-way intersections of sets, again keeping only the largest sets. Further, if any three-way intersection contains a set formed in the pair-wise intersection, the pair-wise intersection set is dropped. The algorithm is continued in this fashion until intersection over all the sets is formed. There are $O(2^n)$ different subsets for n sets in the collection, so that this approach is infeasible in practice except for small values of n.

The problem of generating all prime rectangles of a matrix is the same as the maximal set intersection problem. Form a matrix from the collection of sets, using the standard bit-vector representation for each set; that is, assign a column per element in the base set, and place a 1 in the row for a particular set if the element is that set. A rectangle in this matrix is an intersection over a subset of the original collection of sets. The columns of the rectangle give the set arising from the intersection, and the rows of the rectangle identify the particular subset which yield this intersection set. If the rectangle is a prime rectangle, then the corresponding intersection is maximal.

3.4.3 Rectangles and Kernels

Rectangles in a matrix provide an alternate way of interpreting the kernels of a logic function. First, note that the definition of a kernel does not provide a useful technique for enumerating the kernels of a logic expression. It does not make sense to test f/c for all cubes c to find the kernels of an expression. It is possible to identify the cubes which generate a kernel by looking at the intersections of the cubes of a function. If an intersection over a subset of the cubes is nonempty, then the cube arising from this intersection is a co-kernel, and will yield a kernel when divided into f. Therefore, the problem of finding the kernels of an expression is the maximal set-intersection problem over the cubes of the expression.

Because of the relationship between prime rectangles and the maximal set inter-

section problem, it is now clear that the kernels of an expression are in one-to-one correspondence with the prime rectangles for the matrix formed from the cubes of the expression.

Example 3.4.2 *Continuing the previous example, the expression*

$$x = abcdg + abcdh + abce + abcf + abi$$

has the following cube-literal matrix:

	a 1	*b* 2	*c* 3	*d* 4	*e* 5	*f* 6	*g* 7	*h* 8	*i* 9
1	1	1	1	1	.	.	1	.	.
2	1	1	1	1	.	.	.	1	.
3	1	1	1	.	1	.	.	1	.
4	1	1	1	.	.	1	.	1	.
5	1	1	.	.	.	.	.	1	.

The kernels (and co-kernels) of X are $g + h(abcd)$, $dg + dh + e + f(abc)$, and $cdg + cdh + ce + cf + i(ab)$. The prime rectangles corresponding to each co-kernel are visually evident in the matrix; they are $(\{1,2\},\{1,2,3,4\})$ for $abcd$, $(\{1,2,3,4\},\{1,2,3\})$ for abc, and $(\{1,2,3,4,5\},\{1,2\})$ for ab.

The following proposition states more precisely the relationship between kernels and co-rectangles, and co-kernels and rectangles.

Proposition 3.4.1 *c is a co-kernel of f if and only if it is the cube corresponding to a prime rectangle of the cube-literal matrix of f with at least two rows. The co-rectangle of the prime rectangle identifies the kernel.*

From the rectangle interpretation of kernels, is also possible to understand more clearly the notion of level of a kernel. A level-0 kernel is the co-rectangle of a prime rectangle which has no other rectangle containing its column set. In other words, it corresponds to a prime rectangle of maximal width. A prime rectangle of maximal height corresponds to a kernel of maximal level, i.e., one whose row set is not contained in any other rectangle.

3.4.4 Complexity of Rectangle Covering

The minimum-weighted rectangle covering problem is NP-hard. A solution to the rectangle covering problem with every rectangle given a weight of 1 would provide a solution

to the NP-complete problem known as the *Rectilinear Picture Compression Problem* [31, page 232]. This problem is to determine if a $\{0,1\}$ matrix can be covered by K or fewer rectangles, where a rectangle is defined as a consecutive set of rows and columns.

Another aspect of complexity is the number of rectangles or prime rectangles in a matrix. The n by n matrix with every element set to 1 has 2^{2n} rectangles only 1 of which is prime. However, it is possible for the number of prime rectangles to be exponential in the size of the matrix as the next example shows. This matrix of size n by n has $2^n - 2$ prime rectangles. Only the intersection over the empty set of rows, and the intersection over all rows fail to yield a prime rectangle.

Example 3.4.3 *The following 8 by 8 matrix has 254 prime rectangles:*

```
. 1 1 1 1 1 1 1
1 . 1 1 1 1 1 1
1 1 . 1 1 1 1 1
1 1 1 . 1 1 1 1
1 1 1 1 . 1 1 1
1 1 1 1 1 . 1 1
1 1 1 1 1 1 . 1
1 1 1 1 1 1 1 .
```

3.4.5 Exact Solution for Rectangle Covering

In this section, an exact algorithm for rectangle covering is presented which follows a procedure similar to the Quine-McCluskey algorithm for two-level minimization.

Assume that the weight function for a rectangle obeys the property that if rectangle R_1 strictly contains rectangle R_2, then the weight of rectangle R_1 is no more than the weight of rectangle R_2. With this assumption, only prime rectangles need to be considered for the minimum-weighted rectangle cover. A similar constraint was placed on the cost function for the two-level minimization problem to restrict the solution to prime implicants.

The rectangle covering problem can be formulated as a minimum cover problem as follows. First, all prime rectangles in B are generated. A technique for generating the prime rectangles is presented in Section 3.7.1. Let $P = \{P_j\}$ be the set of all prime rectangles for the matrix, and let w_j be the weight for rectangle P_j. Construct a matrix M which has a row for each 1 in B and a column for each rectangle P_j. Place a 1 in M_{ij} if the rectangle P_j covers the 1 in B associated with row i of M.

The minimum cover problem for the matrix M is to find a {0,1} row vector x such that $M \cdot x^T \geq (1, \ldots, 1)^T$ and the sum $\sum_j x_j w_j$ is minimum. The minimum solution to this covering problem provides the minimum solution to the rectangle covering problem. The branch and bound technique for the minimum cover problem proposed in Chapter 2 can be used to find a minimum-cost rectangle cover.

Unfortunately, the weight of a rectangle in algebraic decomposition does not obey this assumption on the cost of a rectangle; as is shown in the next section, larger rectangles cost more than the rectangles they contain. Therefore, the minimum solution typically involves nonprime rectangles. To solve the problem exactly in this case requires enumeration of all rectangles, and not just the prime rectangles, for inclusion in the covering problem. Because of the tremendous number of rectangles present in even small problems, this is not a viable exact algorithm.

One approximate approach for rectangle covering is to find the minimum cover using prime rectangles and then apply a reduction step to reduce the size of the rectangles to improve the total cover cost. This reduction step, similar to the REDUCE operation in heuristic two-level minimization programs, is presented in Section 3.7.5.

3.5 Application of Rectangle Covering

In this section, the problems of distillation (*kernel-intersection extraction*) and condensation (*common-cube extraction*) are shown to be variations of the rectangle covering problem. The main result from this section is the derivation of a *weight* function and a *value* function for a rectangle for each of these two problems. These functions are defined for the optimization problem of minimizing the total number of literals in the network.

3.5.1 Common-Cube Extraction

Common-cube extraction is the process of finding cubes common to two or more expressions and extracting the common cube to simplify each of the expressions. The optimization problem is to find the particular cubes to introduce into the network to provide an optimal decomposition. For example, the optimal decomposition can be defined as minimizing the total number of literals summed over all expressions, or minimizing the total number of literals given a bound on the number of levels of logic in the final circuit.

Technology-dependent costs, such as the relative cost and delay of n-input NAND-gates can also be used to define the optimal decomposition.

Common-cube extraction has a relationship with rectangles and the rectangle covering problem as follows. First, the cube-literal matrix for the Boolean network is created. As before, each row corresponds to a cube of some expression, and each column corresponds to a literal present in some cube. The position B_{ij} is set to a 1 if cube i contains literal j. A rectangle in the cube-literal matrix identifies a cube which can be extracted from the network. The columns of the rectangle identify the literals in the common-cube, and the rows identify the cubes (and expressions) fed by the cube.

For example, given the equations:

$$\begin{aligned} F &= abc + abd + eg \\ G &= abfg \\ H &= bd + ef, \end{aligned}$$

the cube-literal matrix is:

		a	b	c	d	e	f	g
		1	2	3	4	5	6	7
F_1	1	1	1	1	.	.	.	.
F_2	2	1	1	.	1	.	.	.
F_3	3	.	.	.	.	1	.	1
G_1	4	1	1	.	.	.	1	1
H_1	5	.	1	.	1	.	.	.
H_1	6	.	.	.	.	1	1	.

The columns are annotated with the literal represented by the column, and the rows are annotated with the cube of the function which the row represents. For example F_1 is the cube abc.

The rectangle $(\{1,2,4\},\{1,2\})$ corresponds to the common cube ab. If this common cube is extracted as the new function X, the equations would be rewritten as:

$$\begin{aligned} F &= Xc + Xd + eg \\ G &= Xfg \\ H &= bd + ef \\ X &= ab. \end{aligned}$$

The process of extracting a cube modifies the Boolean network. A new node is added to the Boolean network with a logic function which is the common cube divisor. All

functions which the cube divides are replaced with the algebraic division of the function by the single cube. In order to extract cubes efficiently in an iterative algorithm, it is desirable to modify the cube-literal matrix incrementally to reflect the extraction of the cube. The advantage is that the cube-literal matrix does not have to be re-created as each cube is extracted.

The modification of the cube-literal matrix is straightforward. A new row is added to the cube-literal matrix to reflect the new single-cube expression added to the network, and a new column is added to represent the new literal in the network. The entries covered by the rectangle are marked with * to reflect that the position has been covered, but other rectangles are also allowed to cover the same position. The effect of rectangles which overlap in a position is considered in Section 3.6.

Continuing with the previous example, the cube-literal matrix, after extraction of the rectangle $(\{1,2,4\},\{1,2\})$ is:

		a	b	c	d	e	f	g	X
		1	2	3	4	5	6	7	8
F_1	1	*	*	1	.	.	.	.	1
F_2	2	*	*	.	1	.	.	.	1
F_3	3	.	.	.	.	1	.	1	.
G_1	4	*	*	.	.	.	1	1	1
H_1	5	.	1	.	1	.	.	.	.
H_1	6	.	.	.	.	1	1	.	.
X_1	7	1	1	.	.	.	.	.	.

Rectangle Weight and Value

The choice of the weight function for a rectangle measures the optimization goal for cube extraction. To minimize the total number of literals in the network, the weight of a rectangle is chosen so that the weight of a rectangle-cover of the cube-literal matrix equals the total number of literals in the network after the new single-cube functions are added to the network. Hence, the minimum-weighted cover corresponds to the optimal simultaneous extraction of a collection of cubes.

For cube extraction, the weight of a rectangle is defined as:

$$w(R,C) = \begin{cases} |C| & \text{if } |R| = 1 \\ |R| + |C| & \text{if } |R| > 1 \end{cases}$$

If a rectangle (R,C) has only a single row, this corresponds to leaving the cube unchanged in the network: hence, the weight of this rectangle counts the number of literals in the cube.

If the rectangle has more than one row, this corresponds to creating a new single-cube function (with $|C|$ literals), and substituting this new function into $|R|$ other cubes at a cost of $|R|$ literals; hence the weight of a multiple-row rectangle is $|R| + |C|$.

When searching for a rectangle to extract, it is useful to define a second function called the *value* of a rectangle. For cube extraction, the value of a rectangle is defined as:

$$v(R, C) = |\{(i, j)|B_{ij} = 1\}| - w(R, C).$$

The rectangle value reflects the desirability of choosing the rectangle, and is defined as the number of literals which would be saved in the network if this rectangle is chosen. This is simply the number of 1 points covered by the rectangle minus the weight of the rectangle. No additional literals are saved for covering a point marked as an asterisk in the matrix; hence, these are not counted in the value function. If a rectangle contains only points which are 1, then the value of a rectangle for cube extraction is the area minus the perimeter.

It is useful to define the weight function and value function in terms of three auxiliary functions – the row and column weight vectors w^r and w^c and the element value matrix V. Each row i has weight $w_i^r = 1$, and each column j has weight $w_j^c = 1$. Each position of the matrix V has $V_{ij} = 1$. As each element B_{ij} which is 1 is covered, the value of V_{ij} is set to 0. The weight of a rectangle (R, C) is then defined as

$$w(R, C) = \begin{cases} \sum_{i \in C} w_j^c & \text{if } |R| = 1 \\ \sum_{i \in R} w_i^r + \sum_{i \in C} w_j^c & \text{if } |R| > 1 \end{cases}$$

and the value of a rectangle (R, C) is defined as

$$v(R, C) = \sum_{i \in R, j \in C} V_{ij} - w(R, C).$$

The rectangle algorithms presented in Section 3.7 make use of w_i^r, w_j^c, and V_{ij} to compute $v()$.

3.5.2 Kernel-Intersection Extraction

As described in Theorem 3.3.1, intersections among the kernels of a collection of expressions are useful for finding common multiple-cube divisors between two or more expressions. If two functions share a common multiple-cube divisor, then the common divisor can be found as the intersection of a kernel from each of the functions.

To turn this into an optimization algorithm, the first step is to enumerate all kernels of each logic expression. If desired, the set of kernels is restricted to a subset of all kernels to reduce the processing time at the possible expense of a decrease in the solution quality. For example, a convenient subset is the set of all level-0 kernels. The problem is then to examine the intersections over the subsets of the set of kernels to find intersections to extract and substitute into a network. This problem is naturally mapped into a rectangle covering problem.

The Co-kernel Cube Matrix

Finding useful intersections of kernels is facilitated with the *co-kernel cube* matrix. A row in this matrix corresponds to a kernel (and its associated co-kernel), and a column corresponds to the cubes which are present in some kernel. The entry B_{ij} is set to 1 if the kernel i contains the cube j.

For example, given the equations:

$$\begin{aligned} F &= af + bf + ag + cg + ade + bde + cde \\ G &= af + bf + ace + bce \\ H &= ade + cde, \end{aligned}$$

the kernels (and co-kernels) of F are $de+f+g(a), de+f(b), a+b+c(de), a+b(f), de+g(c)$ and $a+c(g)$. The kernels (and co-kernels) of G are $ce+f(a,b), a+b(f,ce)$, and the only kernel of H is $a+c(de)$. For ease of presentation, the functions F and G, which themselves are kernels, are not listed in the set of kernels. The co-kernel cube matrix is easily constructed from this data. The unique cubes from all of the kernels are a, b, c, ce, de, f, and g; these cubes are used to label the columns of the matrix. There are thirteen kernels, and the corresponding co-kernels are used to label the rows of the matrix.

The product of a co-kernel for a row and the kernel-cube for a column yields a cube of some expression. For reference, the cubes of the original expressions are numbered from 1 to 13. The number of the cube resulting from the product of the co-kernel for row i and the kernel-cube for column j is placed at position B_{ij} in the co-kernel cube matrix. For example, the co-kernel a when multiplied by the kernel $de + f + g$ yields the cubes numbered 5, 1, and 3, which are ade, af, and ag. Note that there is often more than

one way to form each cube in an expression. For example, cube 1 (af) is created by the co-kernel a multiplying the kernel $de + f + g$, and by the co-kernel f multiplying the kernel $a + b$.

The co-kernel cube matrix for the previous example is:

			a	b	c	ce	de	f	g
			1	2	3	4	5	6	7
F	a	1	.	.	.	.	5	1	3
F	b	2	.	.	.	.	6	2	.
F	de	3	5	6	7	.	.	.	.
F	f	4	1	2	.	.	.	.	.
F	c	5	.	.	.	.	7	.	4
F	g	6	3	.	4	.	.	.	.
G	a	7	.	.	.	10	.	8	.
G	b	8	.	.	.	11	.	9	.
G	ce	9	10	11	.	.	.	.	.
G	f	10	8	9	.	.	.	.	.
G	de	11	12	.	13	.	.	.	.

A rectangle of the co-kernel cube matrix identifies an intersection of kernels; this kernel-intersection is a common subexpression in the network. The columns of the rectangle identify the cubes in the subexpression, and the rows of the rectangle identify the particular functions that the subexpression divides. The entries covered by the matrix correspond to cubes from the original network.

From the previous example, the prime rectangle $(\{3,4,9,10\},\{1,2\})$ identifies the subexpression $a + b$ which divides the functions F and G. Cubes numbered 1, 2, 5, 6, 8, 9, 10, and 11 from the original set of functions are covered by this rectangle. This corresponds to the factorization of the equations into the form:

$$\begin{aligned} F &= deX + fX + ag + cg \\ G &= ceX + fX \\ H &= ade + cde \\ X &= a + b. \end{aligned}$$

When a new subexpression is identified, it is inserted into the Boolean network. This consists of adding a new node to the network and dividing the node into each of the expressions which this node divides. The expressions which the subexpression divide are

apparent from the rows in the rectangle for the subexpression. The new co-kernel cube matrix is then created for the modified Boolean network.

To reduce the complexity of extracting each factor from the network, it is desirable to modify the co-kernel cube matrix incrementally as each subexpression is identified. This is done as follows. New rows are added to the co-kernel cube matrix for each kernel of the new subexpression. The cubes which are formed by the insertion of this new factor into the network are marked as covered in the rectangle. This includes the points directly contained by the rectangle, and other points which are labeled with the same number. The cubes which have been covered by the new subexpression are labeled with *. The effect of rectangles which overlap in a position is considered in Section 3.6.

Continuing the previous example, after the rectangle $(\{3,4,9,10\},\{1,2\})$ is extracted, the modified co-kernel cube matrix is:

			a	*b*	*c*	*ce*	*de*	*f*	*g*
			1	2	3	4	5	6	7
F	*a*	1	.	.	.	.	*	*	3
F	*b*	2	.	.	.	.	*	*	.
F	*de*	3	*	*	7	.	.	.	.
F	*f*	4	*	*	.	.	.	.	.
F	*c*	5	.	.	.	.	7	.	4
F	*g*	6	3	.	4	.	.	.	.
G	*a*	7	.	.	.	*	.	*	.
G	*b*	8	.	.	.	*	.	*	.
G	*ce*	9	*	*	.	.	.	.	.
G	*f*	10	*	*	.	.	.	.	.
H	*de*	11	12	.	13	.	.	.	.
X	1	14	14	15	.	.	.	.	.

Note that no new columns were added in this case when the co-kernel cube matrix was modified.

Rectangle Weight and Value

The weight of a rectangle of the co-kernel cube matrix is chosen to reflect the number of literals in the network if the corresponding common subexpression is inserted into the network. A minimum-weighted rectangle-cover of the co-kernel cube matrix then corresponds to a simultaneous selection of a set of subexpressions to add to the network in order to minimize the total number of literals in the network.

In a manner similar to common-cube extraction, weights w_i^r and w_j^c are defined for the rows and columns of the matrix and the weight of a rectangle is defined in terms of these weights. Also, values V_{ij} are defined for the elements of the matrix, and the value of a rectangle is defined in terms of the values of the elements covered by the rectangle, and the weight of the rectangle.

Let w_j^c be the number of literals in the kernel-cube for column j. If a rectangle (R, C) is used to identify a subexpression, then a new function is formed from the columns of C. This new function has $\sum_{j \in C} w_j^c$ literals. Let w_i^r be the number of literals in the co-kernel for row i plus one. A subexpression divides the expressions indicated from the rows R of the rectangle. After algebraic division by the subexpression, each of these expressions consist of a sum of the corresponding co-kernel cubes multiplying the literal for the new expression. Therefore, the number of literals in the affected functions after extraction of the rectangle is $\sum_{i \in R} w_i^r$.

Therefore, the weight of a rectangle (R, C) of the co-kernel cube matrix is defined as:

$$w(R, C) = \sum_{i \in R} w_i^r + \sum_{j \in C} w_j^c.$$

The value of a rectangle measures the difference in the number of literals in the network if the particular rectangle is selected. The number of literals after the rectangle is selected is the weight of the rectangle as defined above. Let V_{ij} be the number of literals in the cube which is covered by position (i, j) of the co-kernel cube matrix. Then, the number of literals before extraction of the rectangle is simply $\sum_{i \in R, j \in C} V_{ij}$. As elements of the co-kernel cube matrix are covered, elements of V are set to zero. This includes the elements V_{ij} covered by the matrix and all other elements which represent the same cube in the network.

The value of a rectangle (R, C) of the co-kernel cube matrix is thus defined as:

$$v(R, C) = \sum_{i \in R, j \in C} V_{ij} - w(R, C)$$

Note that the weight function and value function are the same form as for common-cube extraction; only the definitions for w_i, w_j, and V have changed.

3.6 Effect of Overlapping Rectangles

The formulation of algebraic decomposition as a rectangle covering problem where the rectangles are allowed to overlap allows for valid decompositions which are nonalgebraic. Further, by introducing don't-care points in the matrix, other nonalgebraic decompositions are possible. This effect is described in this section for both cube-extraction and kernel-extraction.

3.6.1 Cube-Extraction

Consider two rectangles which overlap in the cube-literal matrix. Recall that each point of the cube-literal matrix corresponds to a literal of some cube in the network. For ease of presentation, assume the rectangles overlap in a single point in the matrix and call the cube which contains this point the *overlap cube*. The simultaneous extraction of both rectangles corresponds to duplicating the literal which is contained by both rectangles. The literals of the overlap cube which are covered by both rectangles are replaced with the product of the two extracted cubes. However, the two new cubes do not have disjoint support; hence, this corresponds to a nonalgebraic factoring of the original equation. This operation is valid because the Boolean identity $aa = a$ has been used. The duplication of literals, while seeming to introduce extra literals in the network, can lead to better decompositions.

Consider the cube-extraction example from the previous section. The common cube ab (prime rectangle $(\{1,2,4\},\{1,2\})$) has been extracted and the cube-literal matrix has the form:

		a	b	c	d	e	f	g	X
		1	2	3	4	5	6	7	8
F_1	1	*	*	1	.	.	.	.	1
F_2	2	*	*	.	1	.	.	.	1
F_3	3	.	.	.	.	1	.	1	.
G_1	4	*	*	.	.	.	1	1	1
H_1	5	.	1	.	1	.	.	.	.
H_1	6	.	.	.	.	1	1	.	.
X_1	7	1	1	.	.	.	.	.	.

Assume the rectangle $(\{2,5\},\{2,4\})$ is selected as the next cube to extract. This rectangle overlaps the first rectangle in the point $(2,2)$ which duplicates the literal b in cube

F_2. This corresponds to rewriting the equations as:

$$\begin{aligned} F &= Xc + XY + eg \\ G &= Xfg \\ H &= Y + ef \\ X &= ab \\ Y &= bd. \end{aligned}$$

This is a nonalgebraic decomposition because the product XY is used in F, but X and Y do not have disjoint support.

Recall that the weight of each rectangle reflects the number of literals after extraction of each rectangle; the weight of the cover is the final number of literals in the network. The minimum-weighted rectangle-cover selects precisely the literals to duplicate to optimize the decomposition.

3.6.2 Kernel Extraction

Likewise, rectangles are allowed to overlap in the co-kernel cube matrix. The points of the overlap correspond to duplication of the cubes which are contained by both rectangles. The Boolean identity $a + a = a$ is being used which makes this technique valid.

Consider the previous kernel-extraction example. After the subexpression $a + b$ (corresponding to the prime rectangle $(\{3,4,9,10\},\{1,2\})$) has been extracted, the co-kernel cube matrix is:

			a	*b*	*c*	*ce*	*de*	*f*	*g*
			1	2	3	4	5	6	7
F	*a*	1	.	.	.	.	*	*	3
F	*b*	2	.	.	.	.	*	*	.
F	*de*	3	*	*	7	.	.	.	.
F	*f*	4	*	*	.	.	.	.	.
F	*c*	5	.	.	.	.	7	.	4
F	*g*	6	3	.	4	.	.	.	.
G	*a*	7	.	.	.	*	.	*	.
G	*b*	8	.	.	.	*	.	*	.
G	*ce*	9	*	*	.	.	.	.	.
G	*f*	10	*	*	.	.	.	.	.
H	*de*	11	12	.	13	.	.	.	.
X	1	14	14	15	.	.	.	.	.

If the rectangle $(\{3,6,11\},\{1,3\})$ is extracted next, then the overlap in the point $(3,1)$ corresponds to the duplication of the term *ade* for the decomposition of function F. The rectangle $(\{3,6,11\},\{1,3\})$ corresponds to the subkernel $a+c$, which after extraction, yields the equations:

$$\begin{aligned} F &= deX + fX + deY + gY \\ G &= ceX + fX \\ H &= deY \\ X &= a+b \\ Y &= a+c. \end{aligned}$$

Sharad Malik has made the observation that it is possible to add don't-care points to the co-kernel cube matrix before finding the rectangle cover, as follows. Assume $B_{ij} = 0$. If the co-kernel c_i contains literal l ($\bar{l}$) and kernel cube k_j contains literal $\bar{l}$ (l), then set $B_{ij} = *$; that is, allow the position B_{ij} to be optionally covered if it leads to a better factorization. Given that $c_i k_j = \emptyset$, the decomposition remains valid even if a rectangle covers this position.

Len Berman has suggested inserting don't-cares at other positions where it can be proved that the addition of the implied cube to the corresponding function does not change the input-output behavior of the logic network. Berman has suggested a technique based on global-flow to detect efficiently when this condition is satisfied.

3.7 Rectangle Algorithms

Two algorithms for applying rectangle covering to algebraic decomposition are proposed. The first, **greedy_extract**, selects one rectangle at a time and modifies the matrix to reflect the extraction of the rectangle. The advantage of this technique is that it immediately takes into account common factors between the newly extracted function and the rest of the logic network. The disadvantage of this approach is that it selects only one rectangle at a time and does not easily account for the simultaneous extraction of multiple rectangles. Therefore, **covering_extract** is also presented. This algorithm finds the best collection of factors to extract at each step by solving the minimum-weighted rectangle-covering problem heuristically. These rectangles are then extracted, and the entire process

is repeated to find factors between the new expressions and the remainder of the logic network.

Several other algorithms are also described in this section. **gen_rectangles** is an algorithm for enumerating all prime rectangles in a matrix. This is used to generate the kernels of a logic expression, and is used by **best_rectangle** to find the best-valued prime rectangle in a matrix. **ping_pong** provides a heuristic alternative to **best_rectangle** to find a good-valued, but not necessarily the best-valued, rectangle in a matrix.

gen_rectangles is an adaptation of the algorithm originally described in [17] as a technique for kernel generation. The computer implementation described there was done in APL using full matrix techniques. Even though IBM APL uses a bit-matrix representation for $\{0,1\}$-matrices, using full matrices proved to be a limiting factor on the size of networks which could be optimized in YLE [13]. Other full-matrix implementations of this algorithm have been described, including a version by Karen Bartlett [8] and Albert Wang [77].

However, almost all examples of rectangle generation for logic synthesis involve matrices with very few nonzero entries. For example, in the case of common-cube extraction, a network with 1,000 single-cube expressions each with 10 literals creates a cube-literal matrix with 1,000,000 possible entries of which only 10,000 are nonzero. This matrix provides an example a dense matrix for logic optimization, but is only one percent full. Therefore, sparse matrices appear to be a natural representation for implementing and describing the rectangle covering algorithms.

To assist the implementation of algorithms for rectangle covering, a generic package for sparse matrices has been implemented. This package has proven useful for a variety of other applications, including algorithms for unate covering (as described in Chapter 2 of this thesis) and inverter-phase assignment [76]. A sparse matrix stores only the nonzero elements of a matrix. Stored with each sparse matrix element is the row and column number. Space is reserved at each sparse matrix element for application-specific information. The sparse matrix elements are doubly-linked to the next and previous elements in the row, and are doubly-linked to the next and previous elements in the column. Headers for each row and column provide random access to the first element in a row or column, given the row or column number. A doubly-linked list of the nonempty row and column headers are also maintained to provide fast access to the currently occupied rows and columns. As submatrices of the sparse matrix are extracted, the row and column numbers are not changed; this provides convenient correlation between a row in the sparse matrix, and a

row in any submatrix of the matrix.

Pseudo-code for the algorithms in this section are given in an informal pseudo-C notation. The sparse matrix package provides a number of primitive operations which are used in the description. A sparse matrix element has fields *rownum* and *colnum* which give the row and column number of the element. A sparse matrix row or column has a field *length* which gives the number of nonzero elements in that row or column. Note that in practice the sparse matrix element contains arbitrary user information at the given row and column; however, many algorithms are only interested in the topology of the sparse matrix (i.e., where the nonzero elements are) and not their contents. A rectangle has two fields, the set of rows in the rectangle (*rows*), which can be represented with a sparse matrix column vector, and the set of columns in the rectangle (*cols*), which can be represented with a sparse matrix row vector.

3.7.1 gen_rectangles: Finding All Prime Rectangles

gen_rectangles finds all prime rectangles in a matrix. As mentioned earlier, the worst-case complexity of this algorithm is exponential in the size of the matrix. However, when the matrix is sparse, it is often feasible to enumerate all prime rectangles.

The pseudo-code for the **gen_rectangles** is given in Figure 3.3. **gen_rectangles** calls **gen_rectangles_recur** with appropriate initial arguments. The final step is to process and record the trivial prime rectangles. A single row (column) is a prime rectangle if it is not contained in any other row (column). Also, if the matrix does not contain any columns (rows) with all 1's, then the rectangle consisting of all of the rows (columns) and no columns (rows) is also prime.

The arguments to **gen_rectangles_recur** are the current submatrix which is being searched for prime rectangles (*M*), the current column index (*index*), and the rectangle found up to this point (*rect*). Also passed to this routine are a function to be called when a prime rectangle is discovered (*func*), and a generic piece of state information to be passed to this function (*state*).

The recursive assumption on this routine is that all of the rows of *rect* contain a 1 (or *) for all columns of *rect*. The routine will search the submatrix *M* to find all of the prime rectangles with fewer rows but more columns.

Each column *c* with an index greater than the starting index is examined as a

column to include in the rectangle. If the column has only a single element, then it cannot create a nontrivial rectangle, so only columns with 2 or more elements are of interest. The submatrix *M1* of the original matrix is created by selecting only the rows where the column c has a nonzero value, and a new rectangle is formed from the columns of the old rectangle and the rows for which c has a nonzero value.

Any column of the submatrix *M1* (including c) which is now all 1's can also be added to the rectangle. A pruning operation is also performed at this step. If a column of 1's occurs for a column index less than the starting index, then all rectangles in the current submatrix *M1* have already been examined when that column index was processed. Hence, if this condition is detected, it is not necessary to recur.

At this point, *rect1* represents a new prime rectangle of the matrix, and *M1* is a new submatrix to be searched for more prime rectangles. The caller's function is called to process the prime rectangle. This function returns a status indicating whether the submatrix *M1* should be searched further. If the return value is 0, **gen_rectangles** is called recursively.

3.7.2 best_rectangle: Finding a Maximum-Value Prime Rectangle

Given a technique to generate all prime rectangles, it is now trivial to find the best-valued prime rectangle. **gen_rectangles** is used to find all prime rectangles, and the best-valued prime rectangle is recorded.

Note that for the value function defined in terms of V, w^r, and w^c, the best-valued rectangle is not necessarily prime. The value of the prime rectangle can be improved by deleting rows and columns which do not increase the value. If a row (column) of the prime rectangle does not contain any elements of positive value (i.e., $V_{ij} = 0$ for all $i \in R$), then that row (column) can be deleted from the rectangle.

Bounding Techniques

For common-cube extraction in a matrix with no don't-care entries, the value function takes the simple form of the area of the rectangle minus the semi-perimeter; i.e., $v(R, C) = |R||C| - (|R| + |C|)$. In this special case, a bound is easily placed on the size of the largest value rectangle in a matrix. This can be used to speed-up the search for the best-valued prime rectangle.

This is done by converting **gen_rectangles** into a branch and bound algorithm.

```
gen_rectangles_recur(M, index, rect, func, state) {

    foreach column c of M {
        if (c->length >= 2 && c->colnum >= index) {

            M1 = new matrix;
            foreach element p in column c {
                copy row p->rownum of M to M1;
            }

            rect1 = new rectangle;
            rect1->rows = duplicate column c of M;
            rect1->cols = duplicate rect->cols;

            prune = 0;
            foreach column c1 of M1 {
                if (c1->length == c->length) {
                    if (c1->colnum < c->colnum) {
                        prune = 1;
                        break;
                    } else {
                        add c1->colnum to rect1->cols;
                        delete column c1 from M1;
                    }
                }
            }

            if (not prune) {
                bound = (*func)(M1, rect1, state);
                if (not bound) {
                    gen_rectangles_recur(M1, c->colnum,
                                         rect1, func, state);
                }
            }
        }
    }
}
```

Figure 3.2: Algorithm gen_rectangles_recur.

```
gen_rectangles(M, func, state) {
    foreach trivial rectangle rect {
        if the rectangle is prime {
            (*func)(rect, state);
        }
    }
    rect = new rectangle;
    gen_rectangles_recur(M, 0, rect, func, state);
}
```

Figure 3.3: Algorithm gen_rectangles.

At each step of gen_rectangles_recur, assume a rectangle rect1= (R, C) is identified and M1 = $\tilde{M}$ is the remaining submatrix. Assume row i of $\tilde{M}$ contains the most nonzero entries; likewise, assume column j of $\tilde{M}$ contains the most nonzero entries. Let $n_r = |k|\tilde{M}_{jk} = 1|$ and $n_c = |k|\tilde{M}_{ik} = 1|$. Then the largest rectangle contained in $\tilde{M}$ adds at most n_c columns to the current rectangle and has at most n_r rows. Therefore, a simple bound on the value of the best rectangle for the remaining subproblem is $(|C| + n_c)n_r - (|C| + n_c + n_r)$. If this is less than or equal to the value of the best rectangle seen, then the search through the rectangles of $\tilde{M}$ can be avoided. For this reason, the function func which is passed to gen_rectangles is allowed to return a status indicating whether the remaining submatrix should be examined.

The branching of gen_rectangles_recur can also be modified to attempt to find a large rectangle as soon as possible to allow the bounding to be more effective. Rather than iterating for the columns of M in arbitrary order, the columns are ordered by decreasing count of the number of nonzero elements in each column.

A better bounding scheme uses the observation that for the matrix $M1$ to have a rectangle of size l rows by k columns, then at least l of the rows must have more than k elements. Let $r_i, i = 1, \ldots, n$ be the number of nonzero elements in each row in descending order (i.e., $r_1 \geq r_2 \cdots \geq r_n$), and let $c_j, j = 1, \ldots, m$ be the number of nonzero elements in each column also in descending order. Then, a tighter bound on the size of the value of the

best rectangle in $M1$ (and rect1 $= (R, C)$) is

$$\max\{((|C| + r_{c_i})c_i - (|C| + r_{c_i} + c_i)|i = 1, \ldots, m\}$$

That is, for each value of i, the row cardinality of rank c_i is the bound on the number of rows that can form a rectangle with c_i columns.

Experimental results comparing these bounding schemes is presented in Section 3.7.2. The simple bounding technique is effective because it is inexpensive and it reduces the number of rectangles visited. The tighter bound is expensive in practice to implement; sorting the row and column cardinalities at each step is expensive, and the reduction in the number of rectangles examined does not appear to justify the second bounding technique.

3.7.3 ping_pong: Finding a Maximal-Value Rectangle

ping_pong is a heuristic algorithm to find a good-valued rectangle without generating all prime rectangles of the matrix. The inputs to the algorithm are the matrix B and the value function $v()$ which computes the value of a rectangle of B. $v()$ is itself defined in terms of the row and column weights (w^r and w^c) and the value matrix (V).

ping_pong is given in Figure 3.4. **ping_pong_row** is used to find a rectangle starting from the *best* row in the matrix. To make **ping_pong** less dependent on the starting row, **ping_pong_col** is used to find a rectangle starting from the *best* column in the matrix. The best rectangle between these two passes is the final rectangle returned.

In the description of **ping_pong**, the row-oriented algorithms **ping_pong_row** and **greedy_row** are described. The analogous routines **ping_pong_col** and **greedy_col** are the same algorithms applied to the transpose of the matrix B with the weight function adjusted accordingly; hence, these algorithms are not described in detail.

ping_pong_row is given in Figure 3.5. The row i which maximizes the value of the single row rectangle is chosen as the starting seed. This becomes the seed row for **greedy_row** which is used to find a high-value rectangle intersecting row i. This rectangle becomes the current best rectangle (R_b, C_b). The iteration loop of **ping_pong_row** tries to improve the value of this rectangle. This is done first using **greedy_col**, but restricting the initial column seed to one of the columns in the current best rectangle. The rectangle returned will either be the rectangle (R_b, C_b), or will be a rectangle of higher value. If the rectangle value has improved, the new rectangle is recorded as the best rectangle, and the

```
ping_pong(B, v) {
    /* Find a good rectangle starting from the "best" row */
    (R1, C1) = ping_pong_row(B, v);

    /* Find a good rectangle starting from the "best" column */
    (R2, C2) = ping_pong_col(B, v);

    if (v(R1, C1) > v(R2, C2)) {
        (R, C) = (R1, C1);
    } else {
        (R, C) = (R2, C2);
    }
    return (R, C);
}
```

Figure 3.4: Algorithm **ping_pong**.

process is repeated starting from a row chosen from the set of columns in the best rectangle. This process is repeated until no better rectangle is found.

All that remains is the description for **greedy_row**, which is shown in Figure 3.6. **greedy_row** finds a good-valued rectangle which intersects row i. Row i becomes the seed rectangle (R_s, C_s) and the best rectangle seen (R_b, C_b) is initialized to the seed rectangle. During each pass of the loop, all rows not currently in the seed rectangle are examined, and the row k which, if added to the seed rectangle, maximizes the value of the seed rectangle is chosen. This row is then added to the seed rectangle, and columns not in both the seed column set and row k are deleted from the seed rectangle. This is repeated until the seed rectangle consists of only a single column. A sequence of rectangles with increasing row sets and decreasing column sets is generated in this manner; the best value rectangle seen in this process is returned.

In **ping_pong_row**, if the initial row i leads to a rectangle (R_b, C_b) which is trivial (i.e., a rectangle with only a single row or column), then the next best initial row should be chosen instead, continuing until all rows have been tried as an initial row. If this is done and **ping_pong** returns a trivial rectangle, then there is the assurance that no nontrivial

```
ping_pong_row(B, v) {

    /* Find the seed row of maximum value */
    i = arg max_i {v({i}, {j|B_ij = 1})};
    (R_b, C_b) = greedy_row(B, v, i);

    /* Try to improve the rectangle */
    do {
        /* start from best-valued column of (R_b, C_b) */
        j = arg max_j {v({i|B_ij = 1}, {j})|j ∈ C_b};
        (R_1, C_1) = greedy_col(B, v, j);
        if (v(R_1, C_1) > v(R_b, C_b)) {
            (R_b, C_b) = (R_1, C_1);
        } else {
            break;
        }

        /* start from the best-valued row of (R_b, C_b) */
        i = arg max_i {{i}, v({j|B_ij = 1})|i ∈ R_b};
        (R_2, C_2) = greedy_row(B, v, i);
        if (v(R_2, C_2) > v(R_b, C_b)) {
            (R_b, C_b) = (R_2, C_2);
        } else {
            break;
        }
    } while (1);
    return (R_b, C_b);
}
```

Figure 3.5: Algorithm ping_pong_row.

```
greedy_row(B, w^r, w^c, V, i) {
    (R_s, C_s) = ({i}, {j|B_ij = 1});
    (R_b, C_b) = (R_s, C_s);
    while (|C_s| > 1) {
        k = arg max_k {v(R_s ∪ {k}, C_s ∩ {j|B_kj = 1})|k ∉ R_s};
        (R_s, C_s) = (R_s ∪ {k}, C_s ∩ {j|B_kj = 1});
        if v(R_s, C_s) > v(R_b, C_b) {
            (R_b, C_b) = (R_s, C_s);
        }
    }
    return (R_b, C_b);
}
```

Figure 3.6: Algorithm **greedy_row**.

rectangle exists in the matrix.

ping_pong can be implemented efficiently for a sparse-matrix because all of the operations have a complexity related to the sparsity of the matrix. For example, finding the next row to add to the seed rectangle requires examining only the rows which are connected to a column in the seed row; because the matrix is stored with row and column pointers, this requires examining only a subset of the rows in the matrix. Also, incremental computation of the value and cost for each rectangle as rows and columns are added to the rectangle can be used to reduce the complexity of finding the cost for a rectangle.

3.7.4 greedy_extract: Greedy Selection of Rectangles

greedy_extract is given in Figure 3.7. The inputs are the matrix B and the value function $v()$. The value function is represented by the row and column weights w^r and w^c, and the value matrix V. A rectangle is chosen in **choose_rectangle** by either generating all prime rectangles and choosing the best-valued rectangle (**best_rectangle**), or by finding a good-valued rectangle heuristically (**ping_pong**). The matrices are then modified to reflect the extraction of the common factor as described in Section 3.5.

```
greedy_extract (B, v) {
    do {
        (R, C) = choose_rectangle(B, v);
        if ( v(R, C) > 0) {
            modify B to reflect extraction of (R, C);
        }
    } while ( v(R, C) > 0 );
}
```

Figure 3.7: Algorithm **greedy_extract**.

For common-cube extraction, an additional row, representing the new common cube factor, is added to the matrix B. A new column, representing the fanout of the cube factor, is also added to the matrix B. The value for any entry covered by the rectangle is set to zero. On subsequent iterations, no value is recorded for covering one of these already covered points.

For kernel-intersection extraction, the kernels of the new expression are generated and included in the co-kernel cube-matrix. The value for all entries corresponding to a cube which is covered by the rectangle is set to zero.

The process of selecting a good rectangle and extracting the rectangle is iterated while the value of the rectangle returned by **choose_rectangle** is positive. Recall that the rectangle value for both cube extraction and kernel extraction is defined to be the number of literals saved in the network by extracting the rectangle; hence, the algorithm terminates when no factors further reduce the number of literals in the network.

3.7.5 covering_extract: Simultaneous Selection of Rectangles

An alternate approach to the greedy nature of **greedy_extract** is to find a minimum-weight rectangle-cover and then simultaneously extract all of the rectangles from the matrix. This algorithm is called *covering_extract* and is shown in Figure 3.8.

First, an optimal prime rectangle cover is found using the **rect_prime_cover**. The rectangles are then incrementally modified to obtain a cover with a smaller total cost. The incremental modifications delete redundant rectangles (**rect_irredundant**) and reduce

```
covering_extract(B) {
    P = rect_prime_cover(B);
    P = rect_irredundant(P, B);
    P = rect_reduce(P, B);
    extract the rectangles of P;
}
```

Figure 3.8: Algorithm **covering_extract**.

```
rect_prime_cover(B) {
    P = 0;
    while (there are uncovered points in B) {
        (R, C) = choose_rectangle(B);
        add (R, C) to the rectangle cover P;
        set V(i,j) = 0 for all i in R and j in C;
    }
    return P;
}
```

Figure 3.9: Algorithm **rect_prime_cover**.

the total cost by trimming the rectangles without leaving 1's in the matrix uncovered (**rect_reduce**). With the assumption that rectangle weight is positive and increases with increasing size of the rectangle, both of these operations reduce the total cost of the cover.

rect_prime_cover chooses prime rectangles using either **best_rectangle** or **ping_pong**. After a rectangle is added to the cover, the points in the rectangle are marked as covered. Subsequent rectangles take into account that no benefit is realized from covering these same points again. This is iterated until every 1 in B is covered by some rectangle.

rect_irredundant is a modification of the **rect_reduce** algorithm and hence **rect_reduce** is presented first. Both of these algorithms attempt to improve the cost

of the rectangles selected for the cover.

rect_reduce first counts the number of times each point in the matrix B is covered. Then the essential points for each rectangle are determined. For a given rectangle, if a point is covered only by that rectangle (i.e., the count on the number of times the point is covered is 1), then the row and column for that point are essential for the rectangle. After checking all points in the rectangle, the rectangle is replaced with its essential parts, and the counts are modified to reflect replacement by the new rectangle.

If a rectangle is completely covered by other rectangles, the essential parts of the rectangle will be empty, and *rect_reduce* will delete the rectangle. However, the order in which the rectangles are processed in *rect_reduce* is significant. The reduction of one rectangle may block the reduction or removal of a rectangle which is processed later. Hence a simple heuristic is used to order the rectangles before processing by sorting them in decreasing order by size.

One problem with **rect_reduce** is that if the rectangles are processed in the wrong order, a redundant rectangle may not be detected. Therefore, a simple modification of *rect_reduce*, called *rect_irredundant*, is used first to detect and remove all redundant rectangles. *rect_irredundant* determines the essential row and column sets for each rectangle, but only modifies a rectangle if its essential sets are empty – i.e., the rectangle is deleted. If the rectangle is not redundant, it is skipped and processing continues with the next rectangle. Although **rect_irredundant** is itself order-dependent, it at least guarantees that some of the redundant rectangles will be removed. More sophisticated heuristics for irredundant, such as those used in ESPRESSO, can also be applied to the rectangle covering problem.

The procedure *rect_cover* is analogous to a single pass of the *expand*, *irredundant*, *reduce* sequence of Espresso [19]. This operation can be iterated, as done in Espresso, by defining an *expand* procedure to expand each rectangle from an initial covering into a prime rectangle. This is then made irredundant and reduced with the reduced rectangles becoming the input to the first part for reexpansion. Iteration would continue until no decrease in weight is obtained. As in Espresso, this style of heuristic algorithm depends on finding good heuristics for choosing the direction for expansion, and the sequence in which the rectangles are reduced.

One advantage of using covering extraction rather than greedy extraction is that the collection of rectangles in the cover are providing information on the best set of simul-

```
rect_reduce(P) {

    /* Count how many times each point in B is covered */
    M = 0;
    foreach rectangle (R, C) in P {
        foreach row in R {
            foreach column in C {
                M[row][column] = M[row][column] + 1;
            }
        }
    }

    /* Check each rectangle for nonessential parts */
    foreach rectangle (R, C) in P {
        essential_R = 0;
        essential_C = 0;
        foreach row in R {
            foreach column in C {
                if (M[row][column] == 1) {
                    add row to essential_R;
                    add column to essential_C;
                }
            }
        }

        /* modify counts */
        foreach row in (R - essential_R) {
            foreach column in (C - essential_C) {
                M[row][column] = M[row][column] - 1;
            }
        }

        Replace (R,C) with (essential_R, essential_C);
    }
}
```

Figure 3.10: Algorithm rect_reduce.

taneous factors to remove from the matrix. Note that each pass of the covering extraction algorithm adds only a controlled number of levels of logic to the network. By solving the minimum weight rectangle-covering problem, it is possible to find a low cost solution which minimizes the increase in circuit depth.

3.8 Selective Collapse

The initial Boolean network has an initial set of common factors already identified. However, there is no guarantee that all of these factors are high-quality. Therefore *selective-collapsing* is performed on the initial Boolean network to provide a better starting point for decomposition. The goal of selective collapse is to remove those factors which provide little value to the current network, while retaining those factors which appear to be of high value.

In the limit, selective collapsing can reduce a network to two-level form. However, for many circuits, this is not a reasonable synthesis technique. Many functions cannot be represented efficiently in two-level form. For example, even a simple function such as comparison of two 32-bit values for equality ($\prod_{i=0}^{32} a_i \oplus b_i$) requires 2^{32} product terms in sum-of-products form. Arithmetic structures, such as n-bit adders, also have an exponential number of product terms (as a function of n) in their minimum two-level form. As another example, consider what happens when a multiplexor is placed in front of a function f and the composite function is collapsed to two-levels. Assume f has p product terms in its minimum two-level form. If n values are multiplexed into a single value, $2^n p$ product terms are needed for the minimum representation of the multiplexed function. This is a simple example where the initial network contains some factors which are valuable for representing the function. Therefore, blindly collapsing a network to two-level form often does not make sense.

Selective-collapsing is implemented by defining a value function for each node in the network. A node of low value is collapsed into all of its fanout. Collapsing one node into another is merely the process of representing the second node without using the first node; i.e., given $f(x_1, \ldots, x_n)$ and $g(f, x_1, \ldots, x_n, y_1, \ldots, y_m)$, determine G such that

$$G(x_1, \ldots, x_n, y_1, \ldots, y_m) = g(f(x_1, \ldots, x_n), y_1, \ldots, y_m).$$

G is uniquely defined, but its representation as a Boolean function is not; hence, a two-level

minimized representation is typically used for G.

This may increase the number of literals in the network, but it provides larger functions for decomposition, thus increasing the number of common factors. This heuristic is based on the idea that, in the worst case, the algebraic decomposition techniques will recover the initial factorization while an extra degree of freedom exists to explore alternate decompositions. Of course, because of the limited nature of algebraic decompositions, it is important to control the selective-collapse process. For example, the initial network may represent a nonalgebraic decomposition and selective-collapse followed by algebraic decomposition may fail to find the nonalgebraic factors.

Two value-functions are defined for a node. The first is the *sum-of-products value.* This is the amount by which the literals-in-sum-of-products form for the network would increase if the node were collapsed into all of its fanout. The sum-of-products value of node x with respect to fanout y can be estimated based on the cubes of y which depend on x and the sum-of-products size for x, and the cubes of y which depend on $\overline{x}$ and the sum-of-products size for $\overline{x}$. The sum-of-products value for x, written $sop(x)$, is then summed over each fanout of x.

The second value function is the *factored-form value.* This is the amount by which the literals-in-factored-form for the network would increase if the node were collapsed into all of its fanout. The factored-form value (fac) for a node x is estimated as

$$fac(x) = \left(\left(\sum_{y \in fanout(x)} N(x,y) \right) - 1 \right) (L(x) - 1) - 1.$$

where $N(x,y)$ is the number of times either x or $\overline{x}$ appears in the factored form for y, and $L(x)$ is the number of literals in the factored form for x. The justification for this definition is that there exists a factored form for y, after x is collapsed into it, which uses only $N(x,y)(L(x)-1)$ literals (each occurrence of the literal x in y is replaced with the factored form for x, and each literal $\overline{x}$ is replaced with the dual factored form for x). As a special case, notice that the factored-form value for any node with single fanout to an expression where it is used only once is -1, implying that eliminating the node actually saves literals in the network.

For both $sop(x)$ and $fac(x)$, the value of a node which feeds a primary output needs to be modified. After the node is collapsed into its fanout, it cannot be deleted; hence, the literals of the node itself are not counted as being saved in either $sop(x)$ or

```
eliminate(network, max_sop, max_fac) {

    while (some node is eliminated) {
        foreach node in network {
            if (sop(node) < max_sop && fac(node) < max_fac) {
                foreach fanout of node {
                    collapse node into fanout;
                }
                if (node does not feed a primary output) {
                    delete node from network;
                }
            }
        }
    }
}
```

Figure 3.11: Algorithm **eliminate**.

$fac(x)$.

Selective collapsing is performed by **eliminate** as shown in Figure 3.11. **eliminate** takes two parameters: the maximum sum-of-products value to be retained (*max_sop*), and the maximum factored-form value to be retained (*max_fac*). Any node which has a value less than both thresholds is eliminated from the network. This is repeated until all nodes have value greater than both thresholds.

3.9 Representation of Two-level Functions

When a network of m functions can be represented in two-level form, there are two Boolean networks which are natural candidates for starting the optimization.

The first form, called the *single-output network*, is created with m internal nodes. Each of the m functions is placed at one of the internal nodes. (Recall that each node in the Boolean network represents an arbitrary function; typically the function is represented in two-level sum-of-products form.) In this representation, no common terms are shared between the functions. Each of the functions is then minimized using a two-level minimization

algorithm to find the representation with a minimum number of literals. This provides a Boolean network with a minimum number of literals which, heuristically, is a good starting point for decomposition. Algebraic decomposition is then applied to find common multiple-cube divisors between the functions; when no multiple-cube divisors remain, the common single-cube divisors are found.

The second form, called the *multiple-output network*, is created from the multiple-output minimization of the m functions. Ideally the two-level minimization would minimize the total number of literals needed to write the equations; however, typical two-level multiple-output minimization programs, such as ESPRESSO, minimize the number of unique product terms and then minimize the number of literals for the representation with the fewest number of terms. Note that in the process of performing multiple-output minimization, single cube factors are determined which are shared between the functions.

A multiple-output minimized network is represented by a Boolean network with a cascade of NOR-gates. Each unique product term has a node in the Boolean network which is the NOR of the literals in the product term, and each output function has a node in the Boolean network which is the NOR of the product terms which sum to create that output. Inverters are added, as needed, to form the positive and negative literals of the primary inputs and at each output. NOR-gates are preferred for the representation because with NOR-gates, each node in the Boolean network is represented by a single cube function. Hence, only common-cubes are present in the network.

Example 3.9.1 *Given the equations:*

$$\begin{aligned} f_1 &= \bar{a}b\bar{c}d + \bar{a}bcd + a\bar{b}cd + ab\bar{c}\bar{d} + ab\bar{c}d + abcd \\ f_2 &= \bar{a}\bar{b}\bar{c}\bar{d} + \bar{a}\bar{b}\bar{c}d + \bar{a}\bar{b}cd + \bar{a}b\bar{c}d + \bar{a}bcd + a\bar{b}cd, \end{aligned}$$

the single-output form is:

$$\begin{aligned} f_1 &= ab\bar{c} + acd + bd \\ f_2 &= \bar{a}\bar{b}\bar{c} + \bar{b}cd + \bar{a}d \end{aligned}$$

and the multiple-output form is:

$$\begin{aligned} p_1 &= acd \\ p_2 &= \bar{b}cd \\ p_3 &= \bar{a}bd \end{aligned}$$

$$
\begin{aligned}
p_4 &= ab\bar{c} \\
p_5 &= \bar{a}\bar{b}\bar{c} \\
f_1 &= p_1 + p_3 + p_4 \\
f_2 &= p_2 + p_3 + p_5.
\end{aligned}
$$

There is little that can be said about which starting point is preferred for algebraic decomposition. For functions which are very efficient as a PLA (e.g., in the limit a ROM containing random data), the multiple-output network often leads to better decompositions. These functions are characterized by an output plane in the PLA which is dense. A large number of Boolean common-cube factors are detected by the multiple-output minimization which the algebraic techniques are not able to find. However, in most cases where the output plane is sparse, the single-output representation leds to superior decompositions. Results comparing the single-output form to the multiple-output form are presented in Section 3.10.

3.10 Experimental Results

The rectangle covering algorithms described in this chapter have been implemented in MIS-II. This section provides some experimental results from this implementation on a number of industrial circuits.

All of the results in this section were collected on a Sun 4/260 Computer. A Sun 4/260 is typically ten times faster than a DEC MicroVax-II for integer applications. All run-times are reported in seconds.

For many of the experiments, two measures of circuit quality are compared. The first measure is the number of literals-in-sum-of-products form for the optimized circuit. This is referred to simply as *Literals*. The second measure is the total gate area after technology mapping using the technology mapping algorithm described in Chapter 4. The total area of all of the cells in the mapped circuit is referred to as the *Area* of the circuit. The library used is the benchmark library defined for the *1987 International Workshop on Logic Synthesis* (IWLS). This library is called the IWLS-87 library and a description of this library is given in Chapter 4. For reference, a two-input NAND-gate in this library has an area cost of two.

			Literals			Area		
Example	In	Out	asis	no elim	elim	asis	no elim	elim
C432	36	7	335	335	267	358	358	368
C499	41	32	576	568	558	696	680	676
C880	60	26	648	623	421	607	576	570
C1355	41	32	992	984	558	1,048	1,032	676
C1908	33	25	1,058	925	543	1,007	907	672
C2670	233	140	1,570	1,342	764	1,423	1,187	936
C3540	50	22	2,221	1,985	1,385	2,080	1,955	1,639
C5315	178	123	3,531	3,011	1,838	3,418	2,861	2,296
C6288	32	32	4,705	4,675	3,762	4,722	4,692	4,214
C7552	207	108	4,750	4,155	2,332	4,437	3,972	2,875
Totals			20,386	18,603	13,227	19,796	18,220	15,875

Table 3.1: ISCAS circuit optimization results.

3.10.1 ISCAS Circuit Optimization

The 1985 ISCAS test generation circuits provide a source of large multiple-level logic networks. None of these ten examples cannot be efficiently represented in two-level form, so they provide a set of good test circuits for selective-collapse.

The following experiments were performed for these circuits. First, each circuit was mapped into the IWLS-87 library without applying selective-collapse or kernel and cube decomposition (*asis*). Then kernel and cube decomposition was applied to each circuit without selective-collapse (*no elim*). Finally, selective-collapse with a `max_sop` value of 100 and a `max_fac` value of 0 was applied followed by kernel and cube decomposition (*elim*). The results for both literal count and area are shown in Table 3.1.

The use of algebraic decomposition alone reduced the literal count by 9% and the mapped area by 8%. The combination of selective-collapse and algebraic decomposition reduced the literal count by 35% and the mapped area by 20%. Overall, total literal count is a reasonable predictor of the final mapped area, but there are some startling exceptions. For example, the literal count for C432 was reduced from 335 to 267, but the mapped circuit area increased slightly. As another example, the literal count for C880 reduced from 648 to 421, but the mapped area decreased only slightly (607 to 570).

The total run-time for selective-collapse and algebraic decomposition for all ten circuits was 12.5 minutes on a Sun 4/260.

Example	In	Out	Initial Literals	ping_pong Literals	ping_pong Area	ping_pong Sec	best_rect Literals	best_rect Area	best_rect Sec
apex1	45	45	2,887	1,314	1,514	14	1,304	1,482	858
apex2	39	3	15,531	3,996	3,909	193	3,901	3,766	3,408
apex3	54	50	3,342	1,566	1,900	15	1,631	1,979	1,867
apex4	9	19	5,438	2,219	2,652	30	2,180	2,615	14,179
apex5	117	88	7,369	3,798	3,724	43	3,779	3,686	343
seq	41	36	3,497	1,268	1,421	14	1,254	1,411	564
Totals			38,064	14,161	15,120	309	14,049	14,939	21,219

Table 3.2: Comparison of ping-pong and best-rectangle.

3.10.2 Comparison of best_rectangle and ping_pong

An experiment was performed to compare **best_rectangle** and **ping_pong**. The rectangle matrices chosen for the experiment are the cube-literal matrices for the multiple-output form of six large functions. Five of these examples are circuits designed at AT&T and the sixth comes from a chip design at Berkeley. Each of these functions was first minimized in multiple-output form by ESPRESSO, and then converted to a two-level network of NOR-gates. The resulting cube-literal matrices are dense and provide difficult examples for common-cube extraction. The **greedy_extract** algorithm was used in two forms: first using **best_rectangle** to find a rectangle at each step, and then using **ping_pong** to find a rectangle at each step.

The results of the comparison are given in Table 3.2. Shown in the table are the number of inputs and outputs for each circuit and the number of literals in the initial multiple-output form Boolean network. The initial number of literals is after a multiple-output minimization by ESPRESSO. The next three columns give the optimized literal count, optimized gate area, and run-time for cube extraction using **ping_pong** followed by the same data collected using **best_rectangle**. No bounding was used for **best_rectangle**. Sparse-matrix techniques were used for both algorithms. The results indicate that **best_rectangle** finds slightly better solutions (.8% in literal count and 1.2% in gate area) but is much more costly in terms of execution time (69 times longer). On *apex3*, **ping_pong** found a slightly better circuit. Overall, **ping_pong** appears to offer the best trade-off between execution time and the quality of the optimized circuit.

Example	Final Literals	No Bound Sec	No Bound Rect	Simple Bound Sec	Simple Bound Rect	Best Bound Sec	Best Bound Rect
apex1	1,304	858	116,695	744	101,554	750	95,637
apex2	3,901	3,408	660,948	2,234	347,870	2,426	314,841
apex3	1,631	1,867	111,398	1,708	93,663	2,052	89,661
apex4	2,180	14,179	281,116	12,052	194,312	16,166	186,238
apex5	3,779	343	77,113	303	63,823	337	58,988
seq	1,254	564	106,683	469	89,689	419	83,499
Total	14,049	21,219	1,353,953	17,510	890,911	22,150	828,864

Table 3.3: Comparison of bounding strategies.

3.10.3 Comparison of Bounding Strategies

The experiments from the previous section were repeated using **best_rectangle** with the bounding strategies described in Section 3.7.2. The results are given in Table 3.3. The first columns correspond to no bounding applied; the middle column corresponds to the simple bound based on the length of the longest row and column; and the final column is the bounding technique based on sorting the rows (columns) of the matrix by the number of elements in each row (column). Shown in the table are the number of rectangles evaluated during the cube extraction and the run-time for each strategy. The final result returned by each bounding strategy is the same.

It is evident that the simple bounding algorithm reduces both the number of rectangles examined and the total run-time. Application of this technique reduces the number of rectangles examined by 35% and reduces the total run-time by 21%. The better bounding strategy reduces the number of rectangles examined by 39%; unfortunately, application of this bounding strategy is expensive. In fact, the total run-time increases by 4% when this strategy is used.

3.10.4 Comparison of Single-Output and Multiple-Output Optimization

The same set of six circuits, because they are represented in PLA form, are also good examples to demonstrate the effects of single-output minimization versus multiple-output minimization. The multiple-output minimization results for these circuits have already been presented. These correspond to a multiple-output minimization of the functions using ESPRESSO, followed by a single application of cube extraction (using **best_rectangle** to

Example	In	Out	Single-Output Literals	Single-Output Area	Multiple-Output Literals	Multiple-Output Area
apex1	45	45	2,124	2,686	1,304	1,482
apex2	39	3	410	533	3,901	3,766
apex3	54	50	1,956	2,483	1,631	1,979
apex4	9	19	2,883	3,733	2,180	2,615
apex5	117	88	1,171	1,452	3,779	3,686
seq	41	36	1,721	2,143	1,254	1,411

Table 3.4: Comparison of single-output and multiple-output optimization.

find each factor). The single-output minimization results were collected by minimizing each function individually, and then performing kernel extraction (using all kernels and **best_rectangle** to find each factor), and cube extraction (again, using **best_rectangle**). The results are presented in Table 3.4. Shown are the number of literals and mapped circuit area for each circuit after single-output minimization and multiple-output minimization.

As mentioned previously, the effects of single-output versus multiple-output minimization are difficult to predict. In four circuits, multiple-output minimization provides the smallest circuit; for two circuits, single-output minimization was better. When one technique wins, it tends to win by a large margin (e.g., *apex2* is seven times smaller when single-output minimization is used; *apex1* is almost twice as small when multiple-output minimization is used).

3.11 Conclusions

This chapter has shown how to unify the problems of kernel extraction and common-cube extraction in algebraic decomposition as instances of the rectangle covering problem. Efficient heuristic algorithms have been developed for these problems based on rectangle covering, and extensions which make use of don't-care entries in the rectangle matrices allow for a limited form of non-algebraic decomposition. Experimental results from an implementation of these algorithms demonstrate that optimization of large circuits is handled efficiently.

Several interesting problems based on the rectangle covering problem remain. These are presented below.

Recall that **best_rectangle** is potentially exponential in the size of a matrix. This worst-case is rarely encountered in algebraic decomposition, probably as a result of

the sparsity of the matrices which are encountered. However, the question remains whether a polynomial-time algorithm to find the maximum-valued prime rectangle exists. For the special case of a value function which measures the perimeter of the rectangle, a polynomial-time algorithm does exist (see the comment for problem GT24 of [31]).

A related problem is finding an efficient algorithm to find the maximum-valued nonprime rectangle in a matrix. Application of **best_rectangle** followed by a reduction of the rectangle to a nonprime rectangle does not guarantee finding the best-valued nonprime rectangle. Again, the question is whether this problem is NP-complete.

Finally, two open problems are characterizing the limits of algebraic decomposition and developing an optimum algorithm for algebraic decomposition.

Chapter 4

Technology Mapping

Technology mapping is the process of implementing a set of Boolean equations by selecting logic gates from a library of available gates. The goal is to make optimal use of all of the gates in the library to produce a circuit with critical path delay less than a target value and minimum total area.

In this thesis, a solution to the technology mapping problem is formulated as the solution to a covering problem. A system of Boolean equations is transformed into a directed acyclic graph (DAG) constructed from basic components and this graph is covered using pattern graphs derived from the gates in the library. An algorithm for solving the DAG-covering problem is presented. However, because of the difficulty in finding exact solutions to DAG-covering, a heuristic algorithm is presented which partitions the graph into a forest of trees and uses a dynamic programming algorithm to solve the covering problem on each tree. This heuristic algorithm is called the tree-covering approximation to DAG-covering.

The approach of DAG-covering and the tree-covering approximation were first proposed for technology mapping by Keutzer [49]. This thesis provides an extension of the techniques proposed by Keutzer in several ways. First, an algorithm for DAG-covering is presented based on a transformation to the binate-covering problem. Within the tree-covering approximation, extensions are given for minimum area optimization under a fixed delay constraint and techniques for generating the tree-covering patterns from a library description are provided. Other extensions include a finer-resolution base-function set to improve the quality of optimization and improved heuristics for inverter optimization.

This chapter is organized as follows. An introduction to technology mapping and a review of previous approaches to this problem are presented. The optimization

cost function and the characteristics of a technology library are then introduced. The approach to technology mapping using DAG-covering and an exact algorithm for solving the DAG-covering problem are described. This is followed by a description of the tree-covering approximation for DAG-covering. Application of the tree-covering approximation to optimization for complete complex-gate CMOS libraries is then considered. The tree-covering algorithm described here has been implemented as part of the logic optimization program MIS [20], and results for several industrial circuits and standard benchmark circuits are presented.

4.1 Introduction

The optimization goal for technology mapping is to minimize the total area of the circuit while satisfying a maximum critical-path delay. The area of the circuit is taken as the sum of the areas for all gates in the circuit, and the critical-path delay is the longest path through the circuit using static delay analysis.

This statement of the technology mapping problem appears to be a re-statement of the general logic synthesis problem. The primary difference is the assumption that the Boolean equations have undergone technology-independent optimization; it is assumed that the equations provide a good structure for the final circuit. The role of technology mapping is to finish the synthesis of the circuit by performing the final gate selection from a particular library. The algorithms chosen for technology mapping are simplified because they can be constrained by the structure of the equations produced by the technology-independent optimizations. It is not the role of technology mapping to change the structure of the circuit radically, for example, by finding common subexpressions between two or more parts of the circuit. Likewise, it is not the role of technology mapping to reduce the number of levels of logic along the critical path. The role of technology mapping is the actual gate choice to implement the equations – for example, choosing the fastest gates along the critical path, and using the most area-efficient combination of gates off the critical path.

There are several characteristics which are desirable for a technology mapping algorithm. These are:

1. Adapt easily to different libraries.
2. Support irregular collections of logic functions.

3. Handle detailed technology-dependent cost functions.
4. Efficient execution time.

These points are elaborated below.

First, it is desirable that the technology mapping algorithm be able to adapt to a variety of different libraries with minimal effort. This is difficult because many libraries have an irregular collection of logic functions available as primitives. An algorithm which depends on characteristics of a particular library (for example, availability of a complete set of CMOS and-or-invert gates) is of limited use. Also, an algorithm which is geared to a subset of the gates in a library is limited in its optimization potential. To achieve the goal of library adaptability, an approach to technology mapping should be *user-programmable*. The user should be able to provide new gates to the technology mapper without understanding its detailed operation, and these gates should be used effectively.

During technology mapping, simple cost functions such as transistor count or levels of logic will not provide high-quality circuits. Instead, it is necessary to consider more detailed models for the cost of a gate in the actual target technology. This detailed level of modeling, coupled with gates which have irregular area and delay cost functions, greatly complicates the technology mapping process.

As a simple example, consider using the LSI Logic Corporation Logic Compacted Array library [2] to build a fast 32-input AND-gate. Choosing from among the eight inverters, the two-, three-, four-, five-, six- and eight-input NAND-gates and NOR-gates, and the two-, three-, and four-input AND-gates and OR-gates to build a fast circuit for this simple function is a challenging task. The problem is that the gate costs are irregular and sometimes counter-intuitive – the relative delay between a two-input NAND-gate and the four-input NAND-gate is much different than the relative delay between a four-input NAND-gate and an eight-input NAND-gate; in some situations a two-input NAND-gate followed by an inverter is faster than a two-input AND-gate, and in other situations the opposite is true. All of this is complicated by different delay values based on whether the output of a gate is rising or falling and the total size of the circuit. A fixed architecture for a 32-input AND-gate, such as a tree of two- or four-input AND-gates, cannot be optimal for every design or every library.

Therefore, to provide high-quality results for different libraries and circuits, a technology mapping algorithm must make few assumptions about the relative cost and performance of the gates in a library, and must be prepared to model accurately the cost

functions which are optimized.

The economics of integrated circuit fabrication dictate that it is worth a large amount of computer time to find a better circuit. Therefore, while it is always desirable to have an efficient algorithm, the execution performance of the technology mapping algorithm is less important than the quality of the final result. This is true for the last optimization of a circuit before fabrication. However, the steps of technology-independent optimization and technology mapping are iterated by a logic synthesis system if the performance goals are not initially met. Technology mapping in this case operates as an accurate predictor for the quality of a technology-independent representation and these results are fed back to the technology-independent optimization to improve the final implementation. Therefore, it is desirable that a technology mapping algorithm support a fast execution mode as well as a slower mode which provides higher-quality optimization.

4.2 Previous Work

4.2.1 Rule-Based Techniques

One technique proposed for solving the technology mapping problem uses *local transformations* or *rule-based systems*. This includes the logic synthesis systems LSS [25], Socrates [36] and LORES [45]. LSS uses a rule-based approach, but the reported emphasis has not been on technology mapping *per se*, but instead on technology independent optimizations; however, technology-dependent optimization is also a part of their system [46]. Both LSS and Socrates use rule-based techniques as a part of a larger optimization system. For example, Socrates uses two-level minimization and algebraic decomposition as part of its optimization strategy, and LSS uses cube-factoring, global-flow, and other high-level transformations. This section will focus only on the technology mapping aspects of these systems.

A rule-based system is a collection of rules and techniques for selecting when and where to apply a rule to improve the circuit quality. Each rule is expressed as a pair (*target graph*, *replacement graph*). A rule is applied by identifying a portion of the circuit which contains a subgraph isomorphic to the target graph, and replacing the subgraph with the replacement graph. Each rule application preserves the circuit functionality. Technology mapping from Boolean equations starts with a straightforward translation of the equations

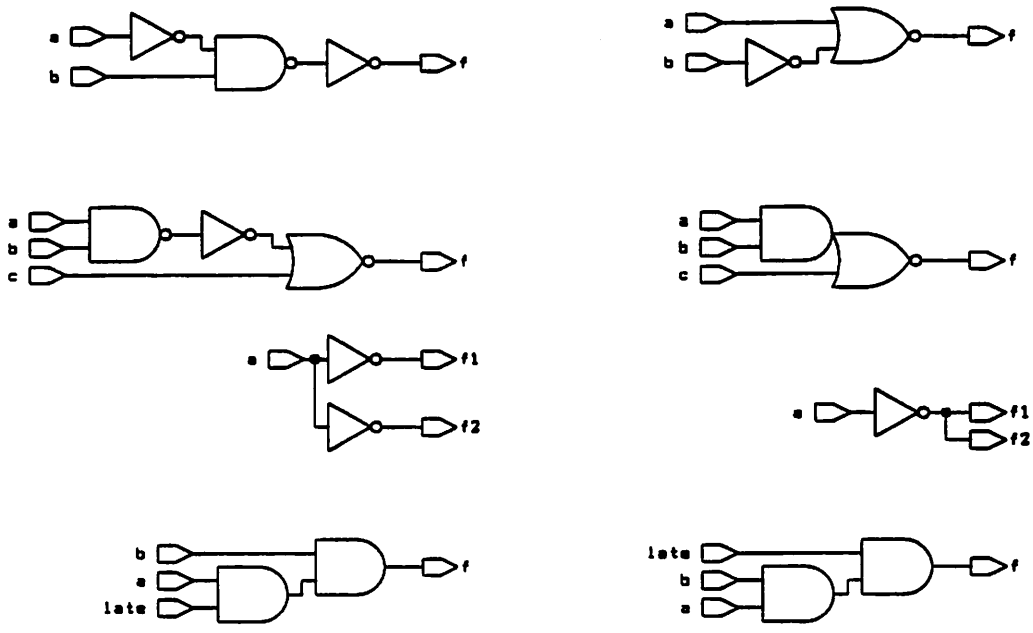

Figure 4.1: Example rules from Socrates.

into gates in the library (for example, using only AND-gates and OR-gates), and the circuit quality is improved through the iterative application of rules. Example rules from Socrates are shown in Figure 4.1.

The operation of a rule-based system can be understood by viewing the optimization as a search on a *state-space graph*. The state-space graph is a directed graph where the nodes represent legal circuit configurations for the desired functionality, and a directed edge exists from node v_i to v_j if a rule application can transform the circuit of node v_i into the circuit of node v_j. Each node in this graph has an associated cost based on the area and delay of the corresponding circuit. The optimization starts from an arbitrary node in the state-space graph, and each application of a rule is a move to an adjacent node. The goal is to find the minimum cost node in the state-space graph.

The difficultly in applying rule-based techniques for optimization is solving the problem of searching the state-space graph for the minimum cost node. The approach reported in LSS and LORES is greedy. The edges from the current node are examined in

a predetermined order and the first edge which improves the value of the cost function is selected. The new node is taken as the current node, and the process is repeated until a local-minimum is reached. A local-minimum node has no out-going edge that improves the value of the cost function.

In Socrates, a search technique replaces the greedy strategy. The search starts from the current node and examines a fixed number of adjacent nodes (the *breadth* of the search). For each of these adjacent nodes, the search is repeated up to a fixed distance from the current node (the *depth* of the search). The best move seen in this set of nodes surrounding the current node is taken as the next node, and the search is continued. A sequence of rule applications is accepted if the cost function at the end of the sequence is improved. This allows intermediate rule applications to temporarily increase the cost function. A state-space search is very expensive in terms of execution time unless the search parameters are controlled. In Socrates, a *meta-level rule-based expert system* [36] controls the breadth and depth of the search at each step and chooses the rules to apply. Results are presented showing a 12% improvement in area optimization results for the partial state-space search over the greedy approach mentioned earlier.

The primary advantages of a rule-based system are its flexibility in the types of rules and cost-functions that can be considered during the optimization, and the relative ease with which the core transformation system can be developed. The primary drawbacks of this approach are the difficulty in creating, maintaining, and modifying the rule-base, and the difficulty of incorporating new gates into a library. Also, rule-based systems tend to be expensive in terms of execution time and offer an unpredicatable result quality. These points are elaborated below.

The execution time of a rule-based system is determined by the number of nodes in the state-space graph which are examined. Examining each node requires a computation of the cost function, including the area and delay of the corresponding circuit. Even using incremental techniques, computation of the delay cost function is expensive (on the order of seconds for a 1-MIP computer on a large circuit [35]). This problem is aggravated by a bounded state-space search where the number of cost function evaluations grows exponentially in the depth of the search. Therefore, the cost of a rule-based approach is especially expensive if searching is used to provide high-quality results.

The quality of the circuits produced by a rule-based system depends on the completeness of the set of rules, and the quality of the heuristics which guide the walk on the

search space graph. If the rules are not complete, the state-space graph may not be connected, leading to the impossibility of reaching particular solutions. Using a greedy search and a predetermined order to evaluate adjacent nodes has the problem of becoming stuck in a local optimum which is far away from the global optimum. It is not clear if a limited state-space search adequately avoids suboptimal local minima.

Another problem with rule-based systems is that it is difficult to incorporate new gates into a library. A technique which provides technology portability is to define a master gate library, and to write all rules in terms of this library. It is then assumed that all libraries will be a subset of this master library. If a library has a gate which is not in the master library, it cannot be involved in the optimization process unless additional rules are added. This difficulty is compounded further by the interaction between the set of rules and the heuristics which control the rule application. For this reason, rule sets are hand-crafted with a particular technology and design style in mind. Adding a new rule is not simply a matter of adding the rule to the set of transformations; it is also necessary to consider how the new rule will interact with the other rules in the system. In the limit, it may be necessary to re-write the heuristics which control the order in which the rules are applied. For this reason, there is a significant effort to port the system to new libraries and technologies. A final problem is the large number of transformations which are required to provide high-quality optimization. Writing, managing and verifying all of these rules is a nontrivial process.

Despite these problems, local transformation techniques have demonstrated the ability to produce high-quality results. For example, LSS Socrates, and LORES all report optimization results competitive with human designers [25,36,45].

4.2.2 DAGON

The approach of using DAG-covering for technology mapping in logic synthesis was first proposed by Kurt Keutzer of AT&T Bell Laboratories in his program DAGON [49]. His thesis was that technology mapping for logic synthesis is closely related to the problem of code generation for programming language compilers, and hence the advanced techniques that have been developed for code generation should be applicable to technology mapping.

DAGON is a technology mapping program written on top of the tree manipulation tool *twig* [73]. *twig* was originally developed to provide a flexible framework for building

efficient algorithms for tree matching and for solving the tree-covering problem. *twig* uses the Aho-Corasick [5] string-matching algorithm for matching and the Aho-Johnson [6] dynamic programming algorithm for optimal tree covering.

The approach advocated in this chapter is based on the ideas of Keutzer.

4.3 Technology Libraries

Technology mapping starts with an optimized logic network, and a description of the available gates. The information required for technology mapping is described in this section.

Definition 4.3.1 *A* gate *is the cell primitive for constructing a digital circuit. It is described by the combinational Boolean function which it implements and its impact on the area and timing cost functions for the circuit.*

A gate is a cell or primitive which is available in a particular implementation design style or technology. For example, the gates in static CMOS gate-array and standard-cell designs typically include NAND-gates, NOR-gates, and a variety of and-or-invert gates, whereas the gates in an ECL gate-array are typically dual-output NOR-gates and exclusive-or gates.

Definition 4.3.2 *A* technology library *is a collection of gates.*

The gates available for technology mapping are described by a technology library. Ideally, technology mapping operates directly from the description of a technology library making no *a priori* assumptions on the types of gates which are available in the library or the relative cost and performance of different gates.

Several assumptions are made with this model of a technology library. These are:

1. *A gate's behavior is captured by a combinational Boolean logic function.*
2. *The effect of physical design on the performance of a design is included in the cost function of the gates.*
3. *A technology library is a finite collection of gates.*
4. *Every interconnection of gates yields a valid circuit.*

The assumption that *the gate behavior is captured by a combinational Boolean logic function* implies that the gates are unidirectional and memoryless. For example, this definition excludes a CMOS transmission gate as a gate in a technology library. However, it does not exclude larger functions built from transmission gates, as long as these functions are described by a logic function and have their cost function information modeled appropriately. This assumption also excludes noncombinational elements such as latches and tri-states from the technology library. Extensions which allow for sequential technology mapping are proposed in Section 4.8.1.

The assumption that *the effect of physical design (primarily routing area and wiring capacitance) can be characterized and included in the cost function* is the key which allows for a separation of the logical design and physical design problems. Treated separately, logic design and physical design are each difficult optimization problems. While a system for solving both problems simultaneously could improve the overall quality of the final design, it is extremely difficult to solve the composite problem. Further, it is believed that predicative characterization of the effects of physical design during logic design is a valid approximation, especially in a gate-array environment.

The assumption that *a library is described by a finite collection of gates* would seem to exclude a complete complex-gate library, such as the set of all CMOS complex-gates with no more than four transistors in series from either supply to the output. However, as is shown in Section 4.6, it is reasonable to treat many complete complex-gate libraries as a fixed library.

Finally, the assumption that *every interconnection of gates yields a legal circuit* simplifies the description of the technology mapping algorithm. Examples of where this assumption fails for CMOS design include disallowing a cascade of gates constructed from transmission gates or enforcing that a gate be used only with complementary inputs. This assumption fails for ECL design when signals are required to connect to inputs with a compatible voltage level, or where a violation of the maximum fanout of a gate leads to an incorrect circuit. Resolving these problems poses no severe difficulties for the algorithms presented in this thesis.

4.3.1 Area Model

Each gate is assigned a cost function parameter which is the area of the gate. The area for the circuit is computed as the sum of the areas for all gates in the circuit. The goal for technology mapping is to minimize the area of a circuit; or, given two circuits with equal areas, the secondary goal is to minimize the total number of cells in the design.

To capture the effect of physical design, the area model includes an approximate area cost for wiring. For example, in standard-cell and compacted-array design, the average routing area is correlated to the number of pins on a net and the number of cells in the design. Hence, for these implementation styles, the wiring area can be estimated by adding an area penalty for each net.

4.3.2 Delay Model

For delay modeling, a static timing model is used to measure the delay of a logic network. The goal of the timing model is to provide an efficient estimate of the delay through the logic network.

The timing for a gate is modeled with a timing arc from each input pin to each output pin. The arc from input pin i to output pin j is parameterized with a *constant delay* α_{ij} and a *load-dependent delay* β_{ij}. Each input pin i of a gate is parameterized with an *input load* γ_i. The load driven by the output j (γ_j) equals the sum of the input load for all pins driven by the output. Using these parameters, the delay from input i to output j is modeled as $\alpha_{ij} + \beta_{ij}\gamma_j$.

A *timing graph* is constructed from the logic network using a node for each net in the logic network. The set of arcs between the inputs and outputs of a gate become edges in the timing graph between the corresponding input nets and output nets with a scalar weight on the edge giving the delay of the arc. Given that the graph is acyclic, a linear-time longest-path algorithm on the timing graph determines the signal *arrival time* and signal *required time* for all nets.

More precisely, the *arrival time* at each node is defined as:

$$A(x) = \max_{y \in i(x)} (A(y) + w(y, x))$$

where $i(x)$ is the set of input edges to node x, and $w(y, x)$ is the delay edge from node y to node x. For a source node z (i.e., a primary input), the input arrival time, $A(z)$, is supplied

to the system.

The *required time* at each node is defined as:

$$R(x) = \min_{y \in o(x)} (R(y) - w(x,y))$$

where $o(x)$ is the set of output edges from node x. For a sink node z (i.e., a primary output), the output required time, $R(z)$ is supplied to the system.

A simple extension to this model allows for separate rise and fall delays by providing parameters α_{ij}^r and α_{ij}^f for the rising and falling constant delay, and β_{ij}^r and β_{ij}^f for the rising and falling load-dependent delay. Each arc is labeled with the relationship between the sense of the input signal and the related sense of the output signal (i.e., whether the output rises, falls, or is indeterminate when a rising signal occurs at the input).

Parameters to represent the drive capability of the primary inputs of the logic network and the load presented to the primary outputs can be included to model the timing behavior of this design as part of a larger design.

The effect of physical design on delay is modeled by including extra load on the net driven by each output of the gate (i.e., by increasing γ_j). Similar to the area penalty for a net, the load penalty for a net is predicted from the size of the logic block and the number of pins on the net.

This style of timing verification has become popular for predicting the performance of digital CMOS integrated circuit modules which are implemented using gate-array and standard-cell design styles. The same model is often used both before and after placement and routing. For example, commercial gate-array vendors use a similar model for static timing analysis of pre-layout designs and provide back-annotation of layout capacitances into the same model for post-layout timing verification.

4.3.3 Technology Library Example

Most of the results presented in this chapter use a technology library defined for benchmark purposes at the 1987 International Workshop on Logic Synthesis (IWLS-87). Figure 4.2 is a description of this library in the format accepted by MIS-II.

The keyword GATE introduces a gate and is followed by the gate name. The number which follows is the gate area. The logic equation for the gate is given next using standard Boolean logic notation. The delay values for each input pin are introduced with

```
GATE inv1     1    O=!a;                PIN * INV 1 999 0.9 0.3 0.9 0.3
GATE inv2     2    O=!a;                PIN * INV 2 999 1.0 0.1 1.0 0.1
GATE inv3     3    O=!a;                PIN * INV 3 999 1.1 0.09 1.1 0.09
GATE inv4     4    O=!a;                PIN * INV 4 999 1.2 0.07 1.2 0.07
GATE nand2    2    O=!(a*b);            PIN * INV 1 999 1.0 0.2 1.0 0.2
GATE nand3    3    O=!(a*b*c);          PIN * INV 1 999 1.1 0.3 1.1 0.3
GATE nand4    4    O=!(a*b*c*d);        PIN * INV 1 999 1.4 0.4 1.4 0.4
GATE nor2     2    O=!(a+b);            PIN * INV 1 999 1.4 0.5 1.4 0.5
GATE nor3     3    O=!(a+b+c);          PIN * INV 1 999 2.4 0.7 2.4 0.7
GATE nor4     4    O=!(a+b+c+d);        PIN * INV 1 999 3.8 1.0 3.8 1.0
GATE and2     3    O=a*b;               PIN * NONINV 1 999 1.9 0.3 1.9 0.3
GATE or2      3    O=a+b;               PIN * NONINV 1 999 2.4 0.3 2.4 0.3
GATE xor      5    O=a*!b+!a*b;         PIN * UNKNOWN 2 999 1.9 0.5 1.9 0.5
GATE xnor     5    O=a*b+!a*!b;         PIN * UNKNOWN 2 999 2.1 0.5 2.1 0.5
GATE aoi21    3    O=!(a*b+c);          PIN * INV 1 999 1.6 0.4 1.6 0.4
GATE aoi22    4    O=!(a*b+c*d);        PIN * INV 1 999 2.0 0.4 2.0 0.4
GATE oai21    3    O=!((a+b)*c);        PIN * INV 1 999 1.6 0.4 1.6 0.4
GATE oai22    4    O=!((a+b)*(c+d));    PIN * INV 1 999 2.0 0.4 2.0 0.4
GATE zero     0    O=CONST0;
GATE one      0    O=CONST1;
```

Figure 4.2: IWLS-87 benchmark library in MIS-II format.

the keyword PIN followed by the pin name. The delay values are the pin sense for the rise-fall delay model (INV which means the output falls when the input rises, NONINV which means the output falls when the input falls, or UNKNOWN which means the sense of the output is indeterminate based on a single changing input), the input pin load γ_i, the maximum capacitance that the gate is allowed to drive, and the parameters α_i^r, β_i^r, α_i^f, and β_i^f for this input pin to the output. The pin name "*" is allowed as a wild-card to specify identical delay values for all input pins. The CONST0 and CONST1 gates have been added to allow for technology mapping on circuits with gates or primary outputs driven by a constant logic value.

The MIS-II technology library format is limited to describing single-output gates. This is a result of the orientation of the technology mapping algorithms towards gates with one output. Techniques for using multiple-output gates in technology mapping are considered in Section 4.8.2.

4.4 Technology mapping using DAG-Covering

The problem of code generation in a compiler is to map a set of expressions onto a set of machine instructions for the target machine. Extensive research into compilers has led to efficient ways of formulating and solving this problem [6]. Each machine instruction is decomposed into a directed, acyclic graph (DAG) of atomic operations, called a pattern. Each instruction has a cost associated with it which represents the relative cost, in execution time, of chosing that instruction. The sequence of high-level expressions is also represented by a DAG of atomic operations. The optimal code generation problem is equivalent to finding an optimum cost cover of the subject DAG by the pattern DAG's.

A similar approach is taken in this chapter for the technology mapping problem. A set of base functions is chosen such as a two-input NAND-gate and an inverter. The logic equations are optimized in a technology-independent manner and are then converted into a graph where each node is restricted to one of the base functions. This graph is called the *subject graph*. The logic function for each library gate is also represented by a graph where each node is restricted to one of the base functions. Each graph for a library gate is called a *pattern graph*. For any given logic function there are many different representations of the function using the base function set. Therefore, each library gate is represented by many different pattern graphs.

```
t1 = a + b c;
t2 = d + e;
t3 = a b + d;
t4 = t1 t2 + f g;
t5 = t4 h + t2 t3;
F = t5';
```

Figure 4.3: Unoptimized set of logic equations.

```
t1 = d + e;
t2 = b + h;
t3 = a t2 + c;
t4 = t1 t3 + f g h;
F = t4';
```

Figure 4.4: Optimized set of logic equations.

For example, Figure 4.3 shows an unoptimized set of logic equations consisting of sixteen literals. Using technology-independent optimization, these equations are optimized into the form shown in Figure 4.4 using only fourteen literals. Figure 4.5 shows a subject graph for the optimized set of equations.

Figure 4.6 shows a set of pattern graphs for the IWLS-87 library presented in Figure 4.2. Most of the graphs for the gates in this library are trees of two-input NAND-gates and inverters. It is sufficient to describe most CMOS logic gates in this form. This includes, for example, all CMOS complex-gates (i.e., series-parallel connections of transistors for a pull-up and pull-down network). In the IWLS-87 library, the only exceptions are the two-input exclusive-or and two-input exclusive-nor gates. An algorithm to generate these patterns from the gate logic function is presented in Section 4.6.3.

The technology mapping problem is viewed as the optimization problem of finding

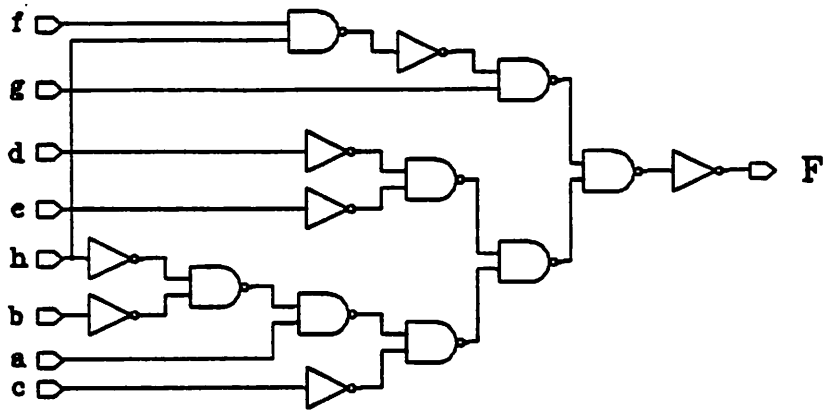

Figure 4.5: Subject graph for the equations of Figure 4.4.

a minimum cost covering of the subject graph by choosing from the collection of pattern graphs for all gates in the library. A *cover* is a collection of pattern graphs such that every node of the subject graph is contained in one (or more) of the pattern graphs. The cover is further constrained so that each input required by a pattern graph is actually an output of some other pattern graph.

For area optimization, the cost of the cover is defined as the sum of the areas of the individual gates. For minimum delay optimization, the cost of a cover is defined as the critical path delay of the resulting circuit using the delay model presented earlier. For the more typical optimization problem of optimizing for minimum area under a given timing constraint, any cover which results in a circuit with critical path delay greater than that allowed for any output is considered an illegal cover; then, the minimum-area legal cover is the optimization goal. If there are no legal covers, the cover of minimum delay is considered the desired solution.

A trivial covering for the subject graph from the previous example uses 8 two-input NAND-gates and 7 inverters for an area cost of 23. Figure 4.7 shows a covering for the subject of Figure 4.5 with an area of 18; an alternate covering is shown in Figure 4.8 with an area of 15.

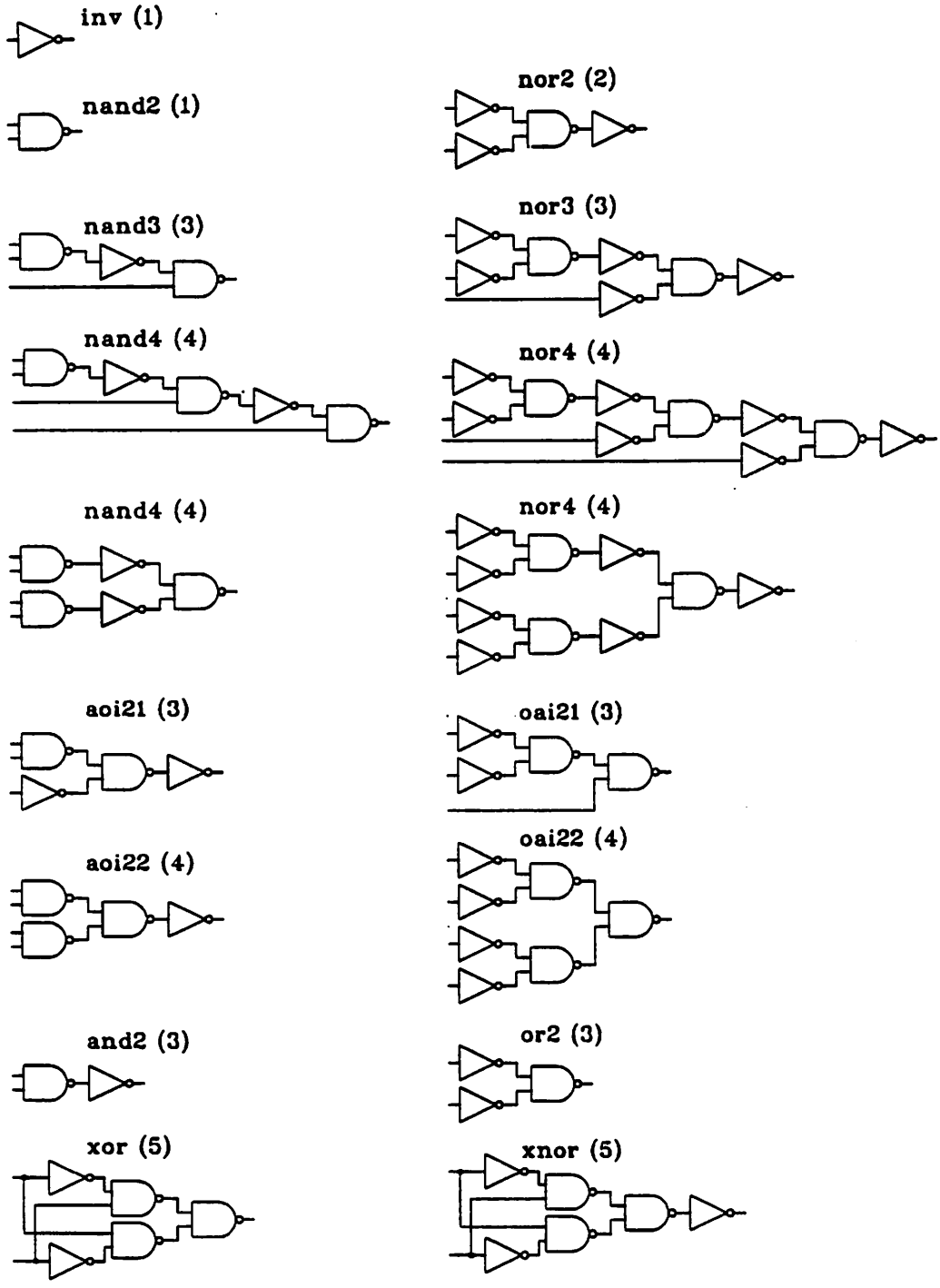

Figure 4.6: Pattern graphs for the IWLS-87 library.

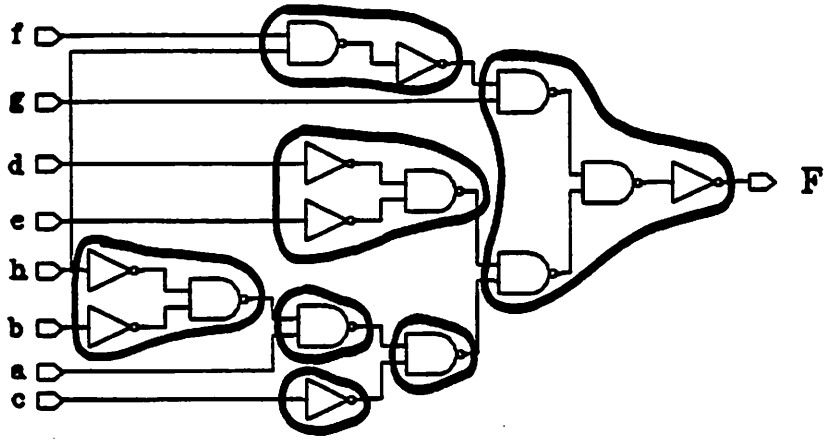

Figure 4.7: A cover for the subject graph of Figure 4.5.

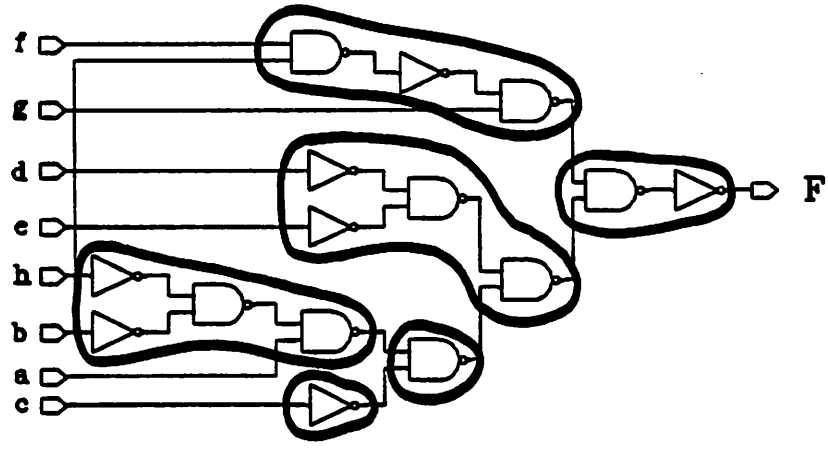

Figure 4.8: A second cover for the subject graph of Figure 4.5.

4.4.1 Choice of Base Functions

The choice of the base-function set is an important consideration for DAG-covering. It should be clear that this choice is arbitrary as long as the base function set is functionally complete. However, this decision does influence the number of patterns needed to represent the gates in a library and the quality of the solution provided by DAG-covering. The goal is to find the base-function set which provides the highest level of optimization and produces a small set of patterns. In this thesis, a base-function set of a two-input NAND-gate and an inverter is used. This choice is motivated by the following propositions.

Proposition 4.4.1 *Use of a two-input* NOR*-gate and inverter as a base-function set does not change the optimization potential of* DAG*-covering.*

More specifically, the set of covers for a subject graph using a two-input NAND-gate and inverter are the same as the set of covers for a subject graph using a two-input NOR-gate and inverter. This assumes that each NAND-gate in the first subject graph is mapped into a NOR-gate in the second subject graph, and similar patterns are used in each case.

Proposition 4.4.2 *A base-function set which includes both a two-input* NAND*-gate and a two-input* NOR*-gate increases the number of patterns required to represent each gate in the library without improving the optimization potential of* DAG*-covering.*

In a similar fashion, the set of covers resulting from the use of a two-input NAND-gate and a two-input NOR-gate is the same as the set of covers resulting from the use of just a two-input NAND-gate. Again, this assumes that each NAND-gate and NOR-gate in the first subject is mapped into a NAND-gate in the second subject graph. The inverters change between the two graphs, but the set of covers is the same. Similar arguments reject including other two-input functions (e.g., AND-gates and OR-gates) in the base-function set.

When both a NAND-gate and NOR-gate are used in the base-function set, the number of patterns to represent some functions increases. For example, using both a two-input NAND-gate and a two-input NOR-gate, a large number of pattern graphs are required for all representations of the gate $f = \overline{ab + cd}$. Variations such as three NAND-gates (with inverters), three NOR-gates (with inverters), and other representations using both NAND-gates and NOR-gates are possible patterns for this gate. Using only the two-input NAND-gate reduces the number of patterns to one as shown in Figure 4.6).

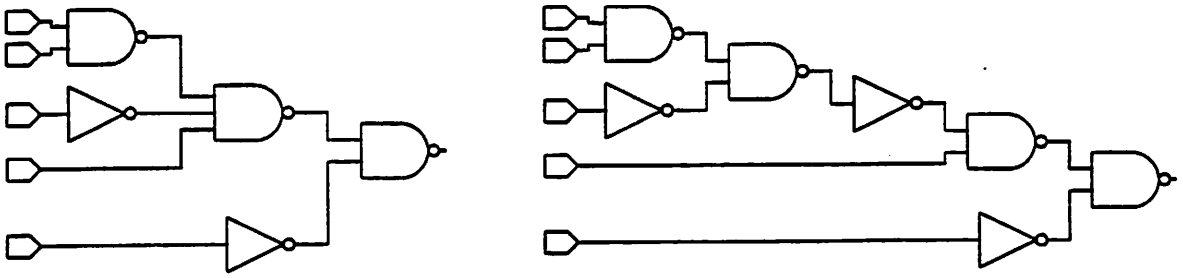

Figure 4.9: Coarse-resolution base function.

Proposition 4.4.3 *A base-function set which includes* NAND*-gates with more than two inputs restricts the set of solutions resulting in lower optimization quality. However, the number of patterns required for some gates is decreased.*

The covering paradigm implies that each node of the subject graph is covered by a pattern, but cannot be split and partially covered by two patterns. Therefore, the granularity of the base function set affects the optimization potential. This is best demonstrated with an example.

Example 4.4.1 *Assume the base function set includes a three-input* NAND*-gate in addition to the two-input* NAND*-gate and inverter. Consider the subject graph shown in Figure 4.9. A cover of this graph is unable to separate the three-input* NAND*-gate into two pieces, each of which combines with the surrounding circuitry. Instead, if only two-input* NAND*-gates and inverters are used as the base functions, the subject graph is represented as shown on the right-hand side of the figure. The second subject graph allows for the same covers as the first subject graph when the three-input* NAND*-gate pattern is included as a pattern; however, other covers which split the three-input* NAND*-gate are also legal covers.*

Thus, a fine resolution base-function set allows for more covers, and hence better quality solutions. However, this has a price – more patterns are required to represent the logic function for some gates. In *DAGON*, two-input, three-input, and four-input NAND-gates are used as the base-function set. With this approach, the logic function

$$f = \overline{abcd + efgh + ijkl + mnop}$$

requires only one pattern – a tree of five four-input NAND-gates. Representing all patterns for this same function using two-input NAND-gates and inverters requires eighteen patterns. However, given the possibility for improved optimization, the finer resolution base function appears to be the better approach.

4.4.2 Creating the Subject Graph

A logic network has many representations as graphs of components from the base-function set and each representation is a potential subject graph for DAG-covering. Each starting point leads to a graph cover of different cost. Even if the covering problem is solved exactly, every one of these starting points should be considered for an optimum solution.

Therefore, heuristics are used to find an optimal form for the subject graph. As mentioned in the introduction, these optimizations include algebraic decomposition and Boolean simplification techniques using technology-independent cost functions. The number of nodes in the subject graph is used as a technology-independent estimate of the area of the circuit. The total number of literals in sum-of-products form, as defined in Chapter 3, is effectively the same area estimator. The longest path from an input to an output in the subject graph is used to estimate the delay of the circuit.

The goal of technology-independent optimization is to find a representation for the circuit which provides a good starting point for DAG-covering. The optimized equations are then transformed into two-input NAND-gate and inverter form in a straightforward manner as follows. Assume an equation is represented in sum-of-products form using p product terms over n variables. A single p-input OR-gate is built, fed by p AND-gates, each with up to n inputs. Each of the AND-gates and OR-gates is constructed from a balanced binary tree of two-input NAND-gates and inverters. At each level of the tree, the number of leaves is divided in half, and a tree is built recursively for that many leaves. The recursion terminates with a tree of one leaf.

Note that balanced trees for the AND-gates and OR-gates are not necessarily the best form for the subject graph. Even when a balanced tree is used, the assignment of the inputs at the leaves of the balanced tree will affect the covering solution. As shown in the next example, the minimum cost cover for the subject graph depends on the form created for each tree in the circuit.

Example 4.4.2 *Consider building a ten-input* AND*-gate. A two-input* AND*-gate is the only*

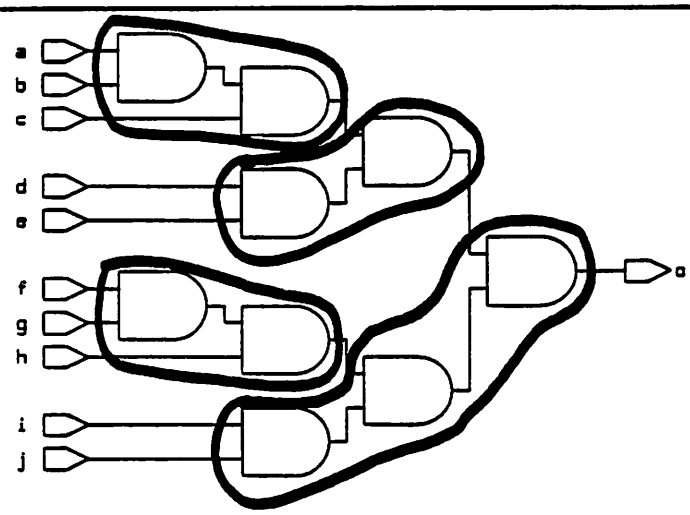

Figure 4.10: Optimal cover for a balanced-tree decomposition.

base function. The available gates in the library are a two-input AND*-gate with cost two, a three-input* AND*-gate with cost three, and a four-input* AND*-gate with cost four. The optimum cover for the balanced binary tree is thirteen, as shown in Figure 4.10. This tree places fives leaves under each branch of the root. However, if an unbalanced tree is used which places seven leaves under the first son, and three leaves under the second son, the optimum cover is twelve as shown in Figure 4.11.*

For this reason, a logic function is represented by all its pattern graphs, rather than just a single pattern graph. In this way, regardless of the form used to decompose the subject graph trees into two-input form, a pattern for the gate is more likely to be detected in the subject graph. This allows extra freedom to re-arrange the subject graph to improve the covering results. An example of a re-arrangement for timing optimization is presented next.

Given that the variables for each tree are arriving at different times, it is clear that a re-balancing of a tree can decrease the critical path delay in the circuit. Therefore, to optimize for delay, the trees are unbalanced to minimize the arrival time at the primary outputs. This is done by ensuring that the subject graph is minimal with respect to the transformation shown in Figure 4.12 (i.e., if the condition shown on the left-hand side of the

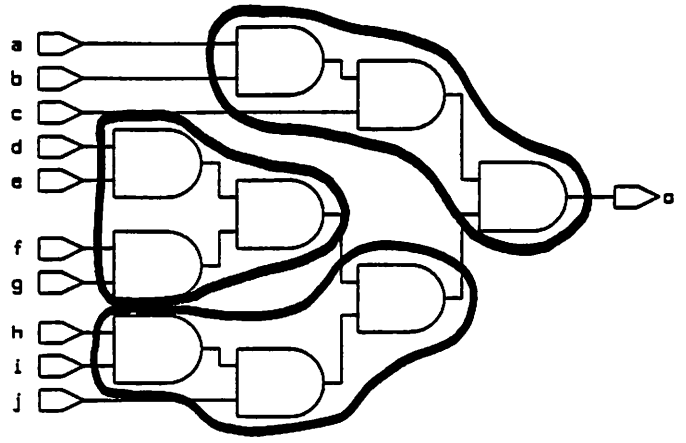

Figure 4.11: Optimal cover for an unbalanced-tree decomposition.

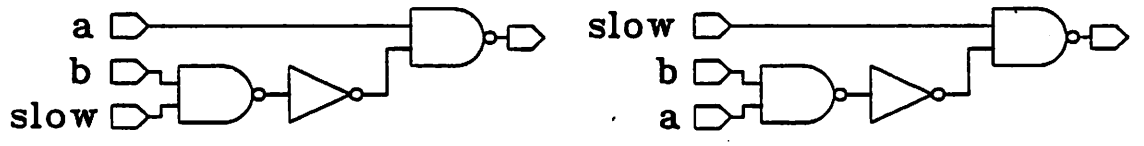

Figure 4.12: Tree-levelizing transformation.

figure is satisfied, re-arrange the connections as shown on the right-hand side to move the late-arriving signal forward). This transformation ensures that over each pair of NAND-gates separated by an inverter, the latest-arriving signal is at the deepest point in the circuit. Note that even if a circuit satisfies the condition shown in Figure 4.12 everywhere, it may still be possible to re-balance a tree to reduce the delay to an output; that is, arbitrary application of the rule shown in the diagram leads to a local minimum for balancing the trees within a circuit and not the global minimum [75].

4.4.3 A DAG-Covering Algorithm

In this section, an algorithm for solving the DAG-covering problem is presented. All matches of any pattern graph anywhere in the subject graph are generated and a covering problem is created where the matches are used to cover the nodes of the subject graph. Restrictions are placed on the matches selected for a cover to ensure that the cover will yield a valid circuit. The covering problem which is created is a special case of a 0/1 integer programming problem known as *binate-covering.*

Graph Terminology

Standard terminology is used for the subject and pattern directed graphs. A *directed graph* is a pair (V, E) consisting of a set of vertices (or nodes) V and a set of edges E; the edges E consist of ordered pairs (v_i, v_j) of vertices.

Because of the orientation of logic design, a source in the graph is called a *primary input*, and a sink in the graph is called a *primary output.* Likewise, the in-arcs of a node are called the inputs or fan-ins of the node, and the out-arcs are called the fan-outs of the node.

For a node v, the fan-in set of v is defined as $i(v) = \{w|(w, v) \in E\}$, and the fan-out set of v is defined as $o(v) = \{w|(v, w) \in E\}$.

The in-degree of a node v is $|i(v)|$ and the out-degree of v is $|o(v)|$.

For the algorithms presented here, it is not necessary to associate a logic function with each node in the subject or pattern graphs because each node is assumed to compute the NAND function. Also, it is not necessary to consider the ordering of the in-arcs of a node because the NAND function is symmetric. These conditions allow a simple graph data structure to be used for both the subject graph and the pattern graphs.

Further, for DAG-covering as considered here, all graphs have nodes with in-degree of either 1 or 2, and for the tree-covering approximation to be covered later, all nodes except the primary inputs of each tree will have an out-degree of 1.

Graph Matching Algorithm

Graph matching is the problem of finding all subgraph isomorphisms of the pattern graphs on the subject graph. Each isomorphism is called a match.

Definition 4.4.1 *A* match *of a pattern graph $G_p = (V_p, E_p)$ on a subject graph $G_s = (V_s, E_s)$ is a one-to-one mapping of the pattern graph nodes into the subject graph nodes $(I : V_p \rightarrow V_s)$ such that:*

1. $\forall e \in E_p, (I(e_1), I(e_2)) \in E_s$.
2. $\forall v \in V_p, |i(v)| \neq 0 \Rightarrow |i(v)| = |i(I(v))|$

The first property states that the edge relationships are preserved between the pattern graph and the subject graph and the second property forces the in-degree of each node (which is not a primary input of the pattern graph) to be identical.

Note that the definition of a match allows a subject graph node which is covered by a pattern graph to have fan-out to nodes which are not also in the pattern graph. However, for the tree-covering approximation, it is necessary to force a match to contain all fan-out nodes of each node in the subject graph covered by the pattern. This leads to the definition of an exact match.

Definition 4.4.2 *An* exact match *of a pattern graph $G_p = (V_p, E_p)$ on a subject graph $G_s = (V_s, E_s)$ is a one-to-one mapping of the pattern graph nodes into the subject graph nodes $(I : V_p \rightarrow V_s)$ such that:*

1. $\forall e \in E_p, (I(e_1), I(e_2)) \in E_s$.
2. $\forall v \in V_p, |i(v)| \neq 0 \Rightarrow |i(v)| = |i(I(v))|$
3. $\forall v \in V_p, |i(v)| \neq 0$ and $|o(v) \neq 0| \Rightarrow |o(v)| = |o(I(v))|$

The additional property for an exact match forces the out-degree of each node (which is not a primary input or primary output of the pattern graph) to be the same in the pattern graph and the subject graph.

This definition of a match allows an arbitrary permutation of the input and output arcs at each node of the subject graph. As mentioned previously, this is allowed because the base-functions are symmetric.

Consider the pattern graph for a three-input NAND-gate. Figure 4.13 shows a match and an exact match of this pattern graph on a subject graph. The left-hand side match is not exact because the inverter has fanout to a node which is not covered by the pattern. The right-hand side match is exact.

The algorithm **generate_all_matches**, shown in Figure 4.14, finds all matches for all patterns by assuming that a fixed node of each pattern graph is anchored on a given

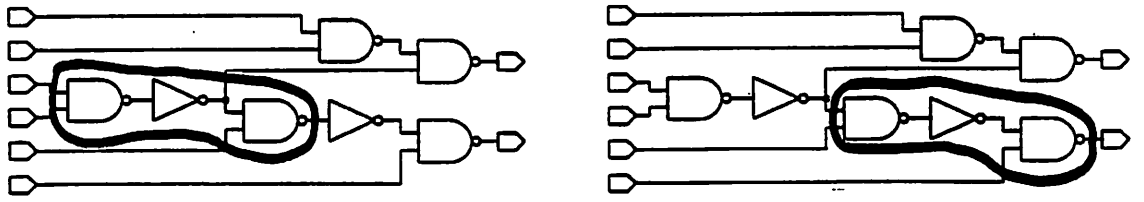

Figure 4.13: Example of a match and an exact match.

```
generate_all_matches(subject_graph, pattern_graphs) {
    foreach node in subject_graph {
        foreach pattern in pattern_graphs {
            graph_match(node, edge_list(pattern));
        }
    }
}
```

Figure 4.14: Algorithm **generate_all_matches**.

node of the subject graph. This is repeated for all nodes of the subject graph. For each successful match, the association between the pattern graph nodes and the subject graph nodes (i.e., the mapping $I()$) is recorded for later use.

Algorithm **graph_match**, shown in Figure 4.18, proceeds by making assertions that a node of the subject graph and a node of the pattern graph belong to a matching pattern. Each assertion *binds* the two nodes together. As the matching algorithm proceeds, these bindings are checked as new nodes are added to the match; if at any point the bindings are inconsistent, then the algorithm back-tracks, searching for a consistent set of bindings. Each time all nodes of the pattern graph are successfully bound to nodes of the subject graph, a match has been detected. Stored with each node in the pattern graph is the current binding and other information such as the in-degree and out-degree of the node. Initially

```
make_edge_list(node, previous_node, direction, edge_list) {
    append (node, previous_node, direction) to edge_list;
    if (node has been visited) {
        return;
    } else {
        mark node as visited;
        foreach fanin_node of node {
            if (edge(fanin_node, node) not already visited) {
                make_edge_list(fanin_node, node, INPUT, edge_list);
            }
        }
        foreach fanout_node of node {
            if (edge(fanout_node, node) not already visited) {
                make_edge_list(fanout_node, node, OUTPUT, edge_list);
            }
        }
    }
}
```

Figure 4.15: Algorithm make_edge_list.

each node in the pattern graph is marked as *unbound*.

The graph matching for a single pattern is done by generating an edge-based data structure called the *edge_list*. This data structure consists of one record for each edge; stored with the edge are the two nodes connected by the edge, and a direction indicating the direction of the edge. The edge-list is ordered such that as each edge (after the first) is visited, the first node connected by the edge has already been bound. The second node is generally not already bound, and all fan-ins or fan-outs of the subject node bound to the first node are tried while searching for successful matches. This allows the matching algorithm to look only at the adjacent nodes of an already bound node when searching for possible bindings for the unbound node. If the pattern graph is connected, it is always possible to form a list of the edges with this property. The algorithm make_edge_list, given in Figure 4.15, shows how this is done.

The first call to make_edge_list is done with node set to the *root* node of the pattern, previous_node set to NIL, direction set to INPUT, and the edge_list list set

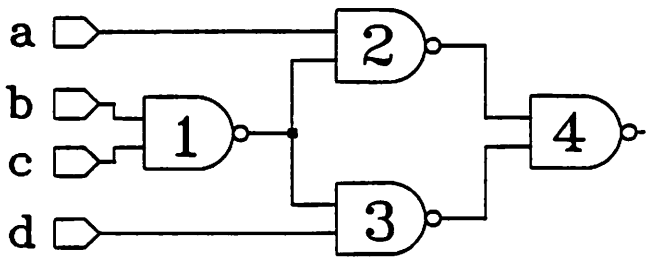

Figure 4.16: Sample pattern graph for edge_list.

to empty. The node chosen as the root node of the pattern is arbitrary and for convenience is taken to be the first primary output.

Example 4.4.3 *Consider the graph shown in Figure 4.16. The edge-list for this graph is shown in Figure 4.17.*

consistent checks that a binding of `subject_node` to `pattern_node` is consistent. The binding is consistent if the two nodes are already bound to each other. Otherwise, if both nodes are unbound, then the binding is consistent if the properties of Definition 4.4.1 are satisfied (for a match), or if the properties of Definition 4.4.2 are satisfied (for an exact match).

bind marks both the subject node and the pattern node as being bound to each other. Thus, if either node is revisited during the matching, **consistent** can guarantee that consistent bindings are made. The algorithm **graph_match** is then called recursively for each fan-in or fan-out of `subject_node`, depending on the stored direction of the edge.

The **graph_match** algorithm explicitly tries all possible orientations of the pattern graph in the subject graph; hence, only one pattern is needed for what might otherwise be a large collection of isomorphic patterns.

Because of the sparsity of the subject graph, the problem of generating all matches is actually quite easy. For a graph with 2,425 nodes, finding all matches for the 19 patterns from the IWLS-87 library, required 27 seconds on a 1-MIP computer. There were 5,686 successful matches.

node	previous	direction
4	nil	
2	4	input
A	2	input
1	2	input
B	1	input
C	1	input
3	1	output
D	3	input
3	4	output

Figure 4.17: Edge-list for the graph of Figure 4.16.

The Unate-Covering Problem

Exact solutions to two-level Boolean minimization rely on transforming the minimization problem into a covering problem, where the goal is to cover all minterms using a minimum number of prime implicants. This covering problem is called the *unate-covering problem.*

At the heart of the unate-covering problem is the covering expression. It is derived in the following manner. Let $p_1, \ldots, p_s$ be Boolean variables representing the presence of the corresponding prime implicant p_j in the solution. For each minterm, a single conjunctive clause is written listing the primes which cover that minterm. For example, assume that a minterm is covered by primes p_1, p_5, and p_{10}. This is represented by the clause $(p_1 + p_5 + p_{10})$. The product of these clauses provides the covering expression for the minimization problem. Any assignment of 0 and 1 to the variables of the covering expression which makes the expression true is a valid solution for the two-level minimization problem. Such an assignment is called a *satisfying assignment.* The satisfying assignment with the fewest number of variables assigned the value 1 (called the *minimum satisfying assignment*) is the solution with the fewest number of prime implicants. This can be generalized to give each prime implicant p_j a weight c_j, in which case the goal is the satisfying assignment with the least total weight (i.e., if p_j is assigned 1, it contributes c_j to the total weight).

The minimum satisfying assignment problem has many equivalent formulations.

```
graph_match(subject_node, pattern_node, edge_list) {
    if (! consistent(subject_node, pattern_node)) {
        return ;
    }
    if (subject_node not already bound to pattern_node) {
        bind(subject_node, pattern_node);
    }

    edge_list = edge_list->next_edge;
    if (edge_list is empty) {
        record successful match;
    } else {
        switch(edge_list->direction) {
        INPUT:
            foreach fanin_node of subject_node {
                graph_match(fanin_node, edge_list->node, edge_list);
            }
            break;
        OUTPUT:
            foreach fanout_node of subject_node {
                graph_match(fanin_node, edge_list->node, edge_list);
            }
            break;
        }
    }
    if (subject_node was not already bound to pattern_node) {
        unbind(subject_node, pattern_node);
    }
}
```

Figure 4.18: Algorithm graph_match.

For example, the minimum-cover problem [31][Page 222] and the minimum hitting set problem [31][Page 222] are different formulations of this problem. The covering expression formulation, however, provides an interesting insight. First, note that the covering expression is a *unate* logic function [19]. Each implicant of the covering expression logic function is a satisfying assignment. That is, given an implicant of the covering expression, derive the assignment of variables so that any variable not present in the implicant has value 0, and any variable present in the implicant has value 1. The interesting point is that the prime implicants of the covering expression correspond to minimal satisfying assignments (minimal in the sense that no asserted variable can be changed to a 0 with the expression remaining satisfied), and that the largest prime implicant corresponds to the minimum satisfying assignment. Because the covering expression is unate, the set of all prime implicants can be formed by multiplying-out the expression into sum-of-products form and performing single-term containment on the result. The largest prime implicant is then easily determined by inspection.

This formulation of the two-level minimization problem has proven successful. Once techniques were discovered which did not force the creation of the covering expression in terms of minterms, even large problems could be solved exactly [62]. Note, however, that a branch-and-bound solution, as described in Chapter 2 of this thesis, is much more efficient than straightforward expansion of the covering expression into prime implicants. The problem with generating the prime implicants of the covering expression is that it explicitly generates all minimal satisfying assignments, and not just one minimum solution to the covering problem.

This problem is called the unate-covering problem because the covering expression is unate in all of its variables.

Binate-Covering Problem

The DAG-covering problem can be formulated in a similar fashion. First, all matches are generated of the pattern graphs in the subject graph. Note that exact matches are not required; a match is allowed to have nodes internal to the match feed nodes not in the match. Then a covering expression is written to express the conditions which lead to a legal cover.

Using a Boolean variable m_i for each successful match of a pattern graph in the

subject graph, a clause is written for each node of the subject graph indicating which matches can cover this node. For example, if a subject node is covered by matches m_2, m_5 and m_{10}, then the clause would be $(m_2 + m_5 + m_{10})$. This is repeated for each subject node, and the product is taken over all subject nodes.

Any satisfying assignment to the expression formed thus far guarantees that all subject nodes are covered, but does not guarantee that another match actually creates as an output the input signal needed for a given match. This can be rectified by adding additional clauses.

Assume that match m_i with n inputs has subject nodes $s_{i_1}, \ldots, s_{i_n}$ as inputs. If match m_i is chosen, one of the matches which realizes s_{i_j} must also be chosen for each input j in order to create a valid circuit. Let S_{i_j} be the disjunctive expression in the variables m_i giving the possible matches which realize subject node s_{i_j} as an output node. Selecting match m_i implies satisfying each of the expressions S_{i_j} for $j = 1 \ldots n$. This can be written $m_i \Rightarrow (S_{i_1}, \ldots, S_{i_n})$ which by a simple Boolean transformation equals $\overline{m_i} + (S_{i_1} \cdots S_{i_n})$. This can be converted to product-of-sums form as $(\overline{m_i} + S_{i_1}) \cdots (\overline{m_i} + S_{i_n})$. Note that each S_i is a disjunctive expression over the variables m_i.

Each match thus generates a number of additional clauses forcing its input nodes to be created by some other match if the given match is selected. Also, one of the matches for each primary output of the circuit must be selected.

An assignment of values to the variables m_i which satisfies the above covering expression is a legal graph cover. For area optimization, each match m_i has a cost c_i which is the area of the gate the match represents; the goal is a satisfying assignment with the least total cost.

Note that this problem is more general than the unate-covering problem for two-level minimization because variables are present in the covering expression in both their true and complemented forms. Hence, the covering expression is a binate logic function, and the problem is referred to as the *binate-covering problem*. In fact, for DAG-covering, almost all variables are present in both their true and complemented forms.[1] For this reason, the problem is referred to as the binate-covering problem.

Example 4.4.4 *Consider the subject graph of Figure 4.19. Fourteen matches of a pattern graph for gates in the* IWLS-87 *library are given in table 4.1. Shown in the table are the*

[1] If a match is fed only by primary inputs of the subject graph, then the corresponding match variable will not appear in its complemented form.

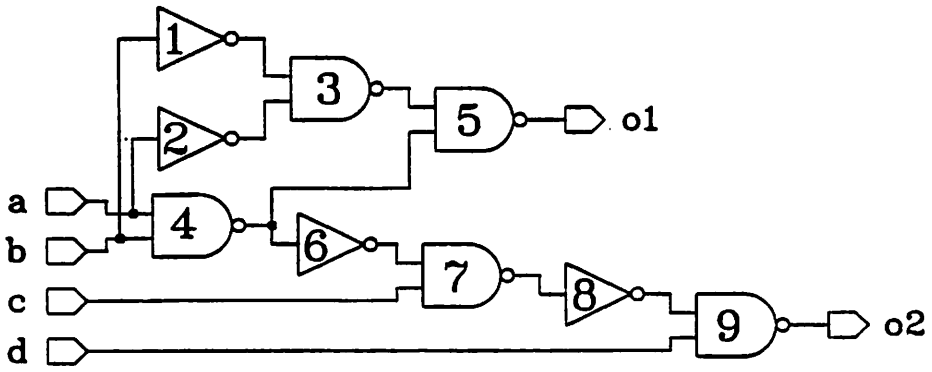

Figure 4.19: Example graph for DAG-covering.

inputs required by the match, and the output which is produced by the match. Also shown are the nodes covered by the match. Note that the first nine matches are the trivial matches of covering each node by either an inverter or a two-input NAND*-gate.*

The covering expression is easily created from this table. The first part consists of the constraints that each node g_i be covered by some match.

$$(m_1 + m_{12} + m_{14})(m_2 + m_{12} + m_{14})(m_3 + m_{12} + m_{14})(m_4 + m_{11} + m_{12} + m_{13})$$
$$(m_5 + m_{12} + m_{14})(m_6 + m_{11} + m_{13})(m_7 + m_{10} + m_{11} + m_{13})(m_8 + m_{10} + m_{13})$$
$$(m_9 + m_{10} + m_{13}).$$

Next, to ensure that a cover leads to a valid circuit, extra clauses are generated. For example, selecting m_3 requires that a match be chosen which produces g_1 as an output, and a match be chosen which produces g_2 as an output. The only match which produces g_1 is m_1, and the only match which produces g_2 is m_2. Also, the primary output nodes g_5 and g_9 must be realized as an output of some match. The matches which realize g_5 as an output are m_5, m_{12}, and m_{14}; the matches which realize g_9 as an output are m_9, m_{10} and m_{13}.

Note that a match which requires a primary input as an input is satisfied trivially. Matches $m_1, m_2, m_4, m_{11}, m_{12}$ and m_{13} are driven only by primary inputs, and hence do not require additional clauses.

$$(\overline{m}_3 + m_1)(\overline{m}_3 + m_2)(\overline{m}_5 + m_3)(\overline{m}_5 + m_4)(\overline{m}_6 + m_4)(\overline{m}_7 + m_6)(\overline{m}_8 + m_7)$$

	gate	cost	inputs	produces	covers
m_1	inv	1	a	g_1	g_1
m_2	inv	1	b	g_2	g_2
m_3	nand2	2	g_1, g_2	g_3	g_3
m_4	nand2	2	a, b	g_4	g_4
m_5	nand2	2	g_3, g_4	g_5	g_5
m_6	inv	1	g_4	g_6	g_6
m_7	nand2	2	g_6, c	g_7	g_7
m_8	inv	1	g_7	g_8	g_8
m_9	nand2	2	g_8, d	g_9	g_9
m_{10}	nand3	3	g_6, c, d	g_9	g_7, g_8, g_9
m_{11}	nand3	3	a, b, c	g_7	g_4, g_6, g_7
m_{12}	xor2	5	a, b	g_5	g_1, g_2, g_3, g_4, g_5
m_{13}	nand4	4	a, b, c, d	g_9	g_4, g_6, g_7, g_8, g_9
m_{14}	oai21	3	a, b, g_4	g_5	g_1, g_2, g_3, g_5

Table 4.1: A partial list of matches.

$$(\overline{m}_9 + m_8)(\overline{m}_{10} + m_6)(\overline{m}_{14} + m_4)(m_5 + m_{12} + m_{14})(m_9 + m_{10} + m_{13})$$

The covering expression has 58 prime implicants, and the least cost prime implicant is $\overline{m}_3 + \overline{m}_5 + \overline{m}_6 + \overline{m}_7 + \overline{m}_8 + \overline{m}_9 + \overline{m}_{10} + m_{12} + m_{13} + \overline{m}_{14}$ *which uses two gates for a cost of nine gate units. This corresponds to a cover which selects matches* m_{12} *(xor2), and* m_{13} *(nand4). Note that the node* g_4 *is covered by both matches. There are two other solutions of cost nine, each with more gates.* $\overline{m}_3 + m_4 + \overline{m}_5 + \overline{m}_7 + \overline{m}_8 + \overline{m}_9 + \overline{m}_{10} + m_{13} + m_{14}$ *corresponds to the selection of matches* m_4 *(nand2),* m_{13} *(nand4), and* m_{14} *(oai21), and* $\overline{m}_3 + m_4 + \overline{m}_5 + m_6 + \overline{m}_8 + \overline{m}_9 + m_{10} + m_{14}$ *corresponds to the selection of matches* m_4 *(nand2),* m_6 *(inv),* m_{10} *(nand3), and* m_{14} *(oai21).*

The binate-covering problem is almost as old as the unate-covering problem. The binate-covering problem has appeared before as a formulation for the state minimization of incompletely-specified finite-state machines [34], as a problem in the design of optimal three-level NAND-gate networks [32], and in the optimal phase-assignment problem [62].

The problem has also been called the *covering-with-closures* problem. Interestingly, in [34], if one transposes the matrix of Figure 4, replaces circles with 0, crosses with 1, and

empty squares with 2, one gets exactly the cubical-format of the binate-covering expression used in that example. Because of the reliance on Boolean expressions and cube-notation for logic functions which has become common-place in logic synthesis, I prefer to call this problem the binate-covering problem, rather than the covering-with-closures problem.

Complexity of DAG-Covering

In [22], it is proven that the graph-covering problem is NP-complete. The proof is based on a reduction from satisfiability. For technology mapping, the graphs are restricted to an in-degree of two or less but have unbounded out-degree. However, the DAG-covering problem remains NP-complete even if the in-degree and out-degree of each node is two or less [48].

Experimentally, exact solutions for the unate-covering problem in two-level minimization are feasible even for problems with thousands of variables. This is primarily because of the use of a branch-and-bound solution technique and the sparsity of the covering problems.

The number of variables for the binate-covering problem in DAG-covering equals the number of successful matches of a pattern graph in the subject graph and the number of clauses is the number of matches plus the number of inputs on all of the matches. Note that the covering expression is sparse – typically, a pattern graph covers only a small number of subject nodes and a subject node can be covered by only a few patterns. For these reasons, there was hope that exact techniques for binate-covering based on a branch-and-bound algorithm would be feasible even for large problems.

The unate-covering algorithm presented in Chapter 2 was modified to perform binate-covering and some experiments were performed on applying this technique to exact DAG-covering. The negative result was that an exact solution for the DAG-covering problem was infeasible except for problems with a small number of variables. Given that subject graphs for VLSI circuits will have tens of thousands of nodes, exact solutions for DAG-covering using a branch-and-bound technique appear infeasible.

Some heuristics algorithms for binate-covering have appeared in the literature [34, 32], but no good approximate techniques are known for problems of this size and complexity.

4.5 Tree-Covering Approximation

One technique for efficiently solving the DAG-covering problem is to explore approximate solutions to the binate-covering problem. An alternate approach is to partition the subject graph into a forest of trees and solve the covering problem on each of the trees. This is called the *tree-covering approximation* to DAG-covering. The motivation for the tree-covering approximation is the existence of an efficient algorithm for the optimal tree-covering problem. In this section, the optimal tree-covering algorithm is described. Special consideration is given to solving the tree-covering problem for the cost functions of minimum delay and minimum area under a delay constraint. Given that a subject graph is first partitioned into a set of trees, several additional heuristics are described to improve the results for technology mapping.

4.5.1 Dynamic Programming Algorithms

Dynamic programming is a general technique for algorithm design which can be applied when the solution to a problem can be viewed as the result of a sequence of decisions. Dynamic programming algorithms rely on the *principle of optimality*:

> *principle of optimality*:[43] An optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

The next section contains an example of the application of the principle of optimality to derive an efficient dynamic programming algorithm for the optimal tree-covering problem.

4.5.2 Optimal Tree-Covering

The tree-covering problem is a simplified form of the DAG-covering problem where the subject graph and the pattern graphs are restricted to *trees*. A tree is a directed acyclic graph where every node (including primary inputs) has a fanout of 1. The tree necessarily has a single primary output called the *root* and the primary inputs of the tree are called the *leaves* of the tree.

Consider the problem of finding a minimum area cover for a subject tree T. A scalar cost is assigned to each tree pattern and the cost for a cover is the sum of the costs

```
optimal_area_cover(node) {

    /* Find optimal cover for all nodes below "node" */
    foreach input of node {
        optimal_area_cover(input);
    }

    /* Using these optimal covers, find the best cover at "node" */
    node->area = INFINITY;
    node->match = 0;
    foreach match at node {
        area = match->area;
        foreach pin of match {
            area = area + pin->area;
        }
        if (area < node->area) {
            node->area = area;
            node->match = match;
        }
    }
}
```

Figure 4.20: Algorithm optimal_area_cover.

for each pattern in the cover. The key observation is stated in the following proposition.

Proposition 4.5.1 *The minimum-area cover for a tree T can be derived from the minimum-area covers for every node below the root of T.*

This is the *principle of optimality* for tree-covering and is used as follows to find an optimum cover for T. For every match at the root of the tree, the cost of an optimal cover containing that match equals the sum of the cost of the corresponding gate and the sum of the costs of the optimal covers for the nodes which are inputs to the match. Note that the optimum covers for each input to the match at the root can be computed once and stored; it is not necessary to re-compute the optimum cover for each input of each match.

The algorithm optimal_area_cover, given in Figure 4.20, shows how this is done. Stored with each node are two fields: match providing the best match seen for that node

and area which is the cost of the best match. Initially, **optimal_area_cover** is called on the root of the tree to be covered. The first step is to recursively call **optimal_area_cover** for each input of the current node. This provides the assertion that the optimal cost cover is known for each subtree below the current node and has been recorded at the node. Then, the graph matching algorithm presented earlier is used to find all exact matches of the pattern trees at the current node. Each match is assigned a cost which is the sum of the costs for each input required by the match (i.e., the cost for the optimal cover for the node in the subtree below the target node, which has already been computed) and the cost of the current match. Every match at the target node is examined and the least cost match is the match in the best cover for the current node. The best match and the cost for this match is recorded at the node. The optimum cover is easily derived from the optimum match which is stored at each node by starting from the optimum match at the root of the tree.

Note that each node in the tree is visited only once. Therefore, the complexity of this algorithm equals the number of nodes in the subject tree times the maximum number of matches at any node in the subject tree. The maximum number of matches is a function of the library and is therefore a constant independent of the subject tree size. As a result, **optimal_area_cover** has linear complexity in the size of the subject tree. The memory requirements for this algorithm are two words for each node in the subject graph to store the optimum matching pattern and the cost of the best match. Therefore, the memory requirements are also linear in the size of the subject tree.

4.5.3 Delay Optimization

The dynamic programming algorithm presented in the last section is optimal for minimizing the cost function of minimum area. In this section, extensions for the cost functions of minimum-delay and minimum-area under a fixed delay constraint are considered.

Minimum Delay Cost Function

The first case considered is finding the minimum-delay tree cover. The delay model described in Section 4.3.2 is assumed. The cost of a tree-cover is defined as the signal arrival time at the root of the tree. The signal arrival times for each primary input are part of the problem definition.

For the first algorithm, the additional simplifying assumption is made that each

gate input has a fixed load (i.e., $\gamma_i = \gamma^0$ where γ^0 is the same for all gates). With this assumption, the delay model assigns a fixed value for the delay from each input to each output, regardless of the gate which is driven. This restriction will be removed later.

Proposition 4.5.2 *The minimum-delay cover of a tree T can be derived from the minimum-delay covers for every node below the root of T.*

Therefore, the principle of optimality continues to hold. **optimal_area_cover** can be applied to delay optimization with a simple change to the cost function evaluation. First, the optimal covers (i.e., the cover which minimizes the delay to the node) for all nodes below T in the tree are computed. At the root of T, the arrival time for the optimum cover for each input of a match is combined with the arcs for the timing model of the pattern to find the arrival time at the output of the match. Instead of summing the area for each input match, the maximum over the input arrival times plus the input arc delay is used. This provides the arrival time for the match. The minimum arrival-time match is chosen as the match for the node. By the principle of optimality, this provides the minimum delay cover for the subject tree. This algorithm is called **optimal_delay_onesize_cover**. Because of the similarity of this algorithm to **optimal_area_cover**, its pseudo-code description is omitted.

Non-Constant Input Load

The assumption that the timing arcs for each match are independent of the gate driven by the match was essential for the direct application of the dynamic programming algorithm to the problem of delay optimization. However, this assumption is not realistic.

In CMOS gate-array and standard-cell libraries, many different gates for the same logic function are typically available. These gates differ in their area and timing cost parameters. A small drive version of a gate has a low value of input load (γ), but also a low drive capability (a high value of β). A higher drive version of a gate costs more in area, has a higher input load, and has a higher constant delay (α), but also has a higher drive capability (a lower value of β). Choosing from among the available gates to optimize timing performance of the design is an important aspect of technology mapping.

In the timing model presented earlier, the input load for a pin (γ) models the capacitance of the electrical gates of the NMOS and PMOS transistors in the library cell. In gate-array (and compacted array) design styles, the transistor sizes (and hence the input

loads) tend to take on a small number of discrete values. This is because larger transistors are created from parallel connections of a fixed-size transistor. In standard-cell design the values of input load vary much more because the transistors of each gate are sized optimally for that gate. Thus, even among gates of the same size, the values of the input load vary.

The dynamic programming algorithm can be extended to provide optimal solutions for variable pin load. The key observation is that the number of distinct input load values in a technology library is bounded by the number of pins on all of the gates and hence is finite. As mentioned above, there are few distinct load values in a CMOS gate-array library (typically less than ten). Even for CMOS standard-cell libraries, the number of distinct load values is typically small (typically less than one hundred).

The first step in the algorithm **optimal_delay_cover** is a pre-processing step over the technology library. A fixed number n load bins are used to discretize the load values for all pins in the library. The values for the load bins are selected and the remaining load values are mapped to their closest value in the chosen set. Functions **load_to_integer(load)** and **integer_to_load(integer)** are defined to map between an actual value of input load and an integer $1 \ldots n$. This is a standard binning procedure to discretize the load values.

optimal_delay_cover is presented in Figure 4.21. The cost and best match at the node are now represented by arrays `arrival[i]` and `match[i]`, indexed by the load value for which this match is optimal. The delay model values for each match include the constant-delay value `alpha[pin]`, the load dependent portion `beta[pin]`, and the input pin load `gamma[pin]`. In this description, these are represented as attributes on the input pin of the match.

At each node, an array of optimal solutions is maintained, one for each value of input load. As before, the algorithm is applied recursively to find the minimum arrival-time cover for each node below the root. But, for this algorithm, the minimum-cost cover is also found for $\gamma^0 = \textbf{integer\_to\_load}(i)$ for each $i = 1, \ldots, n$. As a match is found in the tree, the arrival time for the match is computed once for each value of pin load. For each input to the match, the optimum match for driving the pin load of pin i of the match is assumed, and the arrival time for that match is used.

Proposition 4.5.3 *If n is chosen to include all distinct load values in the library, then* **optimal_delay_cover** *provides the minimum arrival-time cover for the tree.*

The complexity of this algorithm is still linear in the size of the subject tree.

```
optimal_delay_cover(node) {

    /* Find optimal cover for all nodes below "node" */
    foreach input of node {
        optimal_delay_cover(input);
    }

    /* initialize the best match for each load value */
    for i from 1 to n {
        node->arrival[i] = INFINITY;
        node->match[i] = 0;
    }

    /* find the best match at "node" for each load value */
    foreach match at node {
        for i from 1 to n {
            load = integer_to_load(i);

            /* evaluate cost of match driving load "load" */
            max_arrival = - INFINITY;
            foreach pin of match {
                pin_load = load_to_integer(match->gamma[pin]);
                this_arc = match->alpha[pin] + match->beta[pin]*load;
                arrival = pin->arrival[pin_load] + this_arc;
                if (arrival > max_arrival) {
                    max_arrival = arrival;
                }
            }

            /* See if this is the best for these conditions */
            if (max_arrival < node->arrival[i]) {
                node->arrival[i] = max_arrival;
                node->match[i] = match;
            }
        }
    }
}
```

Figure 4.21: Algorithm optimal_delay_cover.

Compared to **optimal_area_cover**, this optimal algorithm is slower by a constant factor dictated by the number of distinct load values in the library. The storage requirements of this algorithm have also increased by the same factor; it is now necessary to store n best matches and the cost for the match at each node in the tree. Therefore, to improve the run-time and/or reduce the memory requirement, fewer values may be used to provide an approximate covering technique.

With the assumption that faster gates tend to cost more in area, **optimal_delay_cover** is wasteful in terms of the area required to achieve a given value of delay. The condition that the minimum-delay cover be used for each node below T when deriving the minimum-delay cover at the root of T is sufficient to guarantee a minimum-delay cover, but is not necessary. Typically only a single input to the match at the root is *critical*; the other inputs can be covered with a lower-cost cover, as long as the delay does not exceed that of the critical input. Extending this observation into an algorithm for minimum-area optimization under a delay constraint is covered in the next section.

Minimum Area under a Delay Constraint

The final problem considered is minimizing the area of the tree-cover given a constraint on the arrival time of the signal at the output. This is handled in a fashion similar to the discretization of the pin load. Bins are used to discretize the arrival times at each node in the tree, and the minimum-area solution for each bin is recorded. However, it is not reasonable to represent the signal arrival times as a mapping onto the integers. For example, a 0.01 nanosecond (ns) timing resolution on a circuit with a critical-path delay of 100 ns requires 10,000 bins to accurately discretize the arrival time at each node. Instead, an approximate solution is suggested here.

A fixed number of bins are chosen, and upper and lower bounds are placed on the arrival time at each node. These bounds are used to create a fixed number of uniformly spaced bins for the arrival time at the node. The algorithm is then applied iteratively, and the arrival time bounds at each node are adjusted based on the result from the previous pass. At each step in the algorithm the minimum area cover which meets the delay constraint is chosen as the solution; if no cover meets the delay constraint, then the minimum area cover producing the fastest circuit is returned. The algorithm terminates when the area for the circuit does not decrease.

```
delay_technology_map(tree) {

    /* annotate each node with max_arrival */
    optimal_area_cover(tree);

    /* annotate each node with min_arrival */
    area = optimal_delay_cover(tree);

    forever {
        this_area = optimal_area_under_delay_constraint(tree);
        if (this_area >= area) {
            break;
        }
        area = this_area;

        /* cut range at each node in half */
        foreach node in tree {
            range = node->max_arrival - node->min_arrival;
            node->min_arrival = node->arrival - range / 4;
            node->max_arrival = node->arrival + range / 4;
        }
    }
}
```

Figure 4.22: Algorithm delay_technology_map.

Pseudo-code for the iterative algorithm is shown in Figure 4.22. Each node in the tree is annotated with **min_arrival**, which is the arrival time for the first arrival time bin, and **max_arrival**, which is the arrival time for the last arrival time bin. First the tree is mapped for the minimum area solution using **optimal_area_cover**. From this cover, each node in the tree is tagged with the arrival time of the match which contains that node. This provides an upper bound on the arrival time for this node. The tree is then mapped for its minimum delay solution using **optimal_delay_cover**. From this cover, each node in the tree is tagged with the arrival time of the match which contains that node. This provides a lower bound on the arrival time for this node.

In the inner loop, **optimal_area_under_delay_constraint** is used to find the minimum-area solution which meets the delay constraint. This is done using the current value of the arrival time bounds at each node, and a fixed number of arrival time bins. Then the range of the arrival time bounds for each node is cut in half, centered around the optimal arrival time found on the previous pass. In this way, each bin provides a more accurate approximation for the arrival times which are optimal for each portion of the tree on the next pass. This process is iterated until the solution no longer improves.

The algorithm **optimal_area_under_delay_constraint** is shown in pseudo-code in Figure 4.23. At each node, the mapping between arrival times and integers is performed with the functions **arrival_to_integer(node, arrival)** and **integer_to_arrival(node, integer)**. m is the number of arrival times for each node. A two-dimensional array of solutions is recorded for each node. The first dimension provides a discretization of the load values, and the second dimension provides a discretization of the arrival times. For each value of (load, arrival), each match is evaluated to determine if it is the minimum area match for this value of load and arrival time. This is done by looking backwards at the solutions for each input of the match.

This is best explained with a simple example. Assume the algorithm is applied at a node and the optimal solution for an arrival time of 20 ns with a load of 1 is desired. Further, assume the delay from input pin 0 to the output is 2 ns (under the assumption of a load of 1), and input pin 0 has a load of 2 units. Then the minimum area solution for input 0 of the match which meets an 18 ns constraint when driving a load of 2 units is selected as the optimal solution for this input. The load value of 2 and the arrival time value of 18 ns are discretized, and used to select the optimal cover for the conditions closest to this set of conditions. This process is continued for all inputs, and the minimum area solution

is summed for each solution to provide the minimum area solution which meets the 20 ns delay constraint, given that the node is driving a load of 1. This is repeated for all of the discretized values of pin-load and arrival time to record a set of optimal solutions for this node in the tree.

After application of this algorithm, the optimal solution is determined from the solutions stored at the root of the tree. The delay constraint and load at the root are used to index the minimum-area solution.

From a complexity standpoint, **optimal_area_under_delay_constraint** is also linear in the size of the subject graph. However, now the constant factor in the algorithm depends on the product of the number of arrival time bins and the number of load bins. This is true even if every input load value and every resolvable arrival time value is represented with a distinct bin. However, in this case, the constant factor becomes very large. This motivates the use of an iterative algorithm to allow fewer bins to be used per pass. The accuracy of the approximation is increased on each iteration as the arrival time bins are narrowed around the solution from the previous iteration.

4.5.4 Partitioning the Subject Graph

To apply the tree-covering approximation, the subject graph must first be converted into a forest of trees. One approach is to break the graph at each multiple-fanout point. Each node with fanout greater than one becomes a root of a tree and each fanout of this node becomes a leaf of a tree. With this technique no nodes in the subject graph are duplicated. This is called the *trivial partition.*

Note that it is not necessary in the actual implementation to partition the subject graph into a forest of trees. If only tree patterns are used for the library gates and exact matching is performed, then a match cannot match across a multiple fanout point. Hence, the algorithms from the previous section can be applied directly to the subject graph. The only requirements are that the cost for each multiple-fanout node is initialized correctly and that each node of the subject graph is visited only once.

Other heuristics can also be used to partition the subject graph; however, all other partitions increase the number of nodes in the forest of trees. This can be useful, however, to improve the quality of the final cover. In the limit, the optimum DAG-cover can be found by applying tree-covering after choosing the correct nodes to duplicate.

```
optimal_area_under_delay_constraint(node) {

    foreach input of node {
        optimal_area_under_delay_constraint(input);
    }

    foreach i from 1 to n {
        foreach j from 1 to m {
            node->area[i][j] = INFINITY;
            node->match[i][j] = 0;
        }
    }

    foreach match at node {
        foreach i from 1 to n {
            load = integer_to_load(i);

            foreach j from 1 to m {
                arrival = integer_to_arrival(node, j);

                area = match->area;
                foreach pin of match {
                    this_arc = match->alpha[pin] + match->beta[pin]*load;
                    pin_arrival = arrival_to_integer(node, arrival-this_arc);
                    pin_load = load_to_integer(match->gamma[pin]);
                    area += pin->area[pin_load][pin_arrival];
                }

                if (area < node->area[i][j]) {
                    node->area[i][j] = area;
                    node->match[i][j] = match;
                }
            }
        }
    }
}
```

Figure 4.23: Algorithm optimal_area_under_delay_constraint.

4.5.5 Phase-Assignment Heuristics

Inverter-pair heuristic

A simple way to improve the quality of circuits produced by the tree-covering algorithm is the *inverter-pair heuristic.* Redundant inverters are added to each tree to improve the number of patterns which can match at each node. This leads to an examination of more possible covers for each tree, leading directly to an improvement in the optimization quality.

The technique works as follows. Each edge in the subject tree and each edge in a pattern which connects two NAND-gates is replaced with a pair of inverters. An extra pattern consisting of a pair of inverters is added to the matching patterns. This extra pattern is given zero area cost and zero delay cost. The tree-covering algorithm is then applied unmodified.

Because the tree-covering algorithm is optimal, adding the extra inverters cannot lead to a cover with a greater cost. Each pair of inverters can be covered by the inverter-pair pattern, which leads to the solution which existed before the inverters were added. However, the advantage is that the tree-covering algorithm is able to make the optimal choice between covering the extra inverters with the inverter-pair pattern at no cost, or splitting the inverters between two patterns if this leads to a cover with less cost. The only disadvantage is that the number of nodes in the subject tree and the pattern trees has increased. The increase in the number of nodes is bounded by a factor of three (two extra inverter nodes for each node in the subject tree); however, the actual increase is typically less because redundant inverters are added only at the output of a NAND-gate, and not at the output of each inverter in the subject tree.

Example 4.5.1 *Consider the subject graph from Figure 4.8. This is the optimal cover for this subject graph which used six gates with a total area cost of fifteen. Figure 4.24 shows the same subject graph with the redundant inverter-pairs added, and the optimal cover for this graph is shown. This cover uses five gates with an area cost of fourteen. Note that the gate* oai22, *which is now in the optimal cover, did not match in the subject graph before the extra inverters were added.*

Adding inverters within a tree is straightforward. If a NAND gate has a fanout of one and feeds another NAND gate, then a pair of inverters is added after the NAND gate.

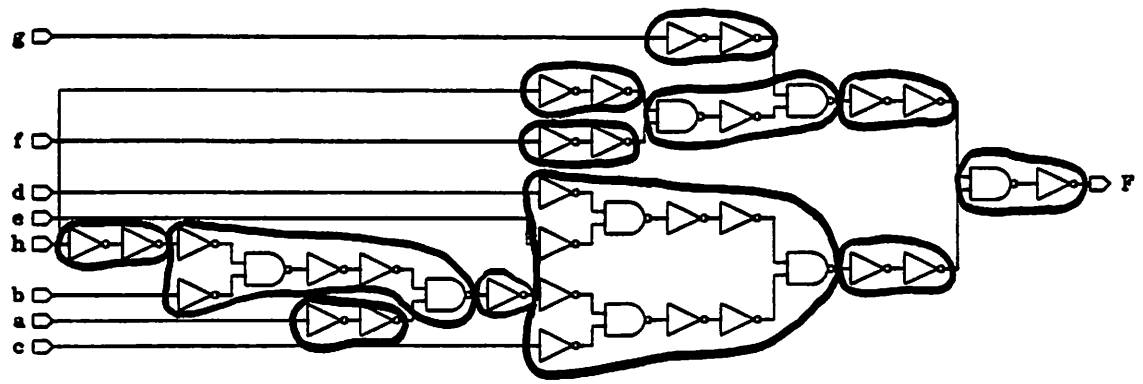

Figure 4.24: Alternate covering using the inverter-pair heuristic.

However, a slightly more complicated approach is taken at the tree boundaries. At a tree boundary, extra inverters are added so that each tree is connected to the branch node through either one or two inverters. Figure 4.25 shows in general how inverters are added at a multiple-branch node.

The motivation for adding the extra inverters in this fashion is again to increase the number of possible matches. Every tree should have one or two inverters at each leaf to allow more patterns to match near the leaves of the tree. The root of each tree could be constructed with no inverter at the root, but leaving one inverter in the previous tree increases the number of matches at the root of the tree.

Cross-Tree Phase Assignment Heuristics

A remaining problem in the application of the tree-covering approximation is solving the phase assignment problem across the tree boundaries. One approach is to defer the problem until after technology mapping. A global phase assignment algorithm, which is optimized for phase assignment on a graph [20], can be used to improve the quality of the mapping by reducing the number of inverters in the circuit. However, another approach is to capture the effect of cross-tree phase assignment during the application of the tree-covering algorithm for each tree. This leads to the *inverter branch-point heuristic.* Note that if the graph-covering problem were solved exactly, then phase assignment would be

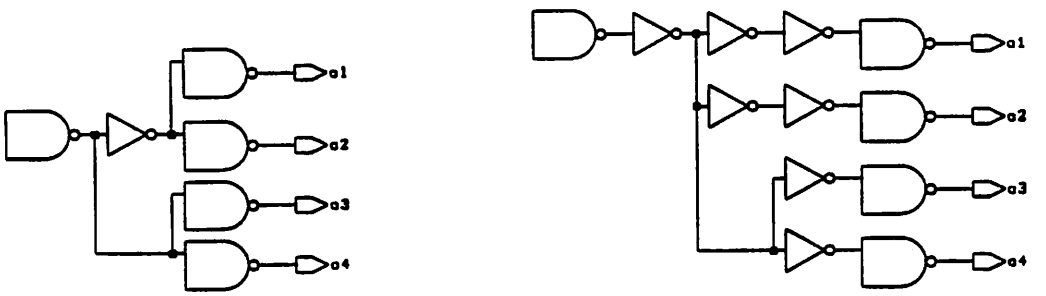

Figure 4.25: Adding inverter-pairs at a branch-point.

solved globally for the entire circuit at the same time. The inverter branch-point heuristic is an attempt to capture, within the tree-covering algorithm, the information known about the phases present between the trees.

As mentioned in the previous section, each tree has either one or two inverters at each leaf. Each tree has exactly one inverter at the root. As a result, each tree has been implicitly assigned a preferred phase for each tree leaf, and a preferred phase for the tree root. With the tree-covering algorithm, each tree is mapped optimally, including phase-assignment internal to the tree. However, this assumes that the choice of input phases for each tree is fixed. Because of the interaction between trees which share a common branch point, it may be possible to make a different choice for the input phases for the tree, leading to a better cover for a tree. As a simple example, consider a branch point which feeds ten other trees. If a single inverter is present at the root of the tree (i.e., the tree produces both the positive phase and negative phase of the signal), then all of the trees fed by this branch point are free to choose (at no cost) either the positive phase of the previous tree (i.e., the output of the inverter), or the negative phase of the previous tree (i.e., the signal from the input of the inverter). This degree of freedom is captured as follows.

At each branch point, a flag called *inverter_paid_for* is used to record whether the tree contains an inverter at its output. When the tree is mapped, if the optimal match at the root of the tree is an inverter, this flag is set. When a tree fed by the branch point is mapped, the cost for an inverter at the leaf of the tree is modified based on the the *inverter_paid_for* flag. If this bit is set, then the cost for an inverter match at the leaf of the

tree is taken as zero, and not the normal cost of the inverter. This allows the tree to select the phase it wishes to take from the previous tree.

After a tree is mapped, a recursive walk is performed over the tree to reach each leaf. At the leaf, if the branch point has not already paid for an inverter, and if the optimal cover for this tree contains an inverter at the leaf, then the *inverter_paid_for* bit is set at the branch point. Thus, any trees mapped after this tree will re-use the inverter in the other tree and will see a free choice of phases for this input.

Hence the *inverter_paid_for* bit records the current status at any time during the mapping as to whether a single inverter is present at the branch point. Note that this heuristic is order-dependent – changing the order of covering for the trees leads to different solutions.

4.5.6 Extension to Non-Tree Patterns

Some gates in a technology library cannot be represented in tree form. Common examples are a two-input exclusive-or gate, a two-to-one multiplexor, and a three-input majority gate (logic function $f = ab + ac + bc$). However, a simple extension allows these patterns to be included.

Definition 4.5.1 *A leaf-*DAG *is a* DAG *where the only nodes with fanout greater than one are the primary inputs.*

Patterns which are trees, and patterns which are leaf-DAGs can be used directly by the tree-covering algorithm. This includes the exclusive-or patterns shown in Figure 4.6. Note, however, that because of the multiple-fanout of one of these matches, the exclusive-or gate must match at the leaves of the tree.

4.5.7 Complexity of Tree-Covering

When applying tree-covering for minimum area optimization or minimum delay optimization for a fixed pin load, the complexity is proportional to the number of matches found on the subject tree. A simple cost function evaluation is done for each match. Therefore, the complexity of the tree-covering algorithm is merely the cost of generating all matches.

One clarification is necessary as to precisely what is meant by the set of unique tree-patterns. The pattern graphs considered up to this point consist of nonisomorphic, unlabeled trees. These are called the *nonexpanded* tree patterns. **graph_match** tries all permutations of the children of each node while searching for all matches. As a result, a three-NAND gate is represented by only a single tree pattern.

However, if the sons of the NAND-gate at the root of the three-input NAND-gate pattern are not considered interchangeable by the matching algorithm, then two patterns would be required to represent this tree. One pattern would place the inverter under the first child, and the second pattern would place the inverter under the second child. When the children of a node are given a fixed ordering, the matching algorithm becomes simpler, but many more patterns are needed to represent each gate. The set of tree patterns which results in this case is called the set of *expanded* patterns. Although an implementation deals with the nonexpanded patterns, it is necessary to count the number of expanded tree patterns when analyzing the complexity of the tree-matching algorithm.

The complexity of **graph_match** is $O(sp)$ where s is the number of nodes in the subject tree, and p is the total number of nodes in the set of *expanded* tree patterns. The reason the *expanded* tree patterns are used is that each node in the expanded representation of a pattern is visited once when **graph_match** is applied to each node in the subject tree. To see this, consider matching a pattern of only degree-two nodes on a full binary tree. A distinct match results for this pattern for every permutation of the children at each node. Therefore, the number of possible matches is the number of expanded trees for the given pattern.

Therefore, for minimum-area optimization and minimum-delay optimization for a fixed pin load, the run-time of the tree-covering algorithm is $O(sp)$. Because p is a constant which depends only on the library, the tree-covering approximation for technology mapping algorithm runs in linear-time in the size of the subject graph.

Minimum delay optimization for variable loads adds only a constant factor to the run-time for tree-covering. If there are n different load values in the technology library, then n cost function evaluations are done for each match. The complexity of this algorithm, therefore, is $O(spn)$. Once again, p and n are constants dependent only on the technology library, and hence the algorithm is linear in the size of the subject graph.

Improved Tree-Matching Techniques

In [41], Hoffmann and O'Donnell explore different algorithms for solving the pattern-matching problem on trees. The algorithms they consider have varying tradeoffs of preprocessing time, matching time, and data space to find all matches of a collection of tree patterns on a subject tree.

graph_match, when restricted to trees, is considered in their paper and is called the *naive* algorithm. It is the algorithm against which more sophisticated pattern-matching algorithms are compared. Note that although many algorithms are given in their paper, there is no single best matching algorithm because of the tradeoffs possible based on the characteristics of the patterns.

As an example of a more sophisticated tree-matching algorithm, consider the *top-down matching algorithm* which is used by the program *twig*. Note that the patterns for top-down matching are the *nonexpanded* pattern set; hence, the children of each node are explicitly ordered. Top-down matching begins with the following preprocessing step. Each node of each pattern is labeled with the type of the node (i.e., for our application, the number of children), and each edge is labeled with the number of the child (i.e., for technology mapping, an edge from a degree-one node is labeled 0, and the left edge from a degree-two node is labeled 0, and the right edge from a degree-two node is labeled 1). Each path from the root of the tree to a leaf is translated into a string which is the concatenation of the node labels and the edge labels along the path from the root to the leaf. Each nonexpanded pattern tree is represented by the collection of all strings from the root to each leaf. A counter is maintained at each node in the subject graph for each pattern. Starting from a fixed node in the subject tree, a preorder traversal of the subject tree creates a string which is matched against the collection of all strings for all patterns. A fast string-matching algorithm, such as the Aho-Corasick algorithm [5], the Knuth-Morris-Pratt algorithm [50], or the Boyer-Moore algorithm [12] is used to determine when the string from the subject graph matches a string in the pattern set. When a successful string match is made, a counter for the corresponding match is incremented; when the count reaches the number of strings in the pattern tree, a match of the corresponding pattern tree has been found. String matching against a large collection of strings can be done efficiently. For example, using the Aho-Corasick string-matching algorithm, it is possible to find all of the strings which match a given string in time proportional to the length of the longest string in

the pattern set. This algorithm provides a significant improvement over the *naive* algorithm for finding all matching strings.

A different algorithm, called the *bottom-up matching algorithm*, improves on this by providing a matching time of $O(s + m)$, where s is the number of nodes in the subject graph, and m is the number of successful matches over the entire tree. The factor s comes from visiting each node once, and the factor m comes from outputting each successful match. Note that the run-time for the bottom-up algorithm is independent of the size of the set of patterns. Intuitively, this is about the best one could hope to do. However, this algorithm has exponential space complexity for some sets of tree patterns.

For the application of tree-covering for technology mapping, the first priority was to understand the strengths and weaknesses of the technique. Execution time was a concern, but was secondary to the quality of results produced by the algorithm. Therefore, the *naive* tree-matching algorithm was used in the implementation. This led to some interesting results which are described below.

First, the naive tree-matching algorithm terminates quickly for many patterns, leading to a sublinear run-time. The worst case complexity of $O(sp)$ is rarely achieved. Further, the optimal tree-covering algorithm complexity consists of two components: (1) the time required to generate all matches at a given node, and (2) the time required to evaluate the cost function for all matches at a node. For a cost function such as minimum area, the time needed to generate the matches dominates the run time. However, for the problem of minimum area under a delay constraint, the limiting factor is the number of cost function evaluations required for each match.

Simple experiments demonstrate that the constant p is small enough so that the naive matching algorithm is sufficient for technology mapping using most libraries. For example, there are 15 gates, 19 patterns and 29 expanded patterns in the IWLS-87 library. For a total tree size of 2,425 nodes, representing an 800 gate-equivalent circuit, it takes 27 seconds on a MicroVax-II to generate 5,686 matches.

For a larger library consisting of all two-level CMOS gates with no more than four series transistors there are 131 gates, 1,171 NAND-gate patterns and 68,689 patterns in expanded form. For the same 800 gate circuit, matching requires 1,340 seconds on a MicroVax-II to generate the 6,948 matches. Note that the number of patterns increased by a factor of 62, and the number of expanded patterns increased by a factor of almost 2,400, but the run-time increased by only a factor of 50. This justifies the claim that many

attempted matches are terminated early by the naive matching algorithm. Simple pruning heuristics in the matcher, such as not attempting to match a tree with an inverter at the root at a two-input node, could further improve the matching time by a constant factor.

For the largest library considered here (the set of all CMOS series-parallel gates with no more than four transistors in series from output to power supply), there are 3,503 gates, 12,631 two-input NAND-gate patterns, and 692,643 expanded patterns, The matching time for this library is probably too slow to be practical; for the same 800 gate circuit, 13,400 seconds are required to generate the 8,902 matches. For matching with a large library, it would be desirable to explore more sophisticated tree-matching algorithms, such as the top-down matching algorithm described earlier, to perform the tree-matching.

4.6 Complete Complex-Gate CMOS Libraries

A CMOS complex-gate is an interconnection of CMOS transistors which implements a more complex logic function than a simple NAND or NOR. Every connection of transistors which satisfies a simple rule, such as no more than four transistors in series from each power supply to the output, is a legal gate. Therefore, to the designer, a complete complex-gate library is a seemingly open-ended collection of gates. Techniques have been developed to perform the physical design steps to build a complex-gate from its transistor graph [74]. At layout time, if a gate is used which has not yet been created, a module generator is called to produce the layout for that gate.

The open-ended nature of a complex-gate library differs from the concept of a fixed technology library as presented earlier. Therefore, the extra problems of automated technology mapping for complete CMOS complex-gate libraries is considered in this section. The approach taken is simple – a complex-gate library is viewed as a fixed library where every legal complex gate is represented explicitly in the library. This allows the DAG-covering approach as presented so far to be directly applied.

To justify this approach, the following questions must be addressed:

- How many gates are there in a practical CMOS complex-gate library ?
- How many patterns are required for the DAG-covering algorithm ?

4.6.1 CMOS Complex Gates

In static CMOS, a logic gate is built by joining an NMOS (pull-down) network to a PMOS (pull-up) network at a common node. Each transistor network is built from source-drain connections of MOS transistors, and each transistor gate is driven by an input to the logic gate. The pull-down network has a special node *Gnd* connected to the circuit ground, and the pull-up network has a special node V_{dd} connected to the circuit power supply. The node common to the two networks is the gate output. Figure 4.26 gives an example of a CMOS complex gate.

In order for the gate to function properly, it is assumed that a conducting path from *Gnd* to V_{dd} does not exist for any combination of inputs. This condition is satisfied if the pull-up network is the graph dual of the pull-down network. Note that if a gate has a pull-up network which is not the dual of the pull-down network, then there exists a gate which computes the same function, which has dual pull-up and pull-down network, and no more transistors. This is clear from the observation that if the pull-up network and pull-down networks are not dual networks, than the larger network can be replaced by the dual of the smaller network without changing the circuit operation.

In this section, only series-parallel transistor networks are considered. A transistor network is series-parallel if it can be constructed from the following definition.

Definition 4.6.1 *A* series-parallel transistor network *is defined as follows. A single transistor is a series-parallel network. A series connection of two series-parallel networks is a series-parallel network. A parallel connection of two series-parallel networks is a series-parallel network.*

The CMOS gate shown in Figure 4.26 is constructed from two series-parallel networks.

CMOS complex-gates are of interest because a logic function can be implemented more efficiently using a single complex-gate, rather than using a network of simpler gates. For example, the complex gate AOI4444[2] can be built in a single CMOS gate using sixteen NMOS and sixteen PMOS transistors. Building the same logic function from simple gates requires five four-input NAND gates and an inverter, or twenty-one NMOS and PMOS transistors.

[2]Logic function $f = \overline{abcd + efgh + ijkl + mnop}$.

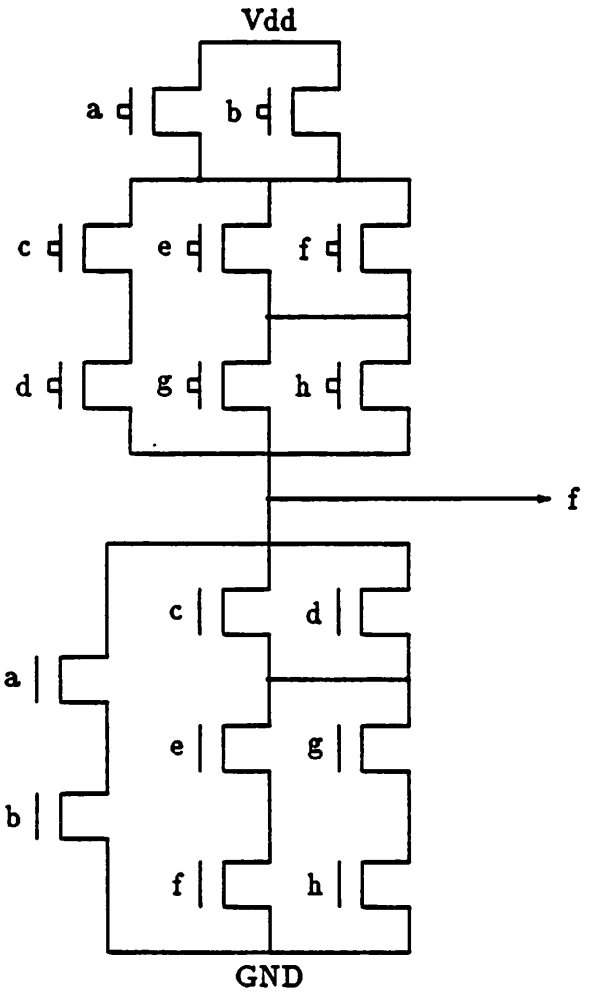

Figure 4.26: A CMOS complex gate.

Using a standard-cell layout style, the minimum width of a CMOS gate is determined by the number of inputs and outputs of the gate (that is, the number of tracks required to route all of the signals to and from the gate). The height is the same for all gates in a row, and hence is fixed. Therefore, the layout of the single complex gate for f could be as small as seventeen tracks wide; the layout using simple gates requires twenty-seven tracks. Substantial savings in area (forty percent in this simple example) can be realized using complex gates. For these reasons, there is interest in algorithms for technology mapping using a CMOS complex-gate library.

4.6.2 Counting the Number of CMOS Complex Gates

With the restriction that the pull-up and pull-down networks are series-parallel, and that the pull-up and pull-down networks are dual networks, a CMOS complex gate can be represented by a single AND-OR tree.

Definition 4.6.2 *An* AND-OR tree *is a labeled tree where each node is a leaf, or has 2 or more children. Each non-leaf node is labeled as an* AND *node or an* OR *node. The node labels alternate between* AND *and* OR*along every path from the root to a leaf.*

Definition 4.6.3 *The* height *of an* AND-OR *tree is the maximum number of nodes along any path from the root to any leaf (excluding the leaf node).*

Definition 4.6.4 *The* dual *of an* AND-OR *tree is the tree resulting from replacing each* AND*-node with an* OR*-node, and each* OR*-node with an* AND*-node.*

An AND-OR tree represents the pull-down network of a CMOS complex-gate where an AND node represents a series connection of its children, and an OR node represents a parallel connection of its children. In a similar manner, the dual of the tree defines the series-parallel construction of the PMOS transistor network in the pull-up part of the gate. Because the AND-OR tree represents the pull-down network in the gate, the logic function implemented by the gate is the complement of the logic function formed by viewing the AND-OR tree as an expression tree.

Definition 4.6.5 *A* CMOS *complex gate is said to satisfy an* (s,p)-constraint *if the gate has at most s transistors from the output to ground, and at most p transistors from the output to the power supply. If s and p are the minimum values for which a gate satisfies an (s,p)-constraint, then the gate is called an* (s, p)-gate.

Given an AND-OR tree, it is easy to compute the values of s and p for which the corresponding gate is an (s,p)-gate. Let T be an AND-OR tree, and let $T_i, i = 1 \ldots n$ be the children of T. Let $s(T)$ be the maximum number of series NMOS transistors in the network built from T. Then,

$$s(T) = \begin{cases} \sum_{i=1}^{n} s(T_i) & \text{if } T \text{ is an AND node} \\ \max_{i=1}^{n} s(T_i) & \text{if } T \text{ is an OR node} \end{cases}$$

Because the PMOS transistor tree is the dual of the NMOS transistor tree, the number of series transistors in the PMOS tree is easily computed from T by reversing the actions for AND nodes and OR nodes.

The AND-OR tree for the CMOS complex gate presented in Figure 4.26 is shown in Figure 4.27. The height of this tree is 4. The gate has 3 devices in series from the output to each supply, and hence is a (3,3)-gate.

Two CMOS complex gates are considered equivalent if the labeled AND-OR trees of the two gates are isomorphic. The problem of counting CMOS complex gates is to determine the number of non-equivalent gates.

Definition 4.6.6 *Let $G(s,p)$ be the set of all nonequivalent series-parallel gates which satisfy a (s,p)-constraint.*

The goal is a formula for $|G(s,p)|$, and an algorithm to efficiently enumerate all of the gates.

Definition 4.6.7 *Let $A(s,p)$ be the subset of $G(s,p)$ excluding those trees which have a root which is an OR-node. Let $O(s,p)$ be the subset of $G(s,p)$ excluding those trees which have a root which is an AND-node.*

Clearly, $G(s,p) = A(s,p) + O(s,p) - 1$, with the extra 1 coming from the overlap between $A(s,p)$ and $O(s,p)$ which is the tree consisting of a single leaf (i.e., in CMOS, an inverter).

Lemma 4.6.1 $|A(s,p)| = |O(p,s)|$.

Proof. By duality, any tree which satisfies an (s,p)-constraint with an AND-node as a root has a dual tree which satisfies an (p,s)-constraint with an OR-node as a root. Hence, the two sets can be placed in one-to-one correspondence. □

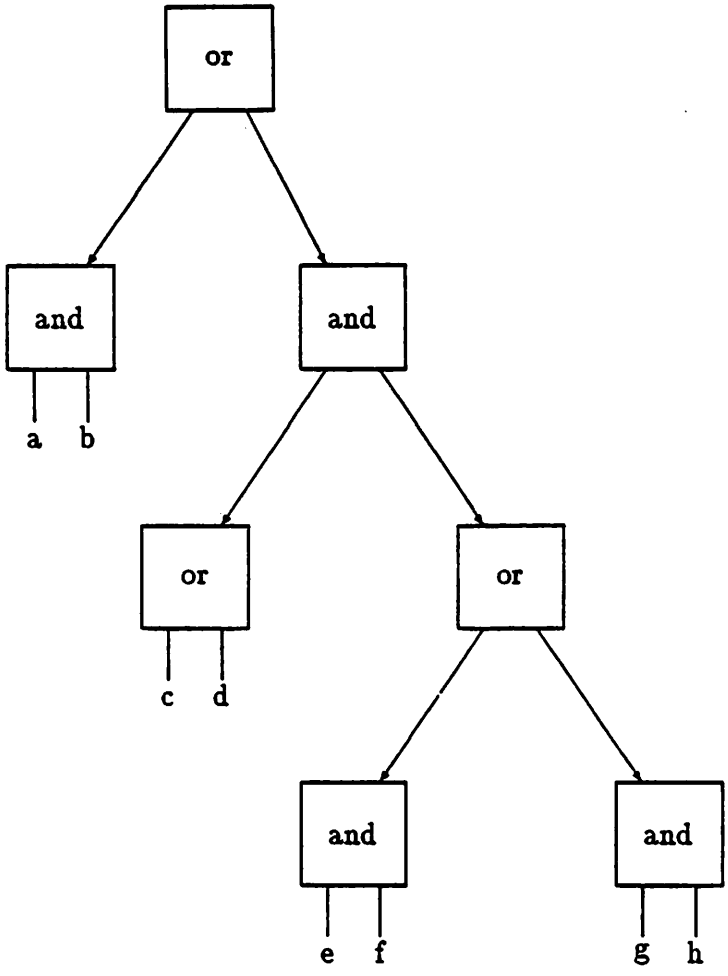

Figure 4.27: AND-OR tree for $ab + (c + d)(ef + gh)$.

Lemma 4.6.2 *If $s_1 \leq s_2$, then $O(s_1, p) \subseteq O(s_2, p)$.*

Proof. Recall that an (s, p)-gate has *at most* s devices in series. Hence, any tree which satisfies an (s_1, p)-constraint also satisfies an (s_2, p)-constraint for $s_1 \leq s_2$. □

Definition 4.6.8 *Given a nondecreasing sequence of integers $0 < x_1 \leq \cdots \leq x_k$, define $h(x_1, \ldots, x_k)$ as the number of nondecreasing sequences $(y_1, \ldots, y_k)$, $0 \leq y_1 \leq \cdots \leq y_k$, satisfying $y_i < x_i, i = 1, \ldots, k$.*

Example 4.6.1 *$h(2,2,3) = 7$ as there are 7 nondecreasing sequences ((0,0,0), (0,0,1), (0,0,2), (0,1,1), (0,1,2), (1,1,1), (1,1,2)) which satisfy the constraint.*

Lemma 4.6.3 *h satisfies the recurrence relation:*

$$\begin{aligned} h(x_1, \ldots, x_k) &= \sum_{i=0}^{x_1 - 1} h(x_2 - i, x_3 - i, \ldots, x_k - i) \\ h(x_1) &= x_1 \end{aligned}$$

Proof. Fix the first element at value i. For the sequence to be nondecreasing, there are i fewer choices for each position after the first. The problem is then reduced to generating all nondecreasing sequences of length $k-1$, from the sequence $(x_2 - i, \ldots, x_k - i)$. The recursion terminates when there is a single element in the sequence. □

Definition 4.6.9 *A* partition *of an integer s, represented $\pi(s)$, is a nondecreasing sequence of integers $0 < s_1 \leq \ldots \leq s_k$ such that $\sum_{i=1}^{k} s_i = s$).*

Theorem 4.6.1

$$|A(s,p)| \leq \sum_{\pi(s)} h(|A(p, s_1)|, \ldots, |A(p, s_k)|)$$

where $\pi(s)$ ranges over all partitions of the integer s and h is as defined in Definition 4.6.8.

Proof. Assume that an AND-OR tree T with an AND-root has s transistors from output to ground, and p transistors from output to supply. Then, the series constraint of

s transistors may be split among the children of T (which must be OR-trees) according to the partitions of the integer s. For any partition $0 < s_1 \leq \cdots \leq s_k$, $O(s_1, p) \subseteq O(s_2, p) \subseteq \cdots \subseteq O(s_k, p)$. To form an AND-OR tree, choose 1 subtree from each of the sets $O(s_i, p)$. However the inclusion between the sets $O(s_i, p)$ leads to overcounting unless the number of ways to form a unique AND-OR tree from the sets $O(s_i, p)$ is considered.

Let $x_i = |O(s_i, p)|$ for $i = 1, \ldots, k$. For each $i, i = 1, \ldots, k-1$, order the trees of the set $O(s_{i+1}, p)$ so that the first x_i trees are in one-to-one correspondence with the trees $O(s_i, p)$. For a given partition of s, a unique tree is formed by choosing a sequence of integers $0 \leq y_1 \leq \cdots \leq y_k$ such that $y_i < x_i, i = 1, \ldots, k$. Identify each y_i as the choice of the tree numbered y_i from the set $O(s_i, p.)$ Hence, the function h counts the number of choices from the overlapping sets $O(s_i, p)$.

By Lemma 4.6.1, $O(s_i, p) = A(p, s_i)$, for $i = 1, \ldots, k$. □

The number of children at each node equals the number of elements in the partition $\pi(s)$. Therefore, the trees computed for two partitions of different length are unique. However, theorem 4.6.1 overcounts the number of trees because of overlap between partitions of s which are of the same length. This is best demonstrated by an example. Consider the case of $s = 4$ with the partitions $s_1 = 1, s_2 = 3$ and $s_1 = 2, s_2 = 2$. The set of trees formed by choosing one tree from $O(1, p)$ and one tree from $O(3, p)$ overlaps the set of trees formed by choosing one tree from $O(2, p)$ and one tree from $O(2, p)$. This overlap is precisely the set of trees formed by choosing one tree from $O(1, p)$ and $O(2, p)$ (that is, the element-wise minimum of the partitions $(1, 3)$ and $(2, 2)$ which is $(1, 2)$).

Theorem 4.6.1 can be modified to count the number of AND-OR trees without overlap. For all partitions of length k, taken two at a time, let t_i be the element-wise minimum of the corresponding elements of each partition. There are

$$h(|A(p, t_1)|, \ldots, |A(p, t_k)|)$$

trees which have been counted twice for this partition. This must be subtracted from $|A(s, p)|$. For all partitions of length k, taken three at a time, let t_i be the element-wise minimum of the corresponding elements of each partition. Then a value of

$$h(|A(p, t_1)|, \ldots, |A(p, t_k)|)$$

	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	7	18	42	90	186
3	3	18	87	396	1,677	6,877
4	4	42	396	3,503	28,435	222,943
5	5	90	1,677	28,435	425,803	6,084,393
6	6	186	6,877	222,943	6,084,393	154,793,519

Table 4.2: Number of gates satisfying an (s, p)-constraint.

must be added to $|A(s,p)|$ to account for the trees which are in the overlap of the partitions taken three at a time. Continue in this manner, alternating sign, until all possible subsets of the partitions of a given length are considered.

Table 4.2 gives the number of trees which satisfy an (s, p)-constraint for values of s and p ranging up to six. A typical maximum value for CMOS designs is $s = 4$ and $p = 4$ for which there are 3,503 complex-gates. Note that the function G is symmetric in s and p.

Table 4.3 lists the 87 unique gates which satisfy a (3,3)-constraint. l gives the height of the corresponding AND-OR tree, s gives the series height of the NMOS transistors, and p gives the series height of the PMOS transistors.

4.6.3 Counting the NAND-Gate Trees for a Function

For optimal results, the tree-covering algorithm requires that each logic function be represented by all of the two-input NAND-gate trees that realize that function. For example, a five-input NAND gate has three unique NAND-gate trees, and the complex gate AOI4444 has eighteen unique NAND-gate trees. The problem considered in this section is how to generate these trees given a description of the logic function for a gate.

The first step is to transform the logic function into an AND-OR tree. This is done by translating the logic function in the library into an expression tree, and then converting the expression tree into normal form where each level of the tree is a different operator. The phase of each input (i.e., normal or complemented) to the expression tree is recorded. During the recursive algorithm, when a tree is created for the leaf, either the leaf or an inverter

l	s	p	function	l	s	p	function
0	1	1	a				
1	2	1	ab	1	1	2	$a+b$
2	2	2	$a(b+c)$	2	2	2	$a+bc$
3	3	2	$a(b+cd)$	3	2	3	$a+b(c+d)$
4	3	3	$a(b+c(d+e))$	4	3	3	$a+b(c+de)$
4	3	3	$a(b+(c+d)(e+f))$	4	3	3	$a+b(cd+ef)$
3	3	2	$a(bc+de)$	3	2	3	$a+(b+c)(d+e)$
4	3	3	$a(bc+d(e+f))$	4	3	3	$a+(b+c)(d+ef)$
4	3	3	$a(bc+(d+e)(f+g))$	4	3	3	$a+(b+c)(de+fg)$
2	2	3	$a(b+c+d)$	2	3	2	$a+bcd$
3	3	3	$a(b+c+de)$	3	3	3	$a+bc(d+e)$
3	3	3	$a(b+cd+ef)$	3	3	3	$a+b(c+d)(e+f)$
3	3	3	$a(bc+de+fg)$	3	3	3	$a+(b+c)(d+e)(f+g)$
2	2	2	$(a+b)(c+d)$	2	2	2	$ab+cd$
3	3	2	$(a+b)(c+de)$	3	2	3	$ab+c(d+e)$
4	3	3	$(a+b)(c+d(e+f))$	4	3	3	$ab+c(d+ef)$
4	3	3	$(a+b)(c+(d+e)(f+g))$	4	3	3	$ab+c(de+fg)$
3	3	2	$(a+b)(cd+ef)$	3	2	3	$ab+(c+d)(e+f)$
4	3	3	$(a+b)(cd+e(f+g))$	4	3	3	$ab+(c+d)(e+fg)$
4	3	3	$(a+b)(cd+(e+f)(g+h))$	4	3	3	$ab+(c+d)(ef+gh)$
2	2	3	$(a+b)(c+d+e)$	2	3	2	$ab+cde$
3	3	3	$(a+b)(c+d+ef)$	3	3	3	$ab+cd(e+f)$
3	3	3	$(a+b)(c+de+fg)$	3	3	3	$ab+c(d+e)(f+g)$
3	3	3	$(a+b)(cd+ef+gh)$	3	3	3	$ab+(c+d)(e+f)(g+h)$
3	3	3	$(a+bc)(d+e+f)$	3	3	3	$a(b+c)+def$
4	3	3	$(a+b(c+d))(e+f+g)$	4	3	3	$a(b+cd)+efg$
4	3	3	$(a+(b+c)(d+e))(f+g+h)$	4	3	3	$a(bc+de)+fgh$
3	3	3	$(ab+cd)(e+f+g)$	3	3	3	$(a+b)(c+d)+efg$
4	3	3	$(ab+c(d+e))(f+g+h)$	4	3	3	$(a+b)(c+de)+fgh$
4	3	3	$(ab+(c+d)(e+f))(g+h+i)$	4	3	3	$(a+b)(cd+ef)+ghi$
2	2	3	$(a+b+c)(d+e+f)$	2	3	2	$abc+def$
3	3	3	$(a+b+c)(d+e+fg)$	3	3	3	$abc+de(f+g)$
3	3	3	$(a+b+c)(d+ef+gh)$	3	3	3	$abc+d(e+f)(g+h)$
3	3	3	$(a+b+c)(de+fg+hi)$	3	3	3	$abc+(d+e)(f+g)(h+i)$
1	3	1	abc	1	1	3	$a+b+c$
2	3	2	$ab(c+d)$	2	2	3	$a+b+cd$
2	3	3	$ab(c+d+e)$	2	3	3	$a+b+cde$
2	3	2	$a(b+c)(d+e)$	2	2	3	$a+bc+de$
2	3	3	$a(b+c)(d+e+f)$	2	3	3	$a+bc+def$
2	3	3	$a(b+c+d)(e+f+g)$	2	3	3	$a+bcd+efg$
2	3	2	$(a+b)(c+d)(e+f)$	2	2	3	$ab+cd+ef$
2	3	3	$(a+b)(c+d)(e+f+g)$	2	3	3	$ab+cd+efg$
2	3	3	$(a+b)(c+d+e)(f+g+h)$	2	3	3	$ab+cde+fgh$
2	3	3	$abc+def+ghi$	2	3	3	$(a+b+c)(d+e+f)(g+h+i)$

Table 4.3: All (3,3)-gates.

connected to the leaf is returned based on the phase. If a single variable is connected to several leaves of the AND-OR tree, distinct inputs are assumed for the discussion which follows. After the patterns are generated assuming distinct inputs, the primary inputs of the pattern are re-connected based on the connectivity of the original AND-OR tree.

The problem of generating all of the unique trees for a given AND-OR tree is broken into two steps. First, all unique trees for a single level of the AND-OR tree (i.e., for a single n-input AND or OR) are enumerated. Then, using a recursive algorithm, the NAND-gate trees for the children at a given level are combined to generate all unique trees.

NAND-Gate Trees for an n-Input AND Gate

The algorithm presented next enumerates the number of two-input AND-gate trees for an n-input AND-gate. Note that given a tree for an n-input AND-gate, it may be converted into a tree for an n-input OR-gate merely by adding an inverter to each of the inputs, and by adding an inverter to the output. Note also that if NAND-gate trees are used instead of AND-gate trees, then each NAND gate is simply followed by an inverter to form an AND-gate. Therefore, inverters in the tree can be ignored for counting purposes, and the problem is reduced to counting the number of nonisomorphic, unlabeled trees of n leaves where each node is a leaf or has a degree of 2. For each of these trees, there is one way to add inverters to the tree to form a NAND-gate tree for an n-input AND or an n-input OR.

Let $f(n)$ be the number of such trees. If n is 1, there is one tree consisting of a single leaf; hence $f(1) = 1$.

If $n > 1$, assume the root of the tree has i leaves under its first son, and $n - i$ leaves under its second son for $i = 1, \ldots, \lfloor n/2 \rfloor$. (Without loss of generality, assume fewer leaves are under the first son if n is odd.)

If $i \neq n - i$ then none of the $f(i)$ trees for the left son can be isomorphic to any of the $f(n - i)$ trees for the right son because they have a different number of leaves; hence, any tree formed from choosing one tree from the set for left son and one tree from the set for the right son generates a unique tree. The number of nonisomorphic trees in this case is $f(i)f(n - i)$.

If $i = n - i$, there are fewer unique trees. Assume the $f(i)$ trees for each son are ordered. Choosing a tree for the right son with an index equal to or greater than the index of the tree for the left son will guarantee a unique tree. Hence, there are $\frac{1}{2}f(i)(f(i) + 1)$

n	$f(n)$	n	$f(n)$
1		9	46
2	1	10	98
3	1	11	207
4	2	12	451
5	3	13	983
6	6	14	2,179
7	11	15	4,850
8	23	16	10,905

Table 4.4: Number of two-input NAND-gate trees for an n-input AND.

ways to generate a unique tree in this case.

The following recurrence relation computes the number of two-input AND-gate trees for an n-input AND gate:

$$f(n) = \begin{cases} \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} f(i)f(n-i) & \text{if } n \text{ odd} \\ \sum_{i=1}^{\frac{n}{2}-1} f(i)f(n-i) + \frac{1}{2}f(\frac{n}{2})(f(\frac{n}{2})+1) & \text{if } n \text{ even} \end{cases} \quad (4.1)$$

Table 4.4 shows the number of unique two-input NAND-gate trees for values of n up to sixteen. Although the function f grows exponentially in n, only the AND-gates with a small number of leaves are of interest (e.g., $n < 10$). For an eight-input AND-gate, there are only twenty-three two-input AND-trees.

NAND-Gate Trees for an AND-OR Tree

The algorithm **generate_nand_trees**, shown in Figure 4.28, generates all two-input NAND-gate trees for a given AND-OR tree T. **generate_nand_trees** generates the NAND-gate trees with duplication; hence, isomorphic trees are removed at each step of the algorithm.

If T is a leaf, the only NAND-gate tree is the tree with a single leaf (or a single inverter at the leaf). This degenerate case terminates the recursive algorithm.

Otherwise, assume that T has n children. Let R be the set of all two-input NAND-gate trees which implement an n-input AND gate (if T is an AND-node), or the set of all two-

```
generate_nand_trees(T) {

    /* recursive termination -- a single leaf */
    if (T is a leaf) {
        if (T is inverted)
            return single inverter fed by T;
         else
            return T;


    } else {

        /* generate all n-input NAND trees for the current node */
        R = generate_nand_trees_single(T->type, n);

        /* T[i] represents child number i of T */
        for i from 1 to n {
            M[i] = generate_nand_trees(T[i]);
        }

        set new_trees to be empty;
        foreach r in R {
            foreach selections of a tree Si from Mi, i = 1 ... n {
                forall permutations of (S1, S2, ..., Sn) {
                    append Si to each leaf of r to get new_tree;
                    add new_tree to new_trees;
                }
            }
        }

        new_trees = delete_duplicate_trees(new_trees);
        return new_trees;
    }
}
```

Figure 4.28: Algorithm generate_nand_trees.

	1	2	3	4
1	1	2	3	5
2	2	7	20	70
3	3	20	115	836
4	4	70	836	12,631

Table 4.5: Number of NAND-gate patterns for (s,p)-gates.

input NAND-gate trees which implement an n-input OR-gate (if T is an OR-node). These are computed using **generate_nand_trees_single**, based on the recurrence relation presented in equation 4.1. Let M_i be the set of all NAND-gate trees for each son T_i, $i = 1, \ldots, k$ of T computed recursively using this procedure. For each $r \in R$, select one tree S_i from each set M_i. For each permutation π of the elements $1, \ldots, n$ form the tree with root r and leaves $S_{\pi(i)}$. The set of trees formed by considering all possible $r \in R$, all possible selections of trees $S_i \in M_i$, and all permutations of the n leaves of r includes all possible NAND-gate implementations for T.

This enumeration algorithm has been implemented and used to count the number of two-input NAND-gate trees for a library of (s,p)-gates for different values of s and p. These results are shown in Table 4.5. A value typical for CMOS design is (4,4), of which there are 3,503 different complex gates, and 12,631 different NAND-gate trees.

Clearly, the worst case performance of this procedure is bad. Using f as defined in 4.1, there are

$$f(k)(k!)\prod_{i=1}^{k}|M_i|$$

different trees to examine. However, the straightforward application of this enumeration procedure is sufficient to generate patterns for common CMOS complex gates. Generation of the 12,631 two-input NAND-gate patterns for all 3,503 (4,4)-gates requires about 12 minutes on a DEC VAX 8650. Of course, this procedure is executed once to create the set of patterns, and is not run for each application of the DAG-covering algorithm.

Interestingly, the (4,4)-function generating the most trees was

$$\overline{abcd + efgh + ij(k+l)(m+n)}$$

which has forty-two two-input NAND-trees.

4.7 Experimental Results

A technology mapping algorithm based on the tree-covering approximation has been implemented in MIS-II. In this section, several experiments with this implementation are reported.

Table 4.6 describes the set of examples which were used for the experiments. Shown in the table are the number of primary inputs (*pi*), the number of primary outputs (*po*), and the total number of literals in sum-of-products form in the network after algebraic decomposition (*literals*).

Ten of the circuits are from the ISCAS Test Generation Benchmark set. These represent, for the most part, large combinational circuit designs which are already represented at the gate level. The examples *misex3* and *rot* are industrial circuits taken from actual chip designs. *seq* is a large finite-state machine from the Berkeley SPUR microprocessor cache controller [29]. *des* is the byte-pipelined data encryption chip described in Chapter 5. Each of these last four examples was translated from a high-level description into a starting form for logic optimization.

All of the results in this section were collected on a Sun 4/260 Computer. A Sun 4/260 is typically ten times faster than a DEC MicroVax-II for integer applications. All run-times are reported in seconds.

4.7.1 Area versus Delay; IWLS-87 Library

The first experiment was to map each circuit using the IWLS-87 technology library. Each circuit was mapped once for minimum area and once for minimum delay. **optimal_area_cover** was used for the area result, and **optimal_delay_onesize_cover** was used for the minimum delay result. Note that, for this library, all gates except the three high-drive inverters have the same input pin load; therefore, the use of load bins as in **optimal_delay_cover** would not improve the results dramatically. The results are presented in Table 4.7.

In the table, *gates* is the number of gates in the final circuit, *area* is the total cell

Example	pi	po	Literals
C432	36	7	264
C499	41	32	558
C880	60	26	421
C1355	41	32	558
C1908	33	25	543
C2670	233	140	764
C3540	50	22	1,385
C5315	178	123	1,838
C6288	32	32	3,762
C7552	207	108	2,332
des	256	245	3,655
misex3	14	14	629
rot	135	107	803
seq	42	35	1,788

Table 4.6: Benchmark circuits for technology mapping.

area in the final circuit, and *delay* is the critical path delay in nanoseconds (ns). In the IWLS-87 library, an inverter is 1 gate, 1 unit of area, and has a delay of 1.2 ns; a 2-input NAND-gate is 1 gate, 2 units of area, and also has a delay of 1.2 ns. The area optimization results include a global phase assignment for area step applied after the technology mapping [76]. Phase assignment was not performed for the delay optimized circuits as MIS-II lacks a global phase assignment algorithm for delay. The cpu times for technology mapping (area mode) and phase assignment (area mode) are given in the last two columns; this is in seconds of computer time on a Sun 4/260. The run-time for both the area and delay modes is identical. This is because, in the implementation, the area and delay cost functions are computed regardless of the optimization criteria. This is done so that, given equal area covers, the faster cover can be preferred. Therefore, only the cpu time for the area mode is shown.

The first thing to notice is that the tree-based technology mapping algorithm is fast. The entire set of fourteen benchmark examples, representing 8,071 gates and 23,886 units of gate area required only 92 seconds on the Sun 4/260 for optimization. The phase assignment which followed required an additional 11.8 ns.

Example	Area Mode			Delay Mode			Cpu	Cpu
	Gates	Area	Delay	Gates	Area	Delay	Map	Phase
C432	127	356	67.8	195	449	60.2	3.1	0.5
C499	210	677	41.5	328	922	36.5	6.3	0.9
C880	225	573	44.8	282	701	38.7	5.2	0.9
C1355	210	677	41.5	328	922	36.5	6.4	0.9
C1908	228	672	59.2	308	859	54.3	5.9	0.9
C2670	328	934	59.3	470	1,233	47.7	9.7	1.5
C3540	606	1,646	77.1	823	2,111	68.3	15.8	3.4
C5315	753	2,289	69.2	960	2,848	57.6	22.4	3.3
C6288	1,860	4,214	204.0	2,636	5,560	171.0	36.7	6.4
C7552	883	2,869	90.3	1,227	3,424	60.5	29.5	5.8
des	1,886	4,902	99.5	2,352	5,902	114.4	55.1	11.8
misex3	321	796	44.8	433	1,026	36.3	7.5	2.2
rot	439	1,006	47.2	553	1,342	41.0	9.2	2.2
seq	895	2,275	49.7	1,169	2,843	39.0	21.7	8.5
Total	8,971	23,886	995.9	12,064	30,142	862.0	92.5	11.8
Ratio	1.00	1.00	1.00	1.34	1.26	0.86		

Table 4.7: Area versus delay for the IWLS-87 Library.

The second point to notice is that delay optimization produced circuits which averaged 26% larger than the area circuits, but with a critical path delay that was 14% less. Because the simple **optimal_delay_onesize_cover** algorithm was used, no attempt was made to reduce the area of the circuit off of the critical path. In fact, for the delay optimized circuits, *all* paths have been mapped for optimum delay. This accounts for the large increase in area.

Note that delay optimization during technology mapping for this library is limited. First, technology mapping does not restructure or rebalance the circuit to speed up the critical path. Second, all of the gates in this library, except the inverters, come in a single size. Hence, the performance improvement for these examples comes from the selection of the correct gates along the critical path.

4.7.2 Area versus Delay; IWLS-89 Library

The International Workshop on Logic Synthesis Benchmark Committee has defined a new library for use in the upcoming 1989 workshop. This standard-cell library includes more CMOS complex gates and contains more detailed timing information for each gate in the library, including separate rise and fall delays. The library has 27 gates. The MIS-II

Example	Area Mode			Delay Mode			Cpu	Cpu
	Gates	Area	Delay	Gates	Area	Delay	Map	Phase
C432	122	449	38.4	163	580	38.8	3.7	0.5
C499	209	810	29.9	322	1,172	22.9	7.0	0.8
C880	206	729	26.9	292	1,015	27.8	6.0	0.8
C1355	209	810	29.9	322	1,172	22.9	7.1	0.8
C1908	235	847	37.9	339	1,187	35.0	6.7	0.8
C2670	341	1,242	31.1	497	1,796	30.0	11.1	1.4
C3540	536	2,052	49.9	781	2,789	47.7	18.7	1.9
C5315	735	2,836	43.2	935	3,611	38.4	25.8	2.7
C6288	1,328	4,993	101.0	2,680	8,757	112.7	41.1	7.8
C7552	868	3,414	56.7	1,323	4,880	44.7	33.1	6.0
des	1,695	6,290	89.6	2,168	7,879	79.9	64.7	7.8
misex3	288	1,044	27.0	388	1,346	21.0	8.8	1.4
rot	414	1,378	27.7	507	1,757	28.0	10.5	1.7
seq	770	2,895	25.0	1,019	3,645	23.0	25.7	5.1
Total	7,956	29,789	614.2	11,736	41,586	572.8	270.0	39.5
Ratio	1.00	1.00	1.00	1.48	1.40	0.93		

Table 4.8: Area versus delay for the IWLS-89 library.

description of this library is reproduced at the end of this chapter.

The area cost values for this library are given in square microns. All area values are divisible by 464 which gives the number of pins on the cell. The number 464 probably represents the uniform height of all of the cells times the width per pin; i.e., the cells are 58 microns high and the width per input of the cell is 8 microns. To make the reporting of results easier, the cell area is reported after division by 464. Note also that for this library, the cell area cost function is the number of pins on the gate (input pins plus the output pin). Therefore, an inverter has an area cost of 2, and a four-input NAND-gate has an area cost of 5. This differs from the IWLS-87 library which used the number of input pins only as the primary cost function. Because all gates are single-output, the difference between these two cost functions is simply the number of gates in the design. The area versus delay experiments were repeated for this library, and the results are presented in Table 4.8.

The run-time for technology mapping has increased due to the increased number of gates in the library and the number of tree-patterns required to represent these gates.

The gates in this library have distinct input pin loads for each pin on each gate. This causes the simple **optimal_delay_onesize_cover** algorithm some difficulty. Nonetheless, the delay optimized circuits using this library are 7% faster than the area optimized

Example	22-1	22-2	33-1	33-2	33-4	44-1	44-2	44-3
Gates	3	7	5	33	87	7	131	625
Patterns	3	7	5	49	115	9	1,171	4,513
C432	719	557	629	473	471	571	407	413
C499	1,432	1,016	1,420	1,004	1,002	1,420	1,004	1,002
C880	1,115	883	1,017	771	775	959	745	737
C1355	1,432	1,016	1,420	1,004	1,002	1,420	1,004	1,002
C1908	1,341	1,035	1,289	983	965	1275	963	945
C2670	2,100	1,396	1,938	1,340	1,306	1,822	1,242	1,192
C3540	3,306	2,514	3,034	2,240	2,174	2,966	2,150	2,038
C5315	4,698	3,256	4,576	3,046	2,986	4,466	2,932	2,916
C6288	8,003	6,863	7,135	5,875	5,883	7,135	5,865	5,883
C7552	5,889	4,073	5,781	3,925	3,873	5,733	3,889	3,827
des	9,950	6,974	8,936	6,222	6,196	8,634	5,844	5,816
misex3	1,565	1,255	1,325	1,023	989	1,283	975	953
rot	1,940	1,556	1,732	1,352	1,304	1,708	1,318	1,278
seq	4,466	3,500	3,830	2,872	2,782	3,702	2,718	2,646
Total	47,956	35,894	44,062	32,130	31,708	43,094	31,056	30,648
Ratio	1.00	0.75	0.92	0.67	0.66	0.90	0.65	0.64

Table 4.9: Technology mapping with different libraries.

circuits. Once again, the large area increase is due to not optimizing for area off of the critical path. It is expected that the load-binning strategy of **optimal_delay_cover** could improve these results significantly.

4.7.3 Technology Mapping Library Comparison

The next experiment was to map the same set of fourteen circuits using eight different libraries. Each library is annotated as $sp-l$ where s is the maximum number of series NMOS transistors, p is the maximum number of series PMOS transistors, and l is the height of the AND-OR tree. Note that 22-2 (33-4) represents the complete set of static CMOS gates with no more than 2 (3) transistors in series. Also, the 22-1, 33-1, and 44-1 libraries represent libraries of just NAND-gates and NOR-gates.

The assumption was made that the area for each cell was equal to the number of pins on the cell. Within AT&T, this is known as the number of *grids* for the cell. In a standard-cell environment, this reflects a best-case assumption; the minimum width of the cell is dictated by the number of connections to and from the cell. The results for this experiment, reported in cell area, is shown in Table 4.9.

The interesting conclusion from this experiment is that the larger libraries, although they contain significantly more gates, do not provide a large improvement over much simpler libraries. For example, the circuits designed using 33-4 (with only 87 gates) are only 3.5% larger than the circuits designed using 44-3 (with 625 gates). Strangely enough, the results for the IWLS-89 benchmark library, with only 27 gates, are superior to the results for the largest library attempted in this section (29,789 grids versus 30,648 grids). The reason for this is that the IWLS-89 library contains an exclusive-or gate with a low relative cost of only 5 grids. Several of the circuits (e.g., C1355) contain a large number of exclusive-or gates and benefit greatly from the use of this gate. Building an exclusive-or gate out of the gates available in the other libraries requires a minimum of 7 grids. In fact, the improvement for C1355 alone is almost 20% over the best result for the libraries in this section.

4.7.4 Inverter Optimization Heuristics

The final experiment compares the effects of the inverter heuristics described earlier. Recall that two separate heuristics are used. The first replaces each wire in the circuit with a double inverter pair. This is called the inverter-pair heuristic. The second records, at the branch points, information concerning the presence of an inverter at the branch point. This is called the branch-point heuristic.

The inverter-branch point heuristic is an attempt to capture knowledge of the underlying graph during the application of the tree-covering algorithm. A second technique which would seem to achieve the same goal is to use an algorithm optimized for inverter optimization on a graph. Solutions to this problem, known as the global phase assignment problem, have been implemented in MIS-II.

The same set of fourteen circuits was mapped using the IWLS-87 library for minimum area. The experiment was performed first without any inverter heuristic, then with the inverter-pair heuristic, and finally with both inverter optimization heuristics. The total circuit area is reported both before and after global phase assignment to explore the interaction between the inverter-optimization heuristics and the step of global phase assignment. The results are shown in Table 4.10.

The inverter-pair heuristic is effective; the total circuit area is reduced by 10% when this heuristic is applied. The results for the inverter-branch point heuristic are more

Example	No Added Inverters		Inverter-pair only		Normal	
	Before Phase	After Phase	Before Phase	After Phase	Before Phase	After Phase
C432	386	353	360	354	358	356
C499	855	833	726	708	692	677
C880	651	624	599	575	577	573
C1355	855	833	726	708	692	677
C1908	793	765	713	688	683	672
C2670	1,225	1,143	1,008	981	957	934
C3540	1,897	1,842	1,700	1,658	1,671	1,646
C5315	2,887	2,810	2,401	2,346	2,323	2,289
C6288	4,720	4,719	4,499	4,468	4,230	4,214
C7552	3,733	3,550	2,935	2,875	2,906	2,869
des	5,183	5,038	5,057	4,953	5,004	4,902
misex3	881	845	835	807	820	796
rot	1,138	1,087	1,060	1,023	1,029	1,006
seq	2,446	2,389	2,380	2,284	2,363	2,275
Totals	27,650	26,831	24,999	24,428	24,305	23,886
Ratio	1.00	0.97	0.90	0.88	0.88	0.86

Table 4.10: Effect of inverter-pair and inverter-branch-point heuristics.

interesting. Note that using the inverter-branch point heuristic effectively performs much of the optimization which was performed by the global phase assignment. The interesting point is that global phase assignment, applied after the inverter-branch point heuristic, continues to improve the optimization result.

4.8 Extensions and Open Issues

A number of extensions to the general DAG-covering approach are described in this section.

4.8.1 Sequential Technology Mapping

A technology library typically has a large number of sequential components. For example, three popular storage elements are D flip-flops, JK flip-flops, and T flip-flops. Each flip-flop type has many variations including a load enable input, and set and clear inputs. Sequential technology mapping extends the technology mapping problem so that the optimal choice of the sequential components is also performed. Sequential technology mapping is

done with a simple extension to the DAG-covering algorithm.

To accommodate sequential technology mapping, the model of the subject graph needs to be extended. Pairs of primary inputs and primary outputs of the subject graph are tagged as one-cycle delayed versions of the same signal. That is, in a technology independent way, a D flip-flop is assumed to exist between the primary output and its paired primary input. Assuming the target library has at least a single D flip-flop, it is always possible to implement a given circuit.

Each sequential component is modeled with a combinational logic equation feeding a one-cycle delay element with an input D and an output Q. For example, a D flip-flop with a data pin d, a synchronous reset pin r and a load pin l is modeled with the combinational equation

$$D = dl + Q\bar{l} + r,$$

or a JK flip-flop with pins j and k is modeled with the equation

$$D = j\overline{Q} + \bar{k}Q.$$

For each sequential element, all two-input NAND-gate and inverter patterns are generated for the combinational circuitry. When matches are generated on the subject graph, the D output of a sequential component is required to match at a primary output; if its equation also includes a dependency on a Q signal, then the Q signal is required to match at a primary input which is tagged as a delayed version of the same primary output.

Using this technique, an optimal choice for the flip-flop components can be made at the same time as the choice for the combinational gates.

4.8.2 Multiple-Output Gates

Many libraries have gates which produce multiple logic functions over a set of inputs. These are called multiple-output gates. Typical examples of multiple-output gates include gates with dual polarity (i.e., the gate produces both f and $\overline{f}$ for the logic function f), gates which provide access to internal signals of a complex cell (e.g., an exclusive-or gate which also produces $a\bar{b}$ and $\bar{a}b$ as outputs), and large functionality cells such as an n-bit multiplexor or a $\log_2 n$-bit to n-bit decoder.

In some libraries, multiple-output gates are also defined which combine two functions which do not share any inputs. For example, two independent inverters in a single

gate. In this case, the decision to place two otherwise separate inverters into a single cell should be based on the physical placement of the two inverters and is not a consideration in the logical design of the circuit. Hence, using this style of multiple-output gates is not defined as part of the technology mapping problem.

With the assumption that the subject graph for a multiple-output gate is connected, the DAG-covering formulation for technology mapping, and the DAG-covering algorithm presented in Section 4.4 make optimal use of all of the multiple-output gates in a library.

The tree-covering approximation, on the other hand, can operate only for single-output gates. The proposed technique for dealing with multiple-output gates within tree-covering falls back to a local improvement scheme. First the circuit is mapped using only single-output logic functions from the set of logic functions available as an output of some gate. All of the logic functions which are available from a multiple-output gate are considered as valid functions for this first step. Then, a local improvement scheme is used to identify situations where multiple gates can be combined into a single multiple-output gate.

4.8.3 Extension of the Base-Function Set

One problem with the tree-covering approximation is dealing with circuits containing a large number of exclusive-or gates. As described earlier, using patterns for exclusive-or gates which are leaf-DAG's (fan-out of more than one only at the primary inputs) in the tree-mapping approximation is straightforward. However, when the subject graph is partitioned for tree-covering, each exclusive-or gate appears at a tree boundary. This often leads to a large number of small trees. Because heuristics are used to estimate the phase-assignment across the trees, optimality is lost when the optimal covers for the small trees are combined. Global phase assignment [20] can be used to make up for this loss in optimality; however, better solutions are expected if more information can be presented to the dynamic programming algorithm to optimally choose the phase assignment around exclusive-or gates.

A further problem is that although every function can be represented as a leaf DAG, this is often not the form the function is expected to have in the subject graph. For example, consider a three-input exclusive-or gate. The logic decomposition step which precedes technology mapping can be expected to produce an interconnection of two two-input exclusive-or gates for a three-input exclusive-or. However, the representation of this

form using two-input NAND-gates includes internal nodes with multiple-fanout and hence is not a leaf-DAG. Therefore, it is not possible to build a tree-representation for the form in which the three-input exclusive-or gate is expected to appear in a subject graph.

One proposal for dealing with this problem is to increase the base-function set by adding a two-input exclusive-or gate. The subject graph then consists of two-input NAND-gates, two-input exclusive-or gates, and inverters. The subject graph is put into this form by first transforming the circuit into two-input NAND and inverter form, and then identifying on the subject graph situations where a two-input exclusive-or exists.

Patterns are created for all exclusive-or and exclusive-nor gates, and extra patterns are created for performing optimal phase assignment around each exclusive-or style gate. For example, a two-input exclusive-or gate is represented by three different patterns: the single node pattern, the pattern with both inputs inverted, and a pattern with a single input and the output inverted. A three-input exclusive-nor is similarly represented using a cascade of two two-input exclusive-or gates, and all possible combinations of inverters which leave the circuit function unchanged. The tree-covering algorithms can then be applied to find an optimal cover using the additional patterns.

Note that adding exclusive-or gates to the base function set has the advantage of increasing the size of the trees in the circuit. Therefore, optimal results can be found for larger portions of the circuit. The disadvantage of adding a function to the base-set has been mentioned earlier; the granularity of the base function set controls the solution space. In this case, any circuit derived by covering the NAND-gates within the exclusive-or gate with two different patterns is no longer a possible solution.

4.9 Conclusions

DAG-covering provides an algorithmic formulation to the technology mapping problem. The limitations of DAG-covering, namely its dependence on the form of the subject graph, fit in nicely with the paradigm of combining DAG-covering with technology-independent optimization to provide an overall solution for logic synthesis.

It has been shown how to use the tree-covering approximation to provide an effective solution for the DAG-covering problem, including the problem of delay optimization in technology mapping.

4.10 IWLS-89 Benchmark Library Description

The following is a description of the International Workshop on Logic Synthesis benchmark library for 1989 in MIS-II format.

```
GATE inv1x 928.00       O = ! a;
PIN a INV 0.0514 999.0 0.4200 4.7100 0.4200 3.6000

GATE inv2x 928.00       O = ! a;
PIN a INV 0.1009 999.0 0.3000 1.9800 0.2900 1.8200

GATE inv4x 1392.00      O = ! a;
PIN a INV 0.1897 999.0 0.2300 1.0800 0.2700 0.8500

GATE xor 2320.00        O = ((!a * b) + (a * !b));
PIN a UNKNOWN 0.1442 999.0 1.7700 5.2300 0.9600 4.6400
PIN b UNKNOWN 0.1381 999.0 1.9400 4.6500 1.1400 5.2200

GATE xnor 2320.00       O = ((!a * !b) + (a * b));
PIN a UNKNOWN 0.1502 999.0 1.1100 4.8600 1.0700 3.3900
PIN b UNKNOWN 0.1352 999.0 1.5500 4.8700 1.0700 3.3900

GATE nand2 1392.00      O = ! (a * b);
PIN a INV 0.0777 999.0 0.6400 4.0900 0.4000 2.5700
PIN b INV 0.0716 999.0 0.4600 4.1000 0.3700 2.5700

GATE nand3 1856.00      O = ! (a * b * c);
PIN a INV 0.1000 999.0 0.8900 3.6000 0.5100 2.4900
PIN b INV 0.0828 999.0 0.7100 4.1100 0.4200 2.5000
PIN c INV 0.0777 999.0 0.5600 4.3900 0.3500 2.4900

GATE nand4 2320.00      O = ! (a * b * c * d);
PIN a INV 0.1030 999.0 1.2700 3.6200 0.6700 2.3900
PIN b INV 0.0980 999.0 1.0900 3.6100 0.6100 2.3900
PIN c INV 0.0980 999.0 0.8200 3.6200 0.5500 2.4000
PIN d INV 0.1050 999.0 0.5800 3.6200 0.3800 2.3900

GATE nor2 1392.00       O = ! (a + b);
PIN a INV 0.0736 999.0 0.3300 3.6400 0.4500 3.6400
PIN b INV 0.0968 999.0 0.5000 3.6400 0.7000 3.6600

GATE nor3 1856.00       O = ! (a + b + c);
PIN a INV 0.0856 999.0 0.8400 5.0400 1.3000 3.4500
PIN b INV 0.0806 999.0 0.7800 5.0300 1.1400 3.4300
PIN c INV 0.0826 999.0 0.5200 5.0300 0.8400 3.4400

GATE nor4 2320.00       O = ! (a + b + c + d);
PIN a INV 0.0887 999.0 0.4100 5.9100 1.1600 3.2000
PIN b INV 0.0867 999.0 0.8500 5.9100 1.5300 3.1800
PIN c INV 0.0867 999.0 1.1100 5.9200 1.7500 3.1900
```

```
PIN d INV 0.0887 999.0 1.2700 5.9100 1.9400 3.2000

GATE aoi21 1856.00      O = ! ((a1 * a2) + b);
PIN a1 INV 0.1029 999.0 0.7500 3.5200 0.6700 2.5300
PIN a2 INV 0.0908 999.0 0.6700 3.6400 0.6200 2.5200
PIN b INV 0.1110 999.0 0.5800 3.6400 0.2100 1.2800

GATE aoi31 2320.00      O = ! ((a1 * a2 * a3) + b);
PIN a1 INV 0.1009 999.0 0.9100 4.0400 0.8100 2.8600
PIN a2 INV 0.1049 999.0 1.0500 3.9300 0.8700 2.8700
PIN a3 INV 0.1059 999.0 1.1500 3.9400 0.9400 2.8600
PIN b INV 0.0979 999.0 0.8900 4.0600 0.2500 1.2800

GATE aoi22 2320.00      O = ! ((a1 * a2) + (b1 * b2));
PIN a1 INV 0.1019 999.0 0.9200 3.4600 0.9400 2.7900
PIN a2 INV 0.0908 999.0 0.8400 3.6400 0.8500 2.7900
PIN b1 INV 0.0958 999.0 0.6100 3.6400 0.4900 2.9300
PIN b2 INV 0.0988 999.0 0.7000 3.6400 0.5400 2.9300

GATE aoi32 2784.00      O = ! ((a1 * a2 * a3) + (b1 * b2));
PIN a1 INV 0.1029 999.0 1.0600 3.8100 0.9600 2.9100
PIN a2 INV 0.1009 999.0 1.2000 3.8100 1.0300 2.9000
PIN a3 INV 0.1060 999.0 1.2900 3.6900 1.0600 2.9100
PIN b1 INV 0.0979 999.0 0.9100 3.8100 0.4300 2.1200
PIN b2 INV 0.1049 999.0 0.7800 3.5900 0.4300 2.1200

GATE aoi33 3248.00      O = ! ((a1 * a2 * a3) + (b1 * b2 * b3));
PIN a1 INV 0.1029 999.0 1.3300 3.9100 1.3000 2.9100
PIN a2 INV 0.1029 999.0 1.4600 3.8400 1.4100 2.9100
PIN a3 INV 0.1120 999.0 1.4700 3.6500 1.4100 2.9100
PIN b1 INV 0.1029 999.0 1.1100 3.5900 0.7600 2.9000
PIN b2 INV 0.0949 999.0 1.0400 3.9100 0.6800 2.9100
PIN b3 INV 0.1039 999.0 0.8400 3.5800 0.6400 2.9000

GATE aoi211 2320.00     O = ! ((a1 * a2) + b + c);
PIN a1 INV 0.1039 999.0 1.1200 4.8100 1.0300 2.3800
PIN a2 INV 0.1090 999.0 1.2900 4.8100 1.0300 2.3800
PIN b INV 0.1080 999.0 1.0400 4.8300 0.5200 1.4000
PIN c INV 0.1008 999.0 0.6800 4.8300 0.5100 1.7900

GATE aoi221 2784.00     O = ! ((a1 * a2) + (b1 * b2) + c);
PIN a1 INV 0.1089 999.0 1.4800 4.4300 1.3300 2.7800
PIN a2 INV 0.0948 999.0 1.4200 4.5600 1.4000 2.7500
PIN b1 INV 0.1029 999.0 0.7600 4.4700 0.7900 2.8900
PIN b2 INV 0.1049 999.0 0.7300 4.5800 0.7800 2.9100
PIN c INV 0.1110 999.0 1.3900 4.5600 0.7000 1.5100

GATE aoi222 3712.00     O = ! ((a1 * a2) + (b1 * b2) + (c1 * c2) );
PIN a1 INV 0.1019 999.0 1.7700 4.5800 1.5600 2.9500
PIN a2 INV 0.0958 999.0 1.7300 4.6900 1.6000 2.9300
```

```
PIN b1 INV 0.1039 999.0 1.3400 4.6800 1.2100 2.9200
PIN b2 INV 0.1039 999.0 1.5000 4.6900 1.2200 2.9200
PIN c1 INV 0.0958 999.0 0.9200 4.6700 0.8100 2.9200
PIN c2 INV 0.1039 999.0 0.7700 4.4700 0.7600 2.9200

GATE oai21 1856.00      O = ! ( (a1 + a2) * b);
PIN a1 INV 0.1019 999.0 0.6900 3.9400 0.5300 2.4700
PIN a2 INV 0.0979 999.0 0.8700 3.9300 0.6300 2.4700
PIN b INV 0.0998 999.0 0.3700 2.0500 0.5700 2.5100

GATE oai31 2320.00      O = ! ( (a1 + a2 + a3) * b);
PIN a1 INV 0.1089 999.0 1.2700 4.7100 1.0300 2.4300
PIN a2 INV 0.1049 999.0 1.1100 4.7100 1.0400 2.5700
PIN a3 INV 0.1090 999.0 0.8500 4.7100 0.6900 2.3800
PIN b INV 0.1059 999.0 0.3800 1.8600 0.8100 2.7300

GATE oai22 2320.00      O = ! ( (a1 + a2) * (b1 + b2));
PIN a1 INV 0.1009 999.0 1.1000 4.0600 0.9000 2.5000
PIN a2 INV 0.1029 999.0 0.9900 4.0600 0.6800 2.3600
PIN b1 INV 0.0958 999.0 0.6900 3.6600 0.7400 2.5300
PIN b2 INV 0.1039 999.0 0.6100 3.6600 0.5600 2.0600

GATE oai32 2784.00      O = ! ( (a1 + a2 + a3) * (b1 + b2));
PIN a1 INV 0.1130 999.0 1.3900 4.4600 1.0400 2.4600
PIN a2 INV 0.1069 999.0 1.2500 4.4600 1.0900 2.6300
PIN a3 INV 0.1140 999.0 0.9900 4.4600 0.7400 2.4200
PIN b1 INV 0.1059 999.0 0.5800 3.2000 0.7900 2.7100
PIN b2 INV 0.1130 999.0 0.6800 3.2100 0.8300 2.3400

GATE oai33 3248.00      O = ! ( (a1 + a2 + a3) * (b1 + b2 + b3));
PIN a1 INV 0.1170 999.0 1.5800 4.3000 1.4800 2.4700
PIN a2 INV 0.1089 999.0 1.5000 4.3100 1.4200 2.6300
PIN a3 INV 0.1079 999.0 1.2400 4.3100 1.1700 2.6500
PIN b1 INV 0.1170 999.0 0.8000 4.3000 0.8200 2.2700
PIN b2 INV 0.1089 999.0 0.0000 4.3000 1.1700 2.6400
PIN b3 INV 0.1109 999.0 1.1300 4.3100 1.3500 2.6500

GATE oai211 2320.00     O = ! ( (a1 + a2) * b * c);
PIN a1 INV 0.1070 999.0 1.1200 4.1700 0.5900 2.3100
PIN a2 INV 0.1131 999.0 1.3000 4.1600 0.7900 2.3600
PIN b INV 0.1050 999.0 0.5100 2.1300 0.6900 2.4000
PIN c INV 0.1050 999.0 0.5000 2.4600 0.5200 2.4100

GATE oai221 2784.00     O = ! ( (a1 + a2) * (b1 + b2) * c);
PIN a1 INV 0.1039 999.0 1.5800 4.1700 1.1100 2.4700
PIN a2 INV 0.1050 999.0 1.4800 4.1700 0.8600 2.3600
PIN b1 INV 0.1080 999.0 0.9400 4.0300 0.8100 2.5000
PIN b2 INV 0.1060 999.0 0.7600 4.0300 0.6400 2.5000
PIN c INV 0.1019 999.0 0.7800 2.2800 0.9000 2.5400
```

```
GATE oai222 3248.00     O = ! ( (a1 + a2) * (b1 + b2) * (c1 + c2));
PIN a1 INV 0.1161 999.0 1.7700 3.7500 1.2100 2.4700
PIN a2 INV 0.1110 999.0 1.6200 3.7500 1.1300 2.4800
PIN b1 INV 0.1009 999.0 1.1700 3.5800 1.0700 2.4800
PIN b2 INV 0.1191 999.0 1.3500 3.5800 1.1000 2.3500
PIN c1 INV 0.1060 999.0 0.9900 3.5900 0.9300 2.4900
PIN c2 INV 0.1140 999.0 0.8200 3.5800 0.7900 2.4800

GATE zero       0       O=CONST0;
GATE one        0       O=CONST1;
```

Chapter 5

Design Example

In this chapter a complete design example is examined to illustrate the use of logic synthesis in the design flow. The methodology starts from a register-transfer level (RTL) specification of the combinational logic and latches in a design. Synthesis tools translate this description into a logic-level representation. The design is simulated at the logic level to guarantee correct operation. Logic optimization is used to improve the quality of the design and to implement the design as a net-list of standard-cell components. Standard-cell place and route is the final step to produce the mask layers for the integrated circuit. The example used is the design of a single-chip implementation of the Data Encryption Standard (DES) for encrypting and decrypting digital data.

5.1 OCT Synthesis Tools

A complete set of tools has been developed at Berkeley for the automatic synthesis of digital designs. The tools are tied together using the OCT database [39] and are available as part of the OCT TOOLS [72] collection of computer-aided design programs.

OCT maintains different representations of a design called *views*. The specific views of interest in the DES design are the *physical view*, the *symbolic view*, and the *logic view*. A physical view represents a design using only mask-level geometries. No connectivity information is present in a physical view. Physical views are used in the DES design for the mask layout of the standard-cells. A symbolic view also contains physical position information and contains mask-level information for the design. However, a symbolic view restricts the geometries which can be used; only wires and instances of other modules are

Tool	Description
BDSYN	Translate BDS into an OCT logic view
BDNET	Translate net-list into an OCT logic view
BDSIM	Switch-level simulation
MIS-II	Logic optimization
WOLFE	OCT interface for TIMBERWOLFSC and YACR
TIMBERWOLFSC	Standard-cell placement and global routing
YACR	Standard-cell detailed routing

Table 5.1: Tools used for the design of DES.

allowed. All connections are through terminals on the modules and electrical connectivity is represented explicitly. A symbolic view is used in the DES design for a module composed of standard cells. A logic view is an abstract form of a symbolic view. No physical position information is available and the components in the design do not necessarily map onto actual cells in any library. Instead, the representation captures the logic functionality of the design using generic components for the logical behavior.

The specific tools used for the DES design are listed in Table 5.1.

Automatic synthesis starts with an RTL model described using a combination of BDSYN and BDNET [69]. BDSYN is a program which translates a subset of Digital Equipment Corporation's behavioral simulation language BDS [4] into a logic-level representation. A model is written for BDSYN with the assumption that the model is translated into a combinational logic block. Logic equations are extracted from the model which preserve the behavior of the original description. The output of the BDSYN translator is a BLIF logic representation. BLIF is a text-file representation of a Boolean network that was developed as an internal format for MIS; in a design scenario, BLIF is converted into an OCT logic view using MIS-II. The combinational portions of the design are composed with latches and tri-states using a textual net-list representation. The program BDNET converts this net-list into an OCT logic view.

An OCT logic view is simulated using the switch-level simulator BDSIM [69]. BDSIM simulates a hierarchical mixture of OCT logic and symbolic views. The elements in the simulation are either transistors or logic gates represented by a logic equation.

MIS-II is a logic optimization program. It is described in more detail in Appendix A. The primary optimization steps of MIS-II Version 2.0 are the algebraic decomposition algo-

rithm described in Chapter 3 and the technology mapping algorithm described in Chapter 4. MIS-II is used on the combinational portions of the design to optimize the logic equations and to map the equations into a cell library. The output of MIS-II is an OCT logic view which only uses cells in the given cell library.

WOLFE is a standard-cell placement and routing program for OCT [61]. It is based on the placement program TIMBERWOLFSC [68] and the channel router YACR [56].

This is a small subset of the programs in the OCT TOOLS set. Many other programs are available to manipulate an OCT design representation. For example, the graphics editor VEM [38] can be used to view and modify the physical and symbolic views of a design. For a list of the programs which work with OCT see [72].

5.2 The Data Encryption Standard (DES)

Digital encryption is the process of transforming a block of binary data (the *plain-text*) into a different representation (the *cipher-text*) under the control of a *key*. Decryption is the inverse process – given the cipher-text and the key, derive the plain-text. The encryption algorithm should be secure. That is, given a plain-text block and a cipher-text block, it should be difficult to derive the key which was used for the encryption. The DES algorithm is one algorithm for digital encryption. It operates on a 64-bit plain-text block using a 56-bit key, and produces a 64-bit cipher-text block.

The DES algorithm was published as a Federal Information Processing Standard in January of 1977 [1]. The goal was to provide a robust algorithm for data encryption which could become a standard for use in digital systems. The standard was intended for use by Federal Departments and Agencies for data encryption in noncritical situations, but it was also encouraged for use by commercial and private organizations where appropriate. The encryption algorithm was developed by IBM, and IBM holds the patents on the algorithm; however, IBM granted royalty-free licenses under these patents to allow wide acceptance of the algorithm.

DES is in wide use for low security applications. Many satellite communications (e.g., cable TV) are encrypted using the DES algorithm. Sun Microsystems is using DES as the basis for their secure ethernet communication product. DES is also used as the security mechanism for passwords in Berkeley Unix [54].

DES has since been withdrawn as a Federal standard due to complaints that it can

be defeated by exhaustive key enumeration. One justification for the small number of keys ($2^{56} < 10^{17}$) was to allow inexpensive implementations of DES. Note that current single-chip DES implementations can perform up to 10^8 encryptions per second (i.e., ten nanosecond encryption time). At this rate, only 8,340 chips are needed in parallel to break the scheme in one day by shear exhaustion. While it remains expensive to build this exhaustive search machine, vulnerability to such a brute-force attack is seen as a weakness in the standard. It is interesting to note that DES has not been broken by any scheme short of exhaustive enumeration of all keys. For example, given a block of plain-text and cipher-text, is it not known how to derive even a single bit of the key. Therefore, a similar algorithm using a larger key (e.g., 128 bits) would avoid the possibility of an exhaustive attack. On the other hand, DES has also not been proven secure in the sense that algorithms based on NP-complete algorithms are.

A public domain C program is available for the DES algorithm.[1] This program performs approximately 200 block encryptions per second on a DEC MicroVax-II which corresponds to an encryption rate of 13,000 bits per second. This speed is practical for low-speed communication channels, but is not suitable for disk-controller interfaces or ethernet interfaces which operate at millions of bits per second. Therefore, special-purpose integrated circuits are used in these applications. The design of such an integrated circuit is the focus of this chapter.

5.3 The DES Algorithm

The following is a brief explanation of the DES algorithm to provide a basis for understanding the RTL descriptions which follow. The DES equations which produce a 64-bit value *out* from a 64-bit value *in* and a 56-bit value *key* are shown in Figure 5.1.

The 64-bit data value *in* is permuted with the initial permutation IP to provide two 32-bit values l_0 and r_0. The basic step of the algorithm is repeated sixteen times. The 32-bit values l_{16} and r_{16} are permuted using the final permutation FP to provide the encrypted data *out*.

The basic step of the algorithm uses a function f of the previous right data block (r_{i-1}) and a permuted subset of the original key (ks_i). An exclusive-or of the function and

[1] James Gillogly is credited for the routines; Phil Karn and Bdale Garbee are given credit for speed improvements and additional features.

$$
\begin{aligned}
\{l_0, r_0\} &= IP(in) \\
\{C_0, D_0\} &= PC1(key) \\
\\
C_1 &= ROTATE(C_0, 1) \\
D_1 &= ROTATE(D_0, 1) \\
ks_1 &= PC2(C_1, D_1) \\
l_1 &= r_0 \\
r_1 &= l_0 \oplus f(r_0, ks_1) \\
&\vdots \\
C_{16} &= ROTATE(C_{15}, 1) \\
D_{16} &= ROTATE(D_{15}, 1) \\
ks_{16} &= PC2(C_{16}, D_{16}) \\
l_{16} &= r_{15} \\
r_{16} &= l_{15} \oplus f(r_{15}, ks_{16}) \\
\\
out &= FP(\{l_{16}, r_{16}\})
\end{aligned}
$$

Figure 5.1: DES Equations.

the previous left data block (l_{i-1})) provides the right data block for the next stage; the left data block for the next stage is the right data block from the previous stage.

The subkeys ks_i are formed from the 28-bit values C_i and D_i. C_0 and D_0 are formed from the initial key using the function $PC1$ (known as *permuted choice 1*). C_i and D_i are rotated one or two positions for each stage depending on the stage number. The 48-bit key ks_i is formed from C_i and D_i using the function $PC2$ (known as *permuted choice 2*).

The heart of DES is the function f. In this function, the 32-bit value r_{i-1} is expanded to 48-bits by duplicating selected bits (the E expansion). The exclusive-or of the 48-bit key ks_i and the expanded value $E(r_{i-1})$ is given to eight lookup tables. Each lookup table takes a block of 6 bits and returns 4 bits. Hence, the lookup tables reduce the 48-bit value to 32-bits. The lookup tables, called the primitive functions, were obtained by random selection from the set of 6-input 4-output Boolean functions. The result of the lookup tables goes through a final permutation FP to produce the output of f.

Decryption is performed by the same hardware by evaluating the equations in reverse order. This is possible because, given the input and output of the exclusive-or at each step, it is possible to determine the unspecified input. Note that the use of the exclusive-or allows for decryption independent of the choice for f.

5.4 DES Implementation Decisions

5.4.1 Design Alternatives

The first design decision for the single-chip DES implementation is the chip architecture. The following alternatives are considered:

1. Fully combinational
2. Word-pipelined
3. Sequential byte-pipelined

Note that in most applications of DES, large blocks of data are encrypted; hence, a natural implementation of a DES chip uses a pipelined architecture.

The *fully combinational* DES design has 120 input signals (56-bit key plus 64-bit plain-text block) and 64 output signals (64-bit cipher-text block). This circuit has sixteen

copies of the basic stage and the computation time is sixteen times the delay of one stage. No circuitry is needed to compute the subkey values ks_i; the value of ks_i at each stage is a static selection of bits from the initial key. No registers are used in this design.

An improvement over the combinational DES design is the *word-pipelined* design. This design separates each stage with a pipeline register. The cycle time for the word-pipelined implementation is the delay of one stage and the area is only slightly increased over the combinational design. A new data word enters the pipeline each cycle and there is a sixteen cycle latency in the pipeline; that is, the encrypted value for a block appears only after sixteen cycles. The word-pipelined design provides much better throughput than the combinational design because the effective encryption time per block is the delay of a single stage; this comes at the expense of a sixteen cycle latency to encrypt a single block and the overhead of sixteen 64-bit pipeline registers. This implementation has the same 120 input signals and 64 output signals as the combinational design.

Another design variation implements a single stage of the DES algorithm plus the control logic to cycle the data through this single stage sixteen times. This design accepts a new 64-bit plain-text block every sixteen cycles and produces the cipher-text block after a sixteen cycle delay. The advantage of this design is that it requires only a single implementation of the basic stage of the DES algorithm. However, there is overhead in computing the subkey (ks_i) for each step in the iteration and for the control logic. Nonetheless, a sequential implementation of DES is expected to be substantially smaller than either the fully combinational or word-pipelined design. The disadvantage is that it takes sixteen cycles to encrypt each data block and only one block enters the chip every sixteen cycles.

A more convenient design is a slight modification of the previous design called the *sequential byte-pipelined* implementation. This design is provided with one byte every other cycle. A 64-bit register on the chip is used to build the next 64-bit data block while the previous 64-bit block is cycled through the basic stage. The cipher-text block is then output from the chip over the next sixteen cycles, with a new byte output every other cycle. This provides a pipeline latency of 32 cycles for a given byte, but leads to a chip interface with only eight data input pins and eight data output pins in addition to a small set of control signals. Once the pipeline is full, the behavior is a DES chip which accepts a byte every other cycle and produces an encrypted data byte every other cycle.

5.4.2 Implementation Technology

The implementation technology for the DES design is two micron CMOS. The entire design is done as a single standard-cell block using the Mississippi State University (MSU) standard-cell library [3]. This is a scalable standard-cell library consisting of thirty combinational logic cells, eight flip-flop and latch cells, twenty-three pad cells, and ten miscellaneous cells. The double-level metal cells (DLM) were used for this design. All of the cells are the same height. Power and ground are routed across the rows over the top of the cells. Each gate input is available at both the top and bottom of the cell using the second metal layer. These cells were designed in Magic [37] and converted into an OCT physical view using CIFTOOCT and OCTMM [51,60].

The MIS-II technology library for the combinational cells is given in Figure 5.2. No data is available for the timing performance of the cells. Dummy delay values of a 1.0 ns fixed delay and a .2 ns/fanout load dependent delay are given in the file to allow MIS-II to read the library.

Some compromises are necessary to describe this technology library for the current implementation of MIS. Cell 1850 is a multiple-output cell (full-adder) and cannot be included. Cells 1660, 1670, 1680, 1760, and 1770 are multiple-output AND and OR gates which provide both the true and complemented output. In order to use these cells with MIS-II only the true output is specified; the complemented functions are available in other gates. MIS-II is unable to use both outputs of these gates which might reduce the number of inverters in the circuit. Cell 1100 is a dual inverter and is not included. Interestingly, the dual inverter is the same size as two independent inverters; the advantage of the dual inverter is that there is more room within the cell area to provide an inverter with improved electrical characteristics. Cells 1520 (4x inverter) and 1320 (noninverting buffer) are present for timing optimization; however, without accurate timing numbers for the cells, they are not included.

The *area* value in the technology file is the horizontal dimension for the cell in λ. This scales easily into microns for two micron CMOS where 1 λ = 1 micron. Each cell has a height of 112 λ. Hence, the cell area in square microns is 112 times the value of cell area.

The only other cell library used for the DES chip is cell 1830 which is a synchronous, rising-edge triggered, D flip-flop. The cell area for 1830 is 88 λ.

5.5 Fully Combinational DES Design

In order to explore the three design alternatives presented in the previous section, the inner loop of the DES algorithm was implemented to determine its size and relative delay. This consists of the function f and the exclusive-or operators for one stage of the algorithm. This design is called ONESTAGE. The size of this design approximates the size of the byte-pipelined design; further, the fully combinational and the word-pipelined implementations should be approximately sixteen times larger than ONESTAGE.

ONESTAGE is described by the BDS program of Figure 5.3. Module one_stage takes a 64-bit data block (data_in) and a 48-bit subkey KSi, and produces a 64-bit data output block (data_out). The eight primitive functions (routines S0 through S7) are contained in file *s.bds* and are shown in Figure 5.4. Each primitive function takes a 6-bit value and returns a 4-bit result using a table lookup as specified by the DES algorithm. The routine main contains an initial exclusive-or against the bits of the key, a static loop to process the correct bits through the primitive functions, and the final result is the exclusive-or of a permutation of f and L. Note that 32-bits of the output block are simply connected to the input block, but the remaining 32 output bits are a complex function of the 112 input bits.

The BDS description is straightforward. However, a few points need to be clarified.

1. Variables are defined using the state statement. All variables are interpreted as temporary wires in a combinational logic network and not as registered data values.
2. state i<> introduces a variable which does not generate logic; these variables are used instead to control the generation of logic (e.g., static FOR loop).
3. The require statement includes the contents of another file.
4. BDS uses the symbol & for the concatenation of bit vectors.

BDSYN is used to translate the BDS description into a multiple-level logic representation in BLIF format. The logic network before optimization has 4,256 literals. The standard optimization script of MIS-II starts with a selective collapse of the network based on the value of the nodes in the network. Kernel and cube extraction are then applied to reduce the literal count of the network. The optimized network has 1,956 literals. Mapping into the MSU cell library provides an implementation using 968 cells with a linear cell dimension of $29,640\lambda$ or a total cell area of 3.3 square millimeters (mm^2). Assuming an overhead factor of two for routing, the placed and routed chip would be 6.6 mm^2 which

translates into 2.6 mm by 2.6 mm. The delay for this implementation is 19.6 gate delays using the dummy delay values for these cells.

From these numbers, the fully combinational DES is estimated to require 106 mm^2 which translates into a chip 10.3 mm on a side. The critical path delay of this design is approximately 320 gate delays.

The word-pipelined design is larger by the area required for the sixteen 64-bit pipeline registers or about 10 mm^2. Hence, for a 10 % area cost, the performance of the design is greatly improved.

The combinational design is an interesting example of a Boolean logic function which is expected to be difficult to represent in two-level sum-of-products form. If it were possible to compute two-level representations of the logic functions $f_i, i = 1 \ldots 64$ for the cipher-text output, then, by intersecting those functions which produce a one in a given cipher block and intersecting the resulting function with the complement of those functions with a 0 in a given cipher block, a set of possible inputs which give rise to this cipher-text block could be derived. By further intersecting this function with a plain-text block, the set of keys which give rise to this (plain-text, cipher-text) pair can be determined. The intersection of two-level sum-of-products expressions is $O(n^2)$ given at most n product terms in the logic functions. Hence, in this manner, the DES algorithm could be compromised from the logic level. However, the DES algorithm appears safe from this attack; collapsing even one stage of the DES algorithm leads to a PLA with more than 40,000 product terms.

5.6 Byte-Pipelined DES Design

The design considered in detail is the sequential byte-pipelined DES design. Because of the low pin count and relatively inexpensive nature of this design, this is the preferred design for a single-chip DES implementation.

The twenty pins for the proposed design are given in Table 5.2.

The chip operates as follows. First, the chip is reset by asserting the `reset` signal. The signal `load_key` is asserted and the key is loaded one byte every other cycle on the `data_in` pins during the next 16 cycles. The `encrypt` signal is then asserted to enable encryption. During the next 32 cycles, two data blocks are sequenced into the chip using the `data_in` pins. At this point, the first encrypted data block appears at the outputs as the third data block is clocked into the chip. The encryption mode can be changed any

Pin	Description
data_in<7:0>	Data input
data_out<7:0>	Data output
encrypt	Encrypt flag (1=encrypt, 0=decrypt)
load_key	Load key flag (1=load key, 0=load data)
reset	Chip reset
clock	Chip clock

Table 5.2: Pin interface for the byte-pipelined DES design.

time, but takes effect only between 64-bit blocks.

The BDS description for this version of the DES chip is shown in Figure 5.5 and the BDNET description for the latches is shown in Figure 5.6. The BDS description describes the combinational portions of the design and a BDNET net-list connects the logic to the latches to complete the description. The BDS descriptions are translated into an OCT logic view using BDSYN and the net-list portion is translated into an OCT logic view by BDNET.

The next step is to simulate this description using BDSIM to verify that it operates as expected. The simulation stimulus performs the following operations. The chip is initialized and the key 0xAAAA_AAAA_AAAA_AAAA is loaded. The encryption flag is set to 1. The data block of 0x5555_5555_5555_5555 is loaded followed by the data block 0x0123_4567_89AB_CDEF. The encryption flag is then set to 0 to start decryption. The encrypted data for 0x5555_5555_5555_5555 (which is 0x533A_D34D_9F90_5C2C) is loaded followed by the encrypted data for 0x0123_4567_89AB_CDEF (which is 0xC78C_CC38_B3EC_2573). Enough zero blocks are then loaded to flush the pipeline. Figure 5.7 shows the output of BDSIM for this sequence of operations. Print commands are inserted at the points where useful data appears on the output signals. It is clear from the simulation output that the chip operates as expected.

The logic network for the combinational portion of this design has 256 inputs and 245 outputs. The initial network has 7,393 literals. After optimization with the standard script in MIS-II, the final network has 3,661 literals. After technology mapping into the MSU cell library, the design has 1,701 gates and a total cell area of 6.0 mm^2. There are 245 edge-triggered D-flip-flops in the design. When these are included, the final cell count is 1,946 and the final cell area is 8.4 mm^2. Note that the gate area for the sequential design is two and a half times larger than ONESTAGE; the extra overhead comes from the registers,

Tool	CPU
BDS, BDNET	71 sec.
MIS	3,398 sec.
WOLFE	12,764 sec.

Table 5.3: Run-time for each step in the DES design.

the key generation logic, and the control logic. The final standard-cell block has a placed and routed area of 4.7 mm by 4.7 mm (22.0 mm^2) without counting the area required for the pad frame. A square aspect ratio was requested when WOLFE was run; the final bounding box was almost exactly square: 4.672 mm by 4.699 mm.

Table 5.3 shows the run-time required for each step in the process on a DEC VAX 8650. The total design time was 4.5 hours.

5.7 Conclusions

In this chapter, a complete design of a nontrivial circuit has been performed automatically using a combination of logic synthesis and place and route tools. The design was written in a high-level language and translated automatically into an optimized mask image in only 4.5 hours of computer time.

5.8 DES Design Description

The following figures present the complete DES design description. This includes the MSU standard-cell library in MIS-II format; the BDS source for one stage of the DES algorithm, the primitive functions, and the complete byte-pipelined implementation; the BDNET net-list for the byte-pipelined implementation; and, the result of a BDSIM simulation for the design.

```
GATE    "1310"          16      O=!1A;
PIN     * INV 1 999 1 .2 1 .2
GATE    "1120"          24      O=!(1A+1B);
PIN     * INV 1 999 1 .2 1 .2
GATE    "1130"          32      O=!(1A+1B+1C);
PIN     * INV 1 999 1 .2 1 .2
GATE    "1140"          40      O=!(1A+1B+1C+1D);
PIN     * INV 1 999 1 .2 1 .2
GATE    "1220"          24      O=!(1A*1B);
PIN     * INV 1 999 1 .2 1 .2
GATE    "1230"          32      O=!(1A*1B*1C);
PIN     * INV 1 999 1 .2 1 .2
GATE    "1240"          40      O=!(1A*1B*1C*1D);
PIN     * INV 1 999 1 .2 1 .2
GATE    "1660"          32      O2=1A*1B;
PIN     * NONINV 1 999 1 .2 1 .2
GATE    "1670"          40      O2=1A*1B*1C;
PIN     * NONINV 1 999 1 .2 1 .2
GATE    "1680"          48      O2=1A*1B*1C*1D;
PIN     * NONINV 1 999 1 .2 1 .2
GATE    "1760"          32      O1=1A+1B;
PIN     * NONINV 1 999 1 .2 1 .2
GATE    "1770"          40      O1=1A+1B+1C;
PIN     * NONINV 1 999 1 .2 1 .2
GATE    "1740"          48      O=1A+1B+1C+1D;
PIN     * NONINV 1 999 1 .2 1 .2
GATE    "1870"          40      O=!(1A*1B+2C*2D);
PIN     * INV 1 999 1 .2 1 .2
GATE    "1880"          32      O=!(1A+2B*2C);
PIN     * INV 1 999 1 .2 1 .2
GATE    "1860"          40      O=!((1A+1B)*(2C+2D));
PIN     * INV 1 999 1 .2 1 .2
GATE    "1890"          32      O=!(1A*(2B+2C));
PIN     * INV 1 999 1 .2 1 .2
GATE    "1970"          56      O=1A*1B+2C*2D;
PIN     * NONINV 1 999 1 .2 1 .2
GATE    "1810"          72      O=1A*1B+2C*2D+3E*3F;
PIN     * NONINV 1 999 1 .2 1 .2
GATE    "1910"          96      O=1A*1B+2C*2D+3E*3F+4G*4H;
PIN     * NONINV 1 999 1 .2 1 .2
GATE    "1930"          64      O=1A*1B*1C+2D*2E*2F;
PIN     * NONINV 1 999 1 .2 1 .2
GATE    "2310"          40      O=1A*!1B+!1A*1B;
PIN     * UNKNOWN 1 999 1 .2 1 .2
GATE    "2350"          48      O=1A*1B+!1A*!1B;
PIN     * UNKNOWN 1 999 1 .2 1 .2
```

Figure 5.2: MIS technology library for the MSU library.

```
GATE    "2310"          40      O=!(1A*1B+!1A*!1B);
PIN     * UNKNOWN 1 999 1 .2 1 .2
GATE    "2350"          48      O=!(1A*!1B+!1A*1B);
PIN     * UNKNOWN 1 999 1 .2 1 .2
GATE    "1610"          32      O=!1A*2B;
PIN     1A INV 1 999 1 .2 1 .2
PIN     2B NONINV 1 999 1 .2 1 .2
GATE    "1620"          32      O=1A+!2B;
PIN     1A NONINV 1 999 1 .2 1 .2
PIN     2B INV 1 999 1 .2 1 .2
GATE    "1350"          48      O=1D1*3SEL+2D2*!3SEL;
PIN     1D1 NONINV 1 999 1 .2 1 .2
PIN     2D2 NONINV 1 999 1 .2 1 .2
PIN     3SEL UNKNOWN 1 999 1 .2 1 .2
GATE    "1430"          8       O=CONST1;
GATE    "1440"          8       O=CONST0;
```

Figure 5.2. MIS technology library for the MSU library. (cont.)

```
MODEL onestage data_out<63:0> = data_in<63:0>, KSi<47:0>;

! include the primitive functions
require 's.bds';


! The 'E' permutation
ROUTINE E<47:0>(k<31:0>);
    RETURN
        k<0> & k<31:27> & k<28:23> & k<24:19> & k<20:15> &
        k<16:11> & k<12:7> & k<8:3> & k<4:0> & k<31>;
ENDROUTINE;


! The 'P' permutation
ROUTINE P<31:0>(k<31:0>);
    RETURN
        k<24> & k<3> & k<10> & k<21> & k<5> & k<29> & k<12> & k<18> &
        k<8> & k<2> & k<26> & k<31> & k<13> & k<23> & k<7> & k<1> &
        k<9> & k<30> & k<17> & k<4> & k<25> & k<22> & k<14> & k<0> &
        k<16> & k<27> & k<11> & k<28> & k<20> & k<19> & k<6> & k<15>;
ENDROUTINE;


ROUTINE main;
    STATE i<>, R<31:0>, L<31:0>;
    STATE f<31:0>, preS<47:0>, scramble<5:0>, tempf<3:0>;

    ! Split the input block into the left and right halves
    L = data_in<31:0>;
    R = data_in<63:32>;

    ! XOR with the KSi to get the 'pre-select' bits
    preS = E(R) XOR KSi;
```

Figure 5.3: BDS description for onestage.

```
    ! Apply the select function to each group of 6 bits, producing a
    ! group of 4 bits.  (Do this for each of the 8 groups).
    FOR i FROM 0 to 7 DO BEGIN

        ! Scramble the bits of preS
        scramble<5> = preS<i*6+0>;
        scramble<4> = preS<i*6+5>;
        scramble<3> = preS<i*6+1>;
        scramble<2> = preS<i*6+2>;
        scramble<1> = preS<i*6+3>;
        scramble<0> = preS<i*6+4>;

        ! Apply the Si function to this 6 bit field
        SELECT i FROM
            [0]: tempf = s0(scramble);
            [1]: tempf = s1(scramble);
            [2]: tempf = s2(scramble);
            [3]: tempf = s3(scramble);
            [4]: tempf = s4(scramble);
            [5]: tempf = s5(scramble);
            [6]: tempf = s6(scramble);
            [7]: tempf = s7(scramble);
        ENDSELECT;

        ! Reverse the bits to get f
        f<i*4+3> = tempf<0>;
        f<i*4+2> = tempf<1>;
        f<i*4+1> = tempf<2>;
        f<i*4+0> = tempf<3>;
    END;

    ! L and R for the next pass
    data_out<63:32> = L XOR P(f);
    data_out<31:0> = R;

ENDROUTINE main;

ENDMODEL onestage;
```

Figure 5.3. BDS description for onestage (cont.).

```
ROUTINE S0<3:0>(val<5:0>);
    SELECT val FROM
    [ 0]: RETURN 14;  [ 1]: RETURN  4;  [ 2]: RETURN 13;  [ 3]: RETURN  1;
    [ 4]: RETURN  2;  [ 5]: RETURN 15;  [ 6]: RETURN 11;  [ 7]: RETURN  8;
    [ 8]: RETURN  3;  [ 9]: RETURN 10;  [10]: RETURN  6;  [11]: RETURN 12;
    [12]: RETURN  5;  [13]: RETURN  9;  [14]: RETURN  0;  [15]: RETURN  7;
    [16]: RETURN  0;  [17]: RETURN 15;  [18]: RETURN  7;  [19]: RETURN  4;
    [20]: RETURN 14;  [21]: RETURN  2;  [22]: RETURN 13;  [23]: RETURN  1;
    [24]: RETURN 10;  [25]: RETURN  6;  [26]: RETURN 12;  [27]: RETURN 11;
    [28]: RETURN  9;  [29]: RETURN  5;  [30]: RETURN  3;  [31]: RETURN  8;
    [32]: RETURN  4;  [33]: RETURN  1;  [34]: RETURN 14;  [35]: RETURN  8;
    [36]: RETURN 13;  [37]: RETURN  6;  [38]: RETURN  2;  [39]: RETURN 11;
    [40]: RETURN 15;  [41]: RETURN 12;  [42]: RETURN  9;  [43]: RETURN  7;
    [44]: RETURN  3;  [45]: RETURN 10;  [46]: RETURN  5;  [47]: RETURN  0;
    [48]: RETURN 15;  [49]: RETURN 12;  [50]: RETURN  8;  [51]: RETURN  2;
    [52]: RETURN  4;  [53]: RETURN  9;  [54]: RETURN  1;  [55]: RETURN  7;
    [56]: RETURN  5;  [57]: RETURN 11;  [58]: RETURN  3;  [59]: RETURN 14;
    [60]: RETURN 10;  [61]: RETURN  0;  [62]: RETURN  6;  [63]: RETURN 13;
    ENDSELECT;
ENDROUTINE;


ROUTINE S1<3:0>(val<5:0>);
    SELECT val FROM
    [ 0]: RETURN 15;  [ 1]: RETURN  1;  [ 2]: RETURN  8;  [ 3]: RETURN 14;
    [ 4]: RETURN  6;  [ 5]: RETURN 11;  [ 6]: RETURN  3;  [ 7]: RETURN  4;
    [ 8]: RETURN  9;  [ 9]: RETURN  7;  [10]: RETURN  2;  [11]: RETURN 13;
    [12]: RETURN 12;  [13]: RETURN  0;  [14]: RETURN  5;  [15]: RETURN 10;
    [16]: RETURN  3;  [17]: RETURN 13;  [18]: RETURN  4;  [19]: RETURN  7;
    [20]: RETURN 15;  [21]: RETURN  2;  [22]: RETURN  8;  [23]: RETURN 14;
    [24]: RETURN 12;  [25]: RETURN  0;  [26]: RETURN  1;  [27]: RETURN 10;
    [28]: RETURN  6;  [29]: RETURN  9;  [30]: RETURN 11;  [31]: RETURN  5;
    [32]: RETURN  0;  [33]: RETURN 14;  [34]: RETURN  7;  [35]: RETURN 11;
    [36]: RETURN 10;  [37]: RETURN  4;  [38]: RETURN 13;  [39]: RETURN  1;
    [40]: RETURN  5;  [41]: RETURN  8;  [42]: RETURN 12;  [43]: RETURN  6;
    [44]: RETURN  9;  [45]: RETURN  3;  [46]: RETURN  2;  [47]: RETURN 15;
    [48]: RETURN 13;  [49]: RETURN  8;  [50]: RETURN 10;  [51]: RETURN  1;
    [52]: RETURN  3;  [53]: RETURN 15;  [54]: RETURN  4;  [55]: RETURN  2;
    [56]: RETURN 11;  [57]: RETURN  6;  [58]: RETURN  7;  [59]: RETURN 12;
    [60]: RETURN  0;  [61]: RETURN  5;  [62]: RETURN 14;  [63]: RETURN  9;
    ENDSELECT;
ENDROUTINE;
```

Figure 5.4: BDS description for the primitive functions.

```
ROUTINE S2<3:0>(val<5:0>);
    SELECT val FROM
    [ 0]: RETURN 10;  [ 1]: RETURN  0;  [ 2]: RETURN  9;  [ 3]: RETURN 14;
    [ 4]: RETURN  6;  [ 5]: RETURN  3;  [ 6]: RETURN 15;  [ 7]: RETURN  5;
    [ 8]: RETURN  1;  [ 9]: RETURN 13;  [10]: RETURN 12;  [11]: RETURN  7;
    [12]: RETURN 11;  [13]: RETURN  4;  [14]: RETURN  2;  [15]: RETURN  8;
    [16]: RETURN 13;  [17]: RETURN  7;  [18]: RETURN  0;  [19]: RETURN  9;
    [20]: RETURN  3;  [21]: RETURN  4;  [22]: RETURN  6;  [23]: RETURN 10;
    [24]: RETURN  2;  [25]: RETURN  8;  [26]: RETURN  5;  [27]: RETURN 14;
    [28]: RETURN 12;  [29]: RETURN 11;  [30]: RETURN 15;  [31]: RETURN  1;
    [32]: RETURN 13;  [33]: RETURN  6;  [34]: RETURN  4;  [35]: RETURN  9;
    [36]: RETURN  8;  [37]: RETURN 15;  [38]: RETURN  3;  [39]: RETURN  0;
    [40]: RETURN 11;  [41]: RETURN  1;  [42]: RETURN  2;  [43]: RETURN 12;
    [44]: RETURN  5;  [45]: RETURN 10;  [46]: RETURN 14;  [47]: RETURN  7;
    [48]: RETURN  1;  [49]: RETURN 10;  [50]: RETURN 13;  [51]: RETURN  0;
    [52]: RETURN  6;  [53]: RETURN  9;  [54]: RETURN  8;  [55]: RETURN  7;
    [56]: RETURN  4;  [57]: RETURN 15;  [58]: RETURN 14;  [59]: RETURN  3;
    [60]: RETURN 11;  [61]: RETURN  5;  [62]: RETURN  2;  [63]: RETURN 12;
    ENDSELECT;
ENDROUTINE;


ROUTINE S3<3:0>(val<5:0>);
    SELECT val FROM
    [ 0]: RETURN  7;  [ 1]: RETURN 13;  [ 2]: RETURN 14;  [ 3]: RETURN  3;
    [ 4]: RETURN  0;  [ 5]: RETURN  6;  [ 6]: RETURN  9;  [ 7]: RETURN 10;
    [ 8]: RETURN  1;  [ 9]: RETURN  2;  [10]: RETURN  8;  [11]: RETURN  5;
    [12]: RETURN 11;  [13]: RETURN 12;  [14]: RETURN  4;  [15]: RETURN 15;
    [16]: RETURN 13;  [17]: RETURN  8;  [18]: RETURN 11;  [19]: RETURN  5;
    [20]: RETURN  6;  [21]: RETURN 15;  [22]: RETURN  0;  [23]: RETURN  3;
    [24]: RETURN  4;  [25]: RETURN  7;  [26]: RETURN  2;  [27]: RETURN 12;
    [28]: RETURN  1;  [29]: RETURN 10;  [30]: RETURN 14;  [31]: RETURN  9;
    [32]: RETURN 10;  [33]: RETURN  6;  [34]: RETURN  9;  [35]: RETURN  0;
    [36]: RETURN 12;  [37]: RETURN 11;  [38]: RETURN  7;  [39]: RETURN 13;
    [40]: RETURN 15;  [41]: RETURN  1;  [42]: RETURN  3;  [43]: RETURN 14;
    [44]: RETURN  5;  [45]: RETURN  2;  [46]: RETURN  8;  [47]: RETURN  4;
    [48]: RETURN  3;  [49]: RETURN 15;  [50]: RETURN  0;  [51]: RETURN  6;
    [52]: RETURN 10;  [53]: RETURN  1;  [54]: RETURN 13;  [55]: RETURN  8;
    [56]: RETURN  9;  [57]: RETURN  4;  [58]: RETURN  5;  [59]: RETURN 11;
    [60]: RETURN 12;  [61]: RETURN  7;  [62]: RETURN  2;  [63]: RETURN 14;
    ENDSELECT;
ENDROUTINE;
```

Figure 5.4. BDS description for the primitive functions (cont.).

```
ROUTINE S4<3:0>(val<5:0>);
    SELECT val FROM
    [ 0]: RETURN  2;  [ 1]: RETURN 12;  [ 2]: RETURN  4;  [ 3]: RETURN  1;
    [ 4]: RETURN  7;  [ 5]: RETURN 10;  [ 6]: RETURN 11;  [ 7]: RETURN  6;
    [ 8]: RETURN  8;  [ 9]: RETURN  5;  [10]: RETURN  3;  [11]: RETURN 15;
    [12]: RETURN 13;  [13]: RETURN  0;  [14]: RETURN 14;  [15]: RETURN  9;
    [16]: RETURN 14;  [17]: RETURN 11;  [18]: RETURN  2;  [19]: RETURN 12;
    [20]: RETURN  4;  [21]: RETURN  7;  [22]: RETURN 13;  [23]: RETURN  1;
    [24]: RETURN  5;  [25]: RETURN  0;  [26]: RETURN 15;  [27]: RETURN 10;
    [28]: RETURN  3;  [29]: RETURN  9;  [30]: RETURN  8;  [31]: RETURN  6;
    [32]: RETURN  4;  [33]: RETURN  2;  [34]: RETURN  1;  [35]: RETURN 11;
    [36]: RETURN 10;  [37]: RETURN 13;  [38]: RETURN  7;  [39]: RETURN  8;
    [40]: RETURN 15;  [41]: RETURN  9;  [42]: RETURN 12;  [43]: RETURN  5;
    [44]: RETURN  6;  [45]: RETURN  3;  [46]: RETURN  0;  [47]: RETURN 14;
    [48]: RETURN 11;  [49]: RETURN  8;  [50]: RETURN 12;  [51]: RETURN  7;
    [52]: RETURN  1;  [53]: RETURN 14;  [54]: RETURN  2;  [55]: RETURN 13;
    [56]: RETURN  6;  [57]: RETURN 15;  [58]: RETURN  0;  [59]: RETURN  9;
    [60]: RETURN 10;  [61]: RETURN  4;  [62]: RETURN  5;  [63]: RETURN  3;
    ENDSELECT;
ENDROUTINE;

ROUTINE S5<3:0>(val<5:0>);
    SELECT val FROM
    [ 0]: RETURN 12;  [ 1]: RETURN  1;  [ 2]: RETURN 10;  [ 3]: RETURN 15;
    [ 4]: RETURN  9;  [ 5]: RETURN  2;  [ 6]: RETURN  6;  [ 7]: RETURN  8;
    [ 8]: RETURN  0;  [ 9]: RETURN 13;  [10]: RETURN  3;  [11]: RETURN  4;
    [12]: RETURN 14;  [13]: RETURN  7;  [14]: RETURN  5;  [15]: RETURN 11;
    [16]: RETURN 10;  [17]: RETURN 15;  [18]: RETURN  4;  [19]: RETURN  2;
    [20]: RETURN  7;  [21]: RETURN 12;  [22]: RETURN  9;  [23]: RETURN  5;
    [24]: RETURN  6;  [25]: RETURN  1;  [26]: RETURN 13;  [27]: RETURN 14;
    [28]: RETURN  0;  [29]: RETURN 11;  [30]: RETURN  3;  [31]: RETURN  8;
    [32]: RETURN  9;  [33]: RETURN 14;  [34]: RETURN 15;  [35]: RETURN  5;
    [36]: RETURN  2;  [37]: RETURN  8;  [38]: RETURN 12;  [39]: RETURN  3;
    [40]: RETURN  7;  [41]: RETURN  0;  [42]: RETURN  4;  [43]: RETURN 10;
    [44]: RETURN  1;  [45]: RETURN 13;  [46]: RETURN 11;  [47]: RETURN  6;
    [48]: RETURN  4;  [49]: RETURN  3;  [50]: RETURN  2;  [51]: RETURN 12;
    [52]: RETURN  9;  [53]: RETURN  5;  [54]: RETURN 15;  [55]: RETURN 10;
    [56]: RETURN 11;  [57]: RETURN 14;  [58]: RETURN  1;  [59]: RETURN  7;
    [60]: RETURN  6;  [61]: RETURN  0;  [62]: RETURN  8;  [63]: RETURN 13;
    ENDSELECT;
ENDROUTINE;
```

Figure 5.4. BDS description for the primitive functions (cont.).

```
ROUTINE S6<3:0>(val<5:0>);
    SELECT val FROM
    [ 0]: RETURN  4;  [ 1]: RETURN 11;  [ 2]: RETURN  2;  [ 3]: RETURN 14;
    [ 4]: RETURN 15;  [ 5]: RETURN  0;  [ 6]: RETURN  8;  [ 7]: RETURN 13;
    [ 8]: RETURN  3;  [ 9]: RETURN 12;  [10]: RETURN  9;  [11]: RETURN  7;
    [12]: RETURN  5;  [13]: RETURN 10;  [14]: RETURN  6;  [15]: RETURN  1;
    [16]: RETURN 13;  [17]: RETURN  0;  [18]: RETURN 11;  [19]: RETURN  7;
    [20]: RETURN  4;  [21]: RETURN  9;  [22]: RETURN  1;  [23]: RETURN 10;
    [24]: RETURN 14;  [25]: RETURN  3;  [26]: RETURN  5;  [27]: RETURN 12;
    [28]: RETURN  2;  [29]: RETURN 15;  [30]: RETURN  8;  [31]: RETURN  6;
    [32]: RETURN  1;  [33]: RETURN  4;  [34]: RETURN 11;  [35]: RETURN 13;
    [36]: RETURN 12;  [37]: RETURN  3;  [38]: RETURN  7;  [39]: RETURN 14;
    [40]: RETURN 10;  [41]: RETURN 15;  [42]: RETURN  6;  [43]: RETURN  8;
    [44]: RETURN  0;  [45]: RETURN  5;  [46]: RETURN  9;  [47]: RETURN  2;
    [48]: RETURN  6;  [49]: RETURN 11;  [50]: RETURN 13;  [51]: RETURN  8;
    [52]: RETURN  1;  [53]: RETURN  4;  [54]: RETURN 10;  [55]: RETURN  7;
    [56]: RETURN  9;  [57]: RETURN  5;  [58]: RETURN  0;  [59]: RETURN 15;
    [60]: RETURN 14;  [61]: RETURN  2;  [62]: RETURN  3;  [63]: RETURN 12;
    ENDSELECT;
ENDROUTINE;


ROUTINE S7<3:0>(val<5:0>);
    SELECT val FROM
    [ 0]: RETURN 13;  [ 1]: RETURN  2;  [ 2]: RETURN  8;  [ 3]: RETURN  4;
    [ 4]: RETURN  6;  [ 5]: RETURN 15;  [ 6]: RETURN 11;  [ 7]: RETURN  1;
    [ 8]: RETURN 10;  [ 9]: RETURN  9;  [10]: RETURN  3;  [11]: RETURN 14;
    [12]: RETURN  5;  [13]: RETURN  0;  [14]: RETURN 12;  [15]: RETURN  7;
    [16]: RETURN  1;  [17]: RETURN 15;  [18]: RETURN 13;  [19]: RETURN  8;
    [20]: RETURN 10;  [21]: RETURN  3;  [22]: RETURN  7;  [23]: RETURN  4;
    [24]: RETURN 12;  [25]: RETURN  5;  [26]: RETURN  6;  [27]: RETURN 11;
    [28]: RETURN  0;  [29]: RETURN 14;  [30]: RETURN  9;  [31]: RETURN  2;
    [32]: RETURN  7;  [33]: RETURN 11;  [34]: RETURN  4;  [35]: RETURN  1;
    [36]: RETURN  9;  [37]: RETURN 12;  [38]: RETURN 14;  [39]: RETURN  2;
    [40]: RETURN  0;  [41]: RETURN  6;  [42]: RETURN 10;  [43]: RETURN 13;
    [44]: RETURN 15;  [45]: RETURN  3;  [46]: RETURN  5;  [47]: RETURN  8;
    [48]: RETURN  2;  [49]: RETURN  1;  [50]: RETURN 14;  [51]: RETURN  7;
    [52]: RETURN  4;  [53]: RETURN 10;  [54]: RETURN  8;  [55]: RETURN 13;
    [56]: RETURN 15;  [57]: RETURN 12;  [58]: RETURN  9;  [59]: RETURN  0;
    [60]: RETURN  3;  [61]: RETURN  5;  [62]: RETURN  6;  [63]: RETURN 11;
    ENDSELECT;
ENDROUTINE;
```

Figure 5.4. BDS description for the primitive functions (cont.).

```
MODEL des
    inreg_new<55:0>,
    outreg_new<63:0>,
    data_new<63:0>,
    count_new<3:0>,
    C_new<27:0>,
    D_new<27:0>,
    encrypt_mode_new<0>
=
    data_in<7:0>,        ! data input pins
    reset<0>,            ! reset pin
    encrypt<0>,          ! encryption mode pin
    load_key<0>,         ! load key enable pin
    inreg<55:0>,         ! the input register
    outreg<63:0>,        ! the output register
    data<63:0>,          ! the data register
    count<3:0>,          ! current loop index
    C<27:0>,             ! current value for C (key, 1st half)
    D<27:0>,             ! current value for D (key, 2nd half)
    encrypt_mode<0>;     ! encryption mode

require 's.bds';                          ! the primitive functions

ROUTINE IP<63:0>(k<63:0>);                ! The 'initial' permutation
    RETURN
        k<6> & k<14> & k<22> & k<30> & k<38> & k<46> & k<54> & k<62> &
        k<4> & k<12> & k<20> & k<28> & k<36> & k<44> & k<52> & k<60> &
        k<2> & k<10> & k<18> & k<26> & k<34> & k<42> & k<50> & k<58> &
        k<0> & k<8> & k<16> & k<24> & k<32> & k<40> & k<48> & k<56> &
        k<7> & k<15> & k<23> & k<31> & k<39> & k<47> & k<55> & k<63> &
        k<5> & k<13> & k<21> & k<29> & k<37> & k<45> & k<53> & k<61> &
        k<3> & k<11> & k<19> & k<27> & k<35> & k<43> & k<51> & k<59> &
        k<1> & k<9> & k<17> & k<25> & k<33> & k<41> & k<49> & k<57>;
ENDROUTINE;

ROUTINE FP<63:0>(k<63:0>);                ! The 'final' permutation (IP^-1)
    RETURN
        k<24> & k<56> & k<16> & k<48> & k<8> & k<40> & k<0> & k<32> &
        k<25> & k<57> & k<17> & k<49> & k<9> & k<41> & k<1> & k<33> &
        k<26> & k<58> & k<18> & k<50> & k<10> & k<42> & k<2> & k<34> &
        k<27> & k<59> & k<19> & k<51> & k<11> & k<43> & k<3> & k<35> &
        k<28> & k<60> & k<20> & k<52> & k<12> & k<44> & k<4> & k<36> &
        k<29> & k<61> & k<21> & k<53> & k<13> & k<45> & k<5> & k<37> &
        k<30> & k<62> & k<22> & k<54> & k<14> & k<46> & k<6> & k<38> &
        k<31> & k<63> & k<23> & k<55> & k<15> & k<47> & k<7> & k<39>;
ENDROUTINE;
```

Figure 5.5: BDS description for the byte-pipelined implementation.

```
ROUTINE E<47:0>(k<31:0>);                ! The 'E' permutation
    RETURN
        k<0> & k<31:27> & k<28:23> & k<24:19> & k<20:15> &
        k<16:11> & k<12:7> & k<8:3> & k<4:0> & k<31>;
ENDROUTINE;

ROUTINE P<31:0>(k<31:0>);                ! The 'P' permutation
    RETURN
        k<24> & k<3> & k<10> & k<21> & k<5> & k<29> & k<12> & k<18> &
        k<8> & k<2> & k<26> & k<31> & k<13> & k<23> & k<7> & k<1> &
        k<9> & k<30> & k<17> & k<4> & k<25> & k<22> & k<14> & k<0> &
        k<16> & k<27> & k<11> & k<28> & k<20> & k<19> & k<6> & k<15>;
ENDROUTINE;

ROUTINE pc1_c<27:0>(k<63:0>);            ! permuted choice 1
    RETURN
        k<35> & k<43> & k<51> & k<59> & k<2> & k<10> & k<18> &
        k<26> & k<34> & k<42> & k<50> & k<58> & k<1> & k<9> &
        k<17> & k<25> & k<33> & k<41> & k<49> & k<57> & k<0> &
        k<8> & k<16> & k<24> & k<32> & k<40> & k<48> & k<56>;
ENDROUTINE;

ROUTINE pc1_d<27:0>(k<63:0>);
    RETURN
        k<3> & k<11> & k<19> & k<27> & k<4> & k<12> & k<20> &
        k<28> & k<36> & k<44> & k<52> & k<60> & k<5> & k<13> &
        k<21> & k<29> & k<37> & k<45> & k<53> & k<61> & k<6> &
        k<14> & k<22> & k<30> & k<38> & k<46> & k<54> & k<62>;
ENDROUTINE;

ROUTINE pc2_c<23:0>(k<27:0>);            ! permuted choice 2
    RETURN
        k<1> & k<12> & k<19> & k<26> & k<6> & k<15> &
        k<7> & k<25> & k<3> & k<11> & k<18> & k<22> &
        k<9> & k<20> & k<5> & k<14> & k<27> & k<2> &
        k<4> & k<0> & k<23> & k<10> & k<16> & k<13>;
ENDROUTINE;

ROUTINE pc2_d<23:0>(k<27:0>);
    RETURN
        k<3> & k<0> & k<7> & k<21> & k<13> & k<17> &
        k<24> & k<5> & k<27> & k<10> & k<20> & k<15> &
        k<19> & k<4> & k<16> & k<22> & k<11> & k<1> &
        k<26> & k<18> & k<8> & k<2> & k<23> & k<12>;
ENDROUTINE;
```

Figure 5.5. BDS description for the byte-pipelined implementation (cont.).

```
! key schedule generator (and main control)
ROUTINE generate_key;
    STATE shift_by_one, freeze;

    IF reset THEN BEGIN
        count_new = 0;
        C_new = 0;
        D_new = 0;

    END ELSE IF load_key AND (count EQL 15) THEN BEGIN
        IF encrypt THEN BEGIN
            count_new = 0;
            C_new = pc1_c(inreg & data_in) SRR 1;
            D_new = pc1_d(inreg & data_in) SRR 1;
        END ELSE BEGIN
            count_new = 0;
            C_new = pc1_c(inreg & data_in);
            D_new = pc1_d(inreg & data_in);
        END;

    END ELSE BEGIN
        shift_by_one = count EQL 0 OR count EQL 7 OR
                            count EQL 14 OR count EQL 15;
        freeze = (count EQL 15) AND (encrypt NEQ encrypt_mode);
        IF freeze THEN BEGIN
            C_new = C;
            D_new = D;
        END ELSE IF shift_by_one THEN BEGIN
            IF encrypt_mode THEN BEGIN
                C_new = C SRR 1;
                D_new = D SRR 1;
            END ELSE BEGIN
                C_new = C SLR 1;
                D_new = D SLR 1;
            END;
        END ELSE BEGIN
            IF encrypt_mode THEN BEGIN
                C_new = C SRR 2;
                D_new = D SRR 2;
            END ELSE BEGIN
                C_new = C SLR 2;
                D_new = D SLR 2;
            END;
        END;
        count_new = count + 1;
    END;
```

Figure 5.5. BDS description for the byte-pipelined implementation (cont.).

```
    IF count EQL 15 THEN
        encrypt_mode_new = encrypt
    ELSE
        encrypt_mode_new = encrypt_mode;
ENDROUTINE;


ROUTINE main;
    STATE i<>, R<31:0>, L<31:0>, KSi<47:0>, stage_out<63:0>;
    STATE f<31:0>, preS<47:0>, scramble<5:0>, tempf<3:0>;

    ! Generate the key for this iteration
    KSi = pc2_d(D) & pc2_c(C);

    ! Split the input block into the left and right halves
    L = data<31:0>;
    R = data<63:32>;

    ! XOR with the key to get the 'pre-select' bits
    preS = E(R) XOR KSi;

    ! Apply the select function to each group of 6 bits, producing a
    ! group of 4 bits.  (Do this for each of the 8 groups).
    FOR i FROM 0 to 7 DO BEGIN

        ! Scramble the bits of preS
        scramble<5> = preS<i*6+0>;
        scramble<4> = preS<i*6+5>;
        scramble<3> = preS<i*6+1>;
        scramble<2> = preS<i*6+2>;
        scramble<1> = preS<i*6+3>;
        scramble<0> = preS<i*6+4>;

        ! Apply the Si function to this 6 bit field
        SELECT i FROM
            [0]: tempf = s0(scramble);
            [1]: tempf = s1(scramble);
            [2]: tempf = s2(scramble);
            [3]: tempf = s3(scramble);
            [4]: tempf = s4(scramble);
            [5]: tempf = s5(scramble);
            [6]: tempf = s6(scramble);
            [7]: tempf = s7(scramble);
        ENDSELECT;
```

Figure 5.5. BDS description for the byte-pipelined implementation (cont.).

```
        ! Scramble the result into f
        f<i*4+3> = tempf<0>;
        f<i*4+2> = tempf<1>;
        f<i*4+1> = tempf<2>;
        f<i*4+0> = tempf<3>;
    END;

    ! L and R for the next pass
    stage_out<63:32> = L XOR P(f);
    stage_out<31:0> = R;

    IF count EQL 15 THEN BEGIN
        data_new = IP(inreg & data_in);
        outreg_new = FP(stage_out<31:0> & stage_out<63:32>);

    END ELSE BEGIN
        data_new = stage_out;

        ! on odd-numbered cycles, shift the input register
        IF ((count AND 1) EQL 1) THEN BEGIN
            inreg_new = inreg<47:0> & data_in;
        END ELSE BEGIN
            inreg_new = inreg;
        END;

        ! on odd-numbered cycles, shift the output register
        IF ((count AND 1) EQL 1) THEN BEGIN
            outreg_new<55:0> = outreg<63:8>;
        END ELSE BEGIN
            outreg_new = outreg;
        END;
    END;

ENDROUTINE main;
ENDMODEL des;
```

Figure 5.5. BDS description for the byte-pipelined implementation (cont.).

```
MODEL des:symbolic;
    TECHNOLOGY scmos;
    VIEWTYPE SYMBOLIC;
    EDITSTYLE SYMBOLIC;

    INPUT data_in<7:0> reset<0> load_key<0> encrypt<0> ;
    OUTPUT outreg<7:0>;
    CLOCK masterClock;
    SUPPLY Vdd;
    GROUND GND;

    INSTANCE des:logic
        data_in<7:0> : data_in<7:0>;
        reset<0> : reset<0>;
        encrypt<0> : encrypt<0>;
        load_key<0> : load_key<0>;
        count<3:0> : count<3:0>;
        count_new<3:0> : count_new<3:0>;
        C<27:0> : C<27:0>;
        C_new<27:0> : C_new<27:0>;
        D<27:0> : D<27:0>;
        D_new<27:0> : D_new<27:0>;
        data<63:0> : data<63:0>;
        data_new<63:0> : data_new<63:0>;
        inreg<55:0> : inreg<55:0>;
        inreg_new<55:0> : inreg_new<55:0>;
        outreg<63:0> : outreg<63:0>;
        outreg_new<63:0> : outreg_new<63:0>;
        encrypt_mode<0> : encrypt_mode<0>;
        encrypt_mode_new<0> : encrypt_mode_new<0>;

    ARRAY %i FROM 0 TO 55 OF
        INSTANCE 1830:physical
            "1DATA" : inreg_new<%i>;
            Q : inreg<%i>;
            QB : UNCONNECTED;
            "Vdd!" : Vdd;
            "GND!" : GND;
            "2CLK" : masterClock;

    ARRAY %i FROM 0 TO 63 OF
        INSTANCE 1830:physical
            "1DATA" : outreg_new<%i>;
            Q : outreg<%i>;
            QB : UNCONNECTED;
            "Vdd!" : Vdd;
```

Figure 5.6: BDNET description for the byte-pipelined implementation.

```
            "GND!" : GND;
            "2CLK" : masterClock;

    ARRAY %i FROM 0 TO 63 OF
        INSTANCE 1830:physical
            "1DATA" : data_new<%i>;
            Q : data<%i>;
            QB : UNCONNECTED;
            "Vdd!" : Vdd;
            "GND!" : GND;
            "2CLK" : masterClock;

    ARRAY %i FROM 0 TO 3 OF
        INSTANCE 1830:physical
            "1DATA" : count_new<%i>;
            Q : count<%i>;
            QB : UNCONNECTED;
            "Vdd!" : Vdd;
            "GND!" : GND;
            "2CLK" : masterClock;

    ARRAY %i FROM 0 TO 27 OF
        INSTANCE 1830:physical
            "1DATA" : C_new<%i>;
            Q : C<%i>;
            QB : UNCONNECTED;
            "Vdd!" : Vdd;
            "GND!" : GND;
            "2CLK" : masterClock;

    ARRAY %i FROM 0 TO 27 OF
        INSTANCE 1830:physical
            "1DATA" : D_new<%i>;
            Q : D<%i>;
            QB : UNCONNECTED;
            "Vdd!" : Vdd;
            "GND!" : GND;
            "2CLK" : masterClock;

    INSTANCE 1830:physical
        "1DATA" : encrypt_mode_new<0>;
        Q : encrypt_mode<0>;
        QB : UNCONNECTED;
        "Vdd!" : Vdd;
        "GND!" : GND;
        "2CLK" : masterClock;
ENDMODEL;
```

Figure 5.6. BDNET description for the byte-pipelined implementation (cont).

```
bdsim> Source list mode: M   More lines: 63   Flags:    Depth 1
bdsim> macro cycle
>     set masterClock 1
>     ev
>     set masterClock 0
>     ev
> $end
bdsim>
bdsim> macro cycle2
>     cycle
>     cycle
> $end
bdsim>
bdsim> macro cycle2_print
>     sh load_key encrypt reset count data_out
>     cycle
>     cycle
> $end
bdsim>
bdsim> macro cycle4
>     cycle2
>     cycle2
> $end
bdsim>
bdsim> macro cycle16
>     cycle4
>     cycle4
>     cycle4
>     cycle4
> $end
bdsim>
bdsim>
bdsim> makevector data_in data_in<7:0>
bdsim> makevector reset reset<0>
bdsim> makevector load_key load_key<0>
bdsim> makevector encrypt encrypt<0>
bdsim> makevector encrypt_mode encrypt_mode<0>
bdsim> makevector data_out outreg<7:0>
bdsim>
bdsim> makevector data data<63:0>
bdsim> makevector data_new data_new<63:0>
bdsim> makevector inreg inreg<55:0>
bdsim> makevector inreg_new inreg_new<55:0>
bdsim> makevector outreg outreg<63:0>
bdsim> makevector outreg_new outreg_new<63:0>
bdsim> makevector count count<3:0>
```

Figure 5.7: BDSIM output for the byte-pipelined implementation.

```
bdsim> makevector count_new count_new<3:0>
bdsim> makevector C C<27:0>
bdsim> makevector C_new C_new<27:0>
bdsim> makevector D D<27:0>
bdsim> makevector D_new D_new<27:0>
bdsim>
bdsim> set reset 1
bdsim> cycle
bdsim>
bdsim> set reset 0
bdsim> set encrypt 1
bdsim> set load_key 1
bdsim> set data_in HAA
bdsim> cycle16
bdsim>
bdsim> set load_key 0
bdsim> set data_in H55
bdsim> cycle2
bdsim> set data_in H55
bdsim> cycle2
bdsim> set data_in H55
bdsim> cycle2
bdsim> set data_in H55
bdsim> cycle2
bdsim> set data_in H55
bdsim> cycle2
bdsim> set data_in H55
bdsim> cycle2
bdsim> set data_in H55
bdsim> cycle2
bdsim> set data_in H55
bdsim> cycle2
bdsim> set data_in H55
bdsim> cycle2
bdsim>
bdsim> set data_in H01
bdsim> cycle2
bdsim> set data_in H23
bdsim> cycle2
bdsim> set data_in H45
bdsim> cycle2
bdsim> set data_in H67
bdsim> cycle2
bdsim> set data_in H89
bdsim> cycle2
bdsim> set data_in HAB
bdsim> cycle2
bdsim> set data_in HCD
```

Figure 5.7. BDSIM output for the byte-pipelined implementation (cont.).

```
bdsim> cycle2
bdsim> set data_in HEF
bdsim> cycle2
bdsim>
bdsim> set encrypt 0
bdsim> set data_in H53
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H0 data_out:H2C
bdsim> set data_in H3A
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H2 data_out:H5C
bdsim> set data_in HD3
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H4 data_out:H90
bdsim> set data_in H4D
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H6 data_out:H9F
bdsim> set data_in H9F
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H8 data_out:H4D
bdsim> set data_in H90
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:HA data_out:HD3
bdsim> set data_in H5C
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:HC data_out:H3A
bdsim> set data_in H2C
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:HE data_out:H53
bdsim>
bdsim> set data_in HC7
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H0 data_out:H73
bdsim> set data_in H8C
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H2 data_out:H25
bdsim> set data_in HCC
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H4 data_out:HEC
bdsim> set data_in H38
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H6 data_out:HB3
bdsim> set data_in HB3
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H8 data_out:H38
bdsim> set data_in HEC
```

Figure 5.7. BDSIM output for the byte-pipelined implementation (cont.).

```
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:HA data_out:HCC
bdsim> set data_in H25
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:HC data_out:H8C
bdsim> set data_in H73
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:HE data_out:HC7
bdsim>
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H0 data_out:H55
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H2 data_out:H55
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H4 data_out:H55
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H6 data_out:H55
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H8 data_out:H55
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:HA data_out:H55
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:HC data_out:H55
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:HE data_out:H55
bdsim>
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H0 data_out:HEF
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H2 data_out:HCD
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H4 data_out:HAB
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H6 data_out:H89
```

Figure 5.7. BDSIM output for the byte-pipelined implementation (cont.).

```
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:H8 data_out:H67
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:HA data_out:H45
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:HC data_out:H23
bdsim> set data_in H00
bdsim> cycle2_print
 load_key:0 encrypt:0 reset:0 count:HE data_out:H01
```

Figure 5.7. BDSIM output for the byte-pipelined implementation (cont.).

Chapter 6

Conclusions

Automatic logic synthesis must produce results competitive with those of manual designers in order to have an impact in the automation of VLSI logic design. Therefore, the primary focus of this thesis has been on optimization algorithms to produce high quality circuits. A secondary focus has been to address the issue of complexity so that the optimization techniques may be applied to large circuits.

A complete system for logic synthesis from a register-transfer level down to an optimal logic implementation has been described. The primary optimization techniques include two-level minimization of logic equations, decomposition of logic equations into an optimal multiple-level form, and the mapping of the optimized logic equations onto the restricted set of cells available in a given library.

The program MIS has been developed which implements each of the optimization techniques described in this thesis. The viability of the techniques described here has been demonstrated with experimental results provided by the MIS program.

MIS is currently in production use at a number of places including Advanced Micro Devices, AT&T, Digital Equipment Corporation, Intel Corporation, and SGS Thomson. Feedback from AT&T, Advanced Micro Devices, and Intel Corporation indicates that MIS performs well at logic design for random logic as compared to human designers. For these reasons, it is argued that the techniques described here comprise an acceptable tool for automating the logic design process.

6.1 Future Work

In this section, I briefly describe some of the areas of combinational logic optimization which I feel require more attention.

One problem which remains unsolved is the formulation of an exact algorithm for algebraic decomposition. Rectangle-covering provides a nice abstraction for the decomposition problem, but it is difficult to extend the this technique into an exact algorithm for algebraic decomposition. The problem is that, in algebraic decomposition, the selection of a particular rectangle immediately affects both the set of available rectangles and the cost of all of the rectangles. The optimum rectangle cover, using a static cost function, does not provide the optimum algebraic decomposition.

A related problem involves the limitations of algebraic decomposition. It is known that for many circuits with a regular decomposition, such as a binary adder or a binary-to-unary converter, a human designer can derive a structure which is superior to the current algebraic algorithms. However, it is not known whether this is an inherent limitation in the algebraic techniques, or whether this is an artifact of the current algebraic decomposition algorithms. With the assumption that the limitation is inherent in the algebraic approximation, investigation of Boolean decomposition becomes of high interest. Because circuits sizes are expanding rapidly, techniques for Boolean decomposition must be competitive with the performance of the algebraic techniques.

Boolean techniques such as two-level don't-care set minimization [9,20], Boolean resubstitution [20], and Boolean factoring [76] have been implemented in MIS with limited success. Part of the problem has been controlling the application of two-level minimization to avoid excessive use of computer time for the optimization. Limiting the computer time has allowed only a limited subset of the don't-care sets for a node in a network to be used. Nonetheless, for many circuits, the application of these Boolean techniques can lead to significant reductions in the size of the final network. Further research on these techniques is required to make them uniformly applicable. Other Boolean techniques, such as global-flow connection minimization [10], may also improve the results of an algebraic decomposition.

The current version of MIS lacks techniques to improve the testability of the optimized circuits. Keutzer and Hachtel [48] have shown that two-level minimization, algebraic decomposition and tree-based technology mapping produce 100% multiple-fault testable circuits. However, in practice, techniques other than algebraic techniques are also used

during the optimization. For example, selective-collapse, as described in Chapter 3, is a nonalgebraic operation. Likewise, other Boolean techniques, such as global-flow connection minimization and don't-care set minimization, can destroy this testability property. The circuits produced using Boolean techniques are often much smaller than algebraic techniques alone. A first attempt at improving the circuit testability is to apply test generation techniques (such as Socrates [67]) during the optimization to remove single-fault redundancies from the network.

Perhaps the weakest point in MIS is the lack of an effective algorithm for timing optimization at a technology-independent level. The goal for timing optimization at this level is to find techniques to greatly influence the timing behavior of a circuit by reducing the number of levels of logic along the critical path. The biggest problem in technology-independent timing optimization is finding the correct model for the timing behavior of a circuit before it is mapped into a technology. Computation of the delay for an unmapped circuit must be fast to allow grading the potential factors based on delay; however, this is at odds with accurate prediction of the timing behavior of the circuit. The timing model is complicated by having to consider detailed technology-specific effects such as loading introduced by routing, load-dependent delays for high-fanout nets, and the effect of timing optimizations such as device sizing and load-splitting. Initial work in this area has been promising [71], but further work is needed to improve the timing optimization results.

Appendix A

MIS

A.1 Introduction

MIS (Multiple-level Interactive Synthesis) is a program for multiple-level logic optimization. This appendix begins with a brief history and then describes the structure of the current version of MIS known as MIS-II. MIS-II is available as part of the OCT TOOLS set of computer-aided design programs [72]. Up-to-date user-level and programmer-level documentation is included with the source for MIS and is not reproduced here.

A.2 Background

MIS started as a research project at Berkeley in the summer of 1985. The goal was to develop a program for multiple-level logic optimization with special emphasis on the optimization algorithms to achieve high-quality results. Work primarily by myself and Albert Wang led to the C language program MIS. The last version of this program was Version 1.4. A paper describing MIS was published in ICCAD-86 [16] and in the IEEE Transactions on CAD [20].

The acronym MIS (Multiple-level Interactive Synthesis) was chosen to emphasize the interactive nature of the program. Many basic tools were developed for manipulating and inspecting a logic network (e.g., reading and writing a variety of formats, printing selected nodes in the Boolean network in either their sum-of-products or factored form, etc.) and these were made available through a command-line interface. MIS was made interactive to assist the algorithm developer; a script capability was intended for use in an actual design environment. A side benefit of the interactive nature of the program was that

it provided a good vehicle for teaching basic logic optimization concepts such as kernels, phase assignment, and multi-level don't-care set minimization.

MIS was successful in stimulating work on many new ideas for logic optimization. These included the sparse matrix techniques for algebraic factorization, the formulation of kernel and cube extraction as the rectangle covering problem, and the DAG-covering techniques for technology mapping. At the same time, it became clear that the first version of MIS had several limitations which made it difficult to grow. One of biggest difficulties was allowing different people to work together on the program as part of the same research effort.

Therefore, in April of 1987, the decision was made to re-implement the basic core of MIS. The goal for the re-implementation was to provide a better level of programmer documentation as well as a higher level of abstraction for the developer wanting to add new algorithms to the system. Because of the difficulty of program maintenance for the first version of MIS, there was a strong desire for a clean separation between the different algorithms in the program.

The C++ programming language was briefly considered as the language for MIS-II. C++ provides many attractive features such as enforced data hiding, function prototypes, function and operator overloading, and in-line expansion of functions for efficiency. On the negative side, C++ was immature at the time. Debugging facilities were nonexistent, compilation time was much slower than C, and many bugs and limitations were present in the Version 1.0 translator available from AT&T. Perhaps the biggest limitation was that C++ lacked techniques for writing generic packages at a level better than those supported by C itself. That is, either a great deal of effort was required to implement a type-safe generic package, or the short-cuts used in C were used to provide non-safe generic packages. After experimentation with C++ it became clear that although C++ provided some nice abstractions, it was not yet a significant improvement over simply enforcing a particular style within C.

Therefore, the decision was made to stick with C, but to enforce a strict division of the program into a set of communicating packages. Generic code was used for the lowest level packages and an interface was defined for the basic Boolean network and logic function manipulation routines. Packages obey the unenforced convention that all communication is through the interface documented and exported by the package. This program is now called MIS-II (or MIS Version 2.0).

All of the successful features of MIS Version 1.4 have been moved to MIS-II. Often simplifications and improvements became obvious once a clean line was drawn between the low level manipulations of logic functions and the higher-level concepts. For example, algebraic resubstitution was made much more efficient once the details of algebraic division were moved into a separate package; more attention was paid to techniques to limit the number of attempted divisions, leading to a tremendous decrease in the number of divisions performed. More importantly, many advances were added to the system which were not available in the earlier version. The sparse-matrix decomposition algorithms described in Chapter 3 and the tree-based technology mapping algorithms described in Chapter 4 were developed in their current form inside of MIS-II.

A.3 Program Organization

MIS-II is organized as a collection of packages. The packages divide into three main groups: generic code for basic data structures, the core data structures for a Boolean network and a Boolean logic function, and optimization algorithms written on top of the previous packages.

Table A.1 lists the packages in MIS-II for Version 2.0 dated March 1988. For each package, the author and a description is given. The number of lines refers to the number of source lines which contain a semicolon (an approximate measure of the amount of code in the package). The total number of source lines for all of the packages is about 50,000.

A.3.1 Package Conventions

The primary data structure in MIS-II is a Boolean network. The Boolean network is managed by the packages **node** and **network**.

To provide for separation of the packages, the **node** data structure contains a number of generic pointers. Each package reserves one of these generic pointers (called a *slot*) for its own use. A package registers *daemons* with the node package which are called when a node is allocated, freed, duplicated, or when a node has its logic function changed. This allows a package to maintain information on the node. A side benefit of this technique is that most packages have all of their data structures internal to the package. This enforces data hiding and allows a package to change its representation of the information on a node

Package	Author	Lines	Description
Generic Packages			
array	R. Rudell	180	Dynamic-array data structure
avl	P. Moore	350	Balanced binary-tree data structure
list	D. Harrison	370	Linked-list data structure
lsort	R. Rudell	130	Linked-list merge-sort algorithm
mm	R. Rudell	650	Improved memory allocation
mincov	R. Rudell	480	Sparse-matrix minimum-cover algorithm
sparse	R. Rudell	590	Sparse-matrix data structure
st	P. Moore	390	Hash table data structure
util	R. Rudell	530	Utility and portability routines
Core Packages			
command	R. Rudell	490	Command interpreter and dispatch
error	R. Rudell	25	Error handling package
espresso	R. Rudell	5,460	Two-level function manipulation
gcd	P. McGeer	180	Algebraic greatest common divisor
io	R. Rudell	1,370	Input/output (PLA, BLIF, equations)
main	R. Rudell	170	Argument parsing, main program
network	R. Rudell	900	Boolean network data structure
node	R. Rudell	1,730	Logic function data structure
octio	R. Spickelmier	1,090	Input/output (OCT)
Optimization Packages			
decomp	A. Wang	440	Single node decomposition
delay	R. Rudell	520	Delay evaluation routines
extract	R. Rudell	1,410	Multiple node decomposition
factor	A. Wang	710	Single node factoring
genlib	R. Rudell	1,360	Pattern generation for technology mapping
map	R. Rudell	1,780	Technology mapping
phase	A. Wang	690	Phase assignment
resub	A. Wang	70	Algebraic resubstitution
sim	R. Rudell	300	Simulation
simplify	A. Wang	380	Two-level minimization using derived DC-sets
test	R. Rudell	60	Prototype package

Table A.1: Packages in MIS-II.

without forcing a re-compile of the entire program. This is a significant advantage, especially when several people are working on the same program at the same time.

Each package in MIS-II obeys the following conventions.

1. `package.h` contains the exported definitions for the package. `mis.h` includes the exported header file for each package so that the exported definitions are visible to all other packages.

2. `package_int.h` is used for private definitions within the package.

3. A package exports the routines `init_package()` and `end_package()` which are called at program initiation and termination.

4. All other symbols exported by a package are prefixed with the package name (e.g., `package_function()`).

5. When possible, routines are made `static` (i.e., local to the file containing the function) and not global.

6. A package registers any number of commands with the command interpreter. Each command is passed a Boolean network (the *current network*), and a set of arguments in C (`argc, argv`) style.

7. A package reserves a slot on a node for its own data and registers daemons with the node package.

8. Each command added by a package is documented with a Unix-style man page; the concatenation of these manual pages forms the manual page for the MIS-II program. The commands facilitate testing and debugging and provide user-level access to the package functions.

9. `package.doc` contains the programmer-level documentation for each package. This describes the data structures and routines exported from the package. Each routine is documented with a C function prototype and a brief description of what the routine does.

10. Each package is required to free any memory which it allocates. `end_package()` gives each package one last chance to do this before the program terminates. The memory

allocator mm provides a stack back-trace, with function names and line numbers, tracing the source of any leaks. This policy greatly simplifies checking each package for memory leaks.

11. The package must produce no lint errors (even if the errors are due to a bug in lint). This avoids the problem of missing real error messages against a background of false error messages.

A.3.2 Generic Packages

Many of the generic packages were not developed for MIS-II, but were developed within the CAD group for other reasons. This includes the balanced binary tree package (avl), the linked-list package (list and lsort) and the hash-table package (st). These packages were incorporated into MIS-II. In some cases, minor modifications were made to increase portability and improve either performance or error handling. The dynamic array package array was developed for MIS-II to round out the set of generic data structure algorithms.

The memory allocation package mm was developed for ESPRESSO, but features were added over time so that the memory allocator became an important part of the development environment. These features include the standard tricks to detect programming errors (fill block with garbage on allocation and free, pad block with extra words to catch write past end of a memory block, etc.), and a technique borrowed from the Magic memory allocator [37] to trace blocks of memory which are allocated and never freed. A decision was made not to allow the portability of MIS-II to rely on the portability of the memory package; therefore, the semantics of the C memory interface were not altered or extended. In several instances, this allowed the mm package to be removed from MIS-II during a port to a new machine, making it much easier to move the program to a new system.

util was added as it became clear that there were enough differences between C run-time libraries which were tedious to program around in each package. The approach taken in util was to provide the abstraction that the underlying run-time library conforms to the ANSI C Standard. Any differences between a given operating system and the standard are conditional compile options within the util package. No attempt was made to make util complete in this respect; instead, features were added to util as specific problems were identified.

The **sparse** and **mincov** packages were developed for the exact minimization algorithm described in Chapter 2. The sparse matrix package was set up as a generic package to allow it to be re-used for other applications. For example, **sparse** is also used by the **extract** and **phase** packages where the sparse matrix provides an elegant abstraction as well as an efficient implementation. **mincov** is also generic in that it can be used for a variety of covering problems which arise in logic optimization. For example, **mincov** has been used for exact solutions to the rectangle covering problem, the two-level minimization problem, and the DAG-covering problem.

A.3.3 Core Packages

The **main** package defines the main program and its command line parsing, and contains the calls to the individual packages to allow each package its own initialization code.

The **command** package provides the basic command line parsing and calls the command routines registered by other packages. Built into the command interpreter is the notion of the current network to which all commands apply. Packages register command names and a function pointer to the code which implements the command. The command, when invoked, is passed the current network, and a set of arguments in the C `argc, argv` style. `getopt()` is used by most of the commands to parse the command line arguments.

The **io** and **octio** packages provide the basic operations of reading and writing Boolean networks and their logic functions in a variety of formats. This includes read and write for the Berkeley PLA format, eqntott equation format, BLIF (Berkeley Logic Intermediate Form) format, LIF (Logic Interchange Form), and designs stored in the Oct database.

The **espresso**, **network**, and **node** packages provide the basic data structures for the logic optimization routines in MIS-II. **network** provides the basic Boolean network graph structure. This includes the primary inputs and primary outputs, and routines for traversing the graph. **node** defines the logic function representation for each node in the network and supports a large set of manipulation routines for these logic functions. The data structure for the logic function is taken directly from the **espresso** data structures, which allowed much of the work for these routines (e.g., function intersection, simplification, prime generation, etc.) to be borrowed directly from the espresso program.

An interesting feature added to the **node** package was the concept of a generic slot which another package could reserve on any node, and a set of daemons which are invoked as the node is allocated, freed, copied, or modified. The idea of tagging a node with package specific data and allowing for that data to be automatically updated proved to be essential in allowing a clean separation between the packages. For example, the **factor** package was free to maintain its own representation of the logic function at a node (i.e., the factored form), independent of the node package. The new package **bdd** was able to use the same technique to store the binary-decision diagram representation of the logic function at the node. Likewise, the **map** package was able to associate a particular gate from the library with a node – a concept somewhat foreign to the original notion of a Boolean network. In each case, no modifications were required to the **node** package to add these new features.

A.3.4 Optimization packages

The remaining packages deal with the logic optimization algorithms that have been implemented on top of the existing set of packages.

The **factor** and **decomp** packages perform basic node-at-a-time factoring and decomposition.

extract implements the algorithms described in Chapter 3 for common factor extraction using the rectangle-covering paradigm. The routines are built on top of the sparse matrix package for efficiency, and contain a number of different options which were used during experimentation and development.

gcd implements the algebraic prime factorization and greatest common divisor algorithms as proposed by McGeer and Brayton.

map implements the tree-covering approximation for technology mapping described in Chapter 4. The related package **genlib** was developed to count the number of CMOS complex gates and the number of patterns needed for a given sized library. The purpose of **genlib** in MIS-II is to generate the mapping patterns for each gate in a library. The timing optimization for technology mapping relies on the **delay** package which implements the rise-fall and intrinsic-drive delay model described in Chapter 4.

phase implements phase assignment to minimize the number of inverters in a logic network. **phase** uses the sparse matrix package to build a representation of the Boolean network which is optimized for the phase assignment computations.

resub implements the algebraic resubstitution algorithms and the filters to effectively allow exploring all possible divisions of one node by another.

sim and **test** are simple packages put together for demonstration purposes. **sim** implements a simple 0-1 simulator for the Boolean network. **test** is an example of a complete but trivial package. It is intended as the starting point for development of a new package.

A.3.5 New packages

As of this writing, several new packages have been added to MIS-II for Version 2.1. Also, many different authors are now taking responsibility for maintaining a package within MIS-II. The new packages include a binary-decision diagram representation for the logic function at a node (**bdd** by S. Malik), improved two-level minimization algorithms in the presence of large don't-care sets (**minimize** by A. Malik), graphical representation of Boolean networks for debugging purposes (**graphics** by W. Christopher), techniques for global timing reorganization (**speed** by A. Wang and K. Singh), and a generic package for max-flow min-cut computations on a graph (**maxflow** by T. Ma).

A.4 Retrospect

The most important organizational aspect of MIS-II is the separation of the program into distinct packages. A single author takes responsibility for all aspects of defining a new command, installing the command, implementing the command argument parsing, and writing the final code invoked by the command. This allows someone to leverage the existing MIS-II environment, including the input/output routines and basic data structures, to experiment with different optimization algorithms. The use of node daemons allows the packages to remain independent.

The software organization of MIS Version 1.0 had a negative impact on the ability of people to use the program and reach a point where they could experiment with logic optimization algorithms. This was made clear in the Spring of 1986 during the logic synthesis class EE290H. Several ambitious projects were scheduled and attempted, but the poor implementation of MIS Version 1.0 held back the degree to which the projects could be completed. This is contrasted to the logic synthesis course taught in the Spring of 1988, where MIS-II allowed several major projects to be completed and integrated into the current version of MIS-II.

One possible improvement to the current system would be to reduce the number of steps required to integrate a new package into MIS-II. It is still necessary to edit the global header file to add a `#include` for the new package; it is necessary in many cases to modify the `node` header file to define a slot on the node for the new package; it is necessary to modify the routines `init_mis()` and `end_mis()` to call the initialization and termination routines for the new package; it is necessary to make symbolic links for the documentation and header files defined by the new package; and it is necessary to insure that the `Makefile` for the package is correct. Several of these steps could be eliminated or improved.

For example, a function `node_register_slot()` could be used to allocate extra fields on a node dynamically rather than forcing the fields to exist statically on the node. The extra overhead at run-time for this scheme is negligible. An automatic `Makefile` production facility would remove the dependence on the developer to create a correct `Makefile` for a new package and could solve the problem of creating the symbolic links for the package documentation and header files. It is hard to imagine the extra benefit of a scheme which would update the global header file or modify the startup routines to include the new package; however, this is certainly possible as well.

One feature which annoys many many first time MIS-II developers is that a node (i.e., a logic function) is allowed which is not part of any network. This can lead to confusion if an attempt is made to examine the fan-out of the node. Likewise, the fan-out of the fan-in of a node that is not in a network does not point to the free-standing node. Another problem is that the programmer is required to perform garbage collection of intermediate results. For example, computation of $f = a * b + c$ is implemented as $t = a * b; f = t + c; free(t)$. Both of these problems could be solved by forcing the node creation routines to insert the new node into a network. Garbage collection, which would be demand-driven, would occur when nodes with no fanout in the network are deleted.

Many other features were always planned, but never implemented. There are:

1. The current set of node daemons is not sufficient to support an incremental delay simulator. The addition of several new daemons could solve this problem.

2. Network daemons and network slots are needed to allow packages to associate persistent information with a network.

3. The command interpreter is missing several useful features of *csh*, such as history,

input and output redirection, and variable substitution. The command interpreter also lacks the ability to deal with more than one network at a time.

Bibliography

[1] *Data Encryption Standard.* U. S. Department of Commerce, National Bureau of Standards, January 1977. Federal Information Processing Standards Publication (FIPS PUB 46).

[2] *LCA 10000 Macrocells Databook.* LSI Logic Corporation, 1986.

[3] *SLM/DLM Cell Family Documentation.* Technical Report, Microelectronics Design Division, Institute for Technology Development, Mississippi State University, November 1986. Draft.

[4] *VAX DECSIM Reference Manual.* Digital Equipment Corporation, December 1985. Not generally available.

[5] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, :333–340, June 1975.

[6] A. Aho and S. Johnson. Optimal code generation for expression trees. *Jour. ACM*, :488–501, July 1976.

[7] R. Ashenhurst. The decomposition of switching functions. In *Proc. Int. Symp. Theory of Switching*, pages 74–116, April 1959.

[8] K. Bartlett. *A Weak Division Approach to Multilevel Synthesis.* Master's thesis, University of Colorado, 1986.

[9] K. Bartlett, R. Brayton, G. Hachtel, R. Jacoby, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multilevel logic minimization using implicit don't-cares. IEEE *Trans. Comp. Aided. Design*, CAD-7(6):723–740, June 1988.

[10] L. Berman and L. Trevillyan. A global approach to circuit size reduction. In *Advanced Research in VLSI, 5th MIT Conference*, pages 203–214, MIT Press, 1988.

[11] D. Bostick, G. Hachtel, R. Jacoby, M. Lightner, P. Moceyunas, C. Morrison, and D. Ravenscroft. The boulder optimal logic design system. In *Proc. Int. Conf. CAD (ICCAD-87)*, pages 62–65, November 1987.

[12] R. Boyer and J. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[13] R. Brayton. Personal Communication, July 1985.

[14] R. Brayton. Factoring logic functions. IBM *J. Res. Develop.*, 31(2):187–198, March 1987.

[15] R. Brayton, N. Brenner, C. Chen, G. DeMicheli, C. McMullen, and R. Otten. The yorktown silicon compiler. In *Proc. Int. Symp. Circ. Syst. (ISCAS-85)*, 1985.

[16] R. Brayton, E. Detjens, S. Krishna, T. Ma, P. McGeer, L. Pei, N. Phillips, R. Rudell, R. Segal, A. Wang, R. Yung, and A. Sangiovanni-Vincentelli. Multiple-level logic optimization system. In *Proc. Int. Conf. CAD (ICCAD-86)*, pages 356–359, November 1986.

[17] R. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *Proc. Int. Symp. Circ. Syst. (ISCAS-82)*, pages 49–54, 1982.

[18] R. Brayton and C. McMullen. Synthesis and optimization of multistage logic. In *Proc. Int. Conf. Comp. Des. (ICCD-84)*, pages 23–28, 1984.

[19] R. Brayton, C. McMullen, G. Hachtel, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[20] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Mis: a multiple-level logic optimization system. IEEE *Trans. Comp. Aided. Design*, CAD-6(6):1062–1081, November 1987.

[21] D. Brown. A state-machine synthesizer - sms. In *Proc. 18th Design Automation Conference*, pages 301–304, June 1981.

[22] J. Bruno and R. Sethi. Code generation for a one register machine. *Jour. ACM*, :146–160, August 1975.

[23] H. Curtis. Generalized tree circuit – the basic building block of an extended decomposition theory. *Jour. ACM*, (8):562–581, 1961.

[24] M. Dagenais, V. Agarwal, and N. Rumin. Mcboole: a new procedure for exact logic minimization. IEEE *Trans. Comp. Aided. Design*, C-33:229–238, January 1986.

[25] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L. Trevillyan. Lss: a system for production logic synthesis. IBM *J. Res. Develop.*, 28(5):537–545, September 1984.

[26] J. Darringer, W. Joyner, L. Berman, and L. Trevillyan. Logic synthesis through local transformations. IBM *J. Res. Develop.*, 25(4):272–280, July 1981.

[27] E. Davidson. An algorithm for nand decomposition under network constraints. IEEE *Trans. Comp.*, C-18(12):1098–1109, December 1969.

[28] D. Dietmeyer and Y. Su. Logic design automation of fan-in limited nand networks. IEEE *Trans. Comp.*, C-18(1):11–22, January 1969.

[29] R. Katz (ed). Design decisions in spur. *IEEE Computer*, 19(10):8–22, November 1986.

[30] H. Fleisher and L. Maissel. An introduction to array logic. IBM *J. Res. Develop.*, 19:98–109, March 1975.

[31] M. Garey and D. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979.

[32] J. Gimpel. The minimization of tant networks. IEEE *Trans. Elec. Comp.*, EC-16(1):18–38, February 1967.

[33] J. Gimpel. A reduction technique for prime implicant tables. IEEE *Trans. Elec. Comp.*, EC-14:535–541, August 1965.

[34] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. IEEE *Trans. Elec. Comp.*, EC-14(3):350–359, June 1965.

[35] D. Gregory. Personal Communication, April 1988.

[36] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel. Socrates: a system for automatically synthesizing and optimizing combinational logic. In *Proc. 23th Design Automation Conference*, pages 79–85, June 1986.

[37] G. Hamachi, R. Mayo, J. Ousterhout, W. Scott, and G. Taylor, editors. *1985 VLSI Tools: More Works by the Original Artists.* University of California, Berkeley, 1985. Report No. UCB/CSD 85/225.

[38] D. Harrison. *VEM: Interactive Graphics for Oct.* Master's thesis, University of California, Berkeley, May 1989.

[39] D. Harrison, P. Moore, R. Spickelmier, and A. R. Newton. Data management and graphics editing in the berkeley design environment. In *Proc. Int. Conf. CAD (ICCAD-86)*, pages 20–24, November 1986.

[40] L. Hellerman. A catalog of three-variable or-invert and and-invert logical circuits. IEEE *Trans. Elec. Comp.*, EC-12(3):198–223, June 1963.

[41] C. Hoffman and M. O'Donnell. Pattern matching in trees. *Jour. ACM*, 29(1):68–95, January 1982.

[42] S. Hong, R. Cain, and D. Ostapko. Mini: a heuristic approach for logic minimization. IBM *J. Res. Develop.*, 18:443–458, September 1974.

[43] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.

[44] S. Robinson III and R. House. Gimpel's reduction technique extended to the covering problem with costs. IEEE *Trans. Elec. Comp.*, EC-16:509–514, August 1967.

[45] J. Ishikawa, H. Sato, M. Hiramine, K. Ishida, S. Oguri, Y. Kazuma, and S. Murai. A rule based reorganization system lores/ex. In *Proc. Int. Conf. Comp. Des. (ICCD-88)*, pages 262–266, October 1988.

[46] W. Joyner, L. Trevillyan, D. Brand, T. Nix, and S. Gundersen. Technology adaptation in logic synthesis. In *Proc. 23th Design Automation Conference*, pages 94–100, June 1986.

[47] R. Karp. Combinatorics, complexity, and randomness. *Comm. of the ACM*, 29(2):98–111, February 1986.

[48] K. Keutzer. Personal Communication, February 1989.

[49] K. Keutzer. Dagon: technology binding and local optimization by dag matching. In *Proc. 24th Design Automation Conference*, pages 341–347, June 1987.

[50] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[51] T. Laidig. Ciftooct – convert cif files to oct. In Rick Spickelmier, editor, *Oct Tools Distribution 2.1*, University of California, Berkeley, March 1988.

[52] E. McCluskey. Minimization of boolean functions. *Bell System Technical Journal*, 35:1417–1444, April 1956.

[53] C. McMullen and J. Shearer. Prime implicants, minimum covers, and the complexity of logic simplification. IEEE *Trans. Comp.*, C-35(8):761–762, August 1986.

[54] R. Morris and K. Thompson. Password security: a case history. In *Unix Programmers Manual's Manual, Seventh Edition, Volume 2A*, Bell Laboratories, January 1979.

[55] W. Quine. The problem of simplifying truth functions. *Amer. Math. Monthly*, 59:521–531, 1952.

[56] J. Reed. *YACR2: Yet Another Channel Router.* Master's thesis, University of California, Berkeley, February 1985. Memorandum UCB/ERL M85/16.

[57] J. Roth. *Computer Logic, Testing, and Verification.* Computer Science Press, 1981.

[58] J. Roth and R. Karp. Minimization over boolean graphs. IBM *J. Res. Develop.*, 6(2):227–238, April 1962.

[59] R. Rudell. *Multiple-Valued Logic Minimization for PLA Synthesis.* Master's thesis, University of California, Berkeley, June 1986. Memorandum UCB/ERL M86/65.

[60] R. Rudell. Octmm – oct mask modification program. In Rick Spickelmier, editor, *Oct Tools Distribution 2.1*, University of California, Berkeley, March 1988.

[61] R. Rudell. Wolfe – oct interface to the timberwolf standard cell placement program. In Rick Spickelmier, editor, *Oct Tools Distribution 2.1*, University of California, Berkeley, March 1988.

[62] R. Rudell and A. Sangiovanni-Vincentelli. Exact minimization of multiple-valued functions for pla optimization. In *Proc. Int. Conf. CAD (ICCAD-86)*, pages 352–355, November 1986.

[63] T. Sasao. An algorithm to derive the complement of a binary function with multiple-valued inputs. IEEE *Trans. Comp.*, C-34:131–140, February 1985.

[64] T. Sasao. Input variable assignment and output phase optimization of pla's. IEEE *Trans. Comp.*, C-33:879–894, October 1984.

[65] T. Sasao. Multiple-valued decomposition of generalized boolean functions and the complexity of programmable logic arrays. IEEE *Trans. Comp.*, C-30:635–643, September 1981.

[66] T. Sasao. Tautology checking algorithms for multiple-valued input binary functions and their application. In *Proc. 14th Int. Symp. Multiple-Valued Logic*, 1984.

[67] M. Schulz, E. Trischler, and T. Sarfert. Socrates: a highly efficient automatic test pattern generation system. IEEE *Trans. Comp. Aided. Design*, CAD-7(1):126–137, January 1988.

[68] C. Sechen and A. Sangiovanni-Vincentelli. The timberwolf placement and routing package. IEEE *Jour. Solid State Circ.*, 20(2):510, April 1985.

[69] R. Segal. *BDSYN: Logic Description Translator; BDSIM: Switch-Level Simulator.* Master's thesis, University of California, Berkeley, May 1987. Memorandum UCB/ERL M87/33.

[70] P. Simanyi. Pop reference manual. In *Berkeley CAD Tools User's Manual*, University of California, Berkeley, September 1983.

[71] K. Singh, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *Proc. Int. Conf. CAD (ICCAD-88)*, pages 282–285. November 1988.

[72] R. Spickelmier, editor. *Oct Tools Distribution 2.1.* University of California, Berkeley, March 1988.

[73] S. Tjiang. *Twig Reference Manual.* Technical Report, AT&T Bell Laboratories, 1985.

[74] T. Uehara and W. vanCleemput. Optimal layout of cmos functional arrays. IEEE *Trans. Comp.*, C-30(5):305–312, May 1981.

[75] A. Wang. Personal Communication, July 1988.

[76] A. Wang. *Algorithms for Multi-Level Logic Optimization.* PhD thesis, University of California, Berkeley, 1989.

[77] A. Wang. *Optimization of Multi-level Logic.* Technical Report, University of California, Berkeley, June 1985. ee244/cs292h Final Project Report.