

CSEE W4861y
Prof. Steven Nowick

LOGIC SYNTHESIS FOR VLSI DESIGN

by

Richard L. Rudell

Copyright © 1989

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

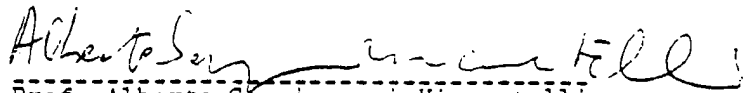
Richard L. Rudell

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.


Prof. Alberto Sangiovanni-Vincentelli
Thesis Committee Chairman

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 <code>gen_rectangles_recur</code>	68
3.3	Algorithm <code>gen_rectangles</code>	69
3.4	Algorithm <code>ping_pong</code>	71
3.5	Algorithm <code>ping_pong_row</code>	72
3.6	Algorithm <code>greedy_row</code>	73
3.7	Algorithm <code>greedy_extract</code>	74
3.8	Algorithm <code>covering_extract</code>	75
3.9	Algorithm <code>rect_prime_cover</code>	75
3.10	Algorithm <code>rect_reduce</code>	77
3.11	Algorithm <code>eliminate</code>	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 <code>generate_all_matches</code>	113
4.15	Algorithm <code>make_edge_list</code>	114
4.16	Sample pattern graph for <code>edge_list</code>	115
4.17	Edge-list for the graph of Figure 4.16.	116
4.18	Algorithm <code>graph_match</code>	117
4.19	Example graph for DAG-covering.	120
4.20	Algorithm <code>optimal_area_cover</code>	124
4.21	Algorithm <code>optimal_delay_cover</code>	128

4.22	Algorithm <code>delay_technology_map</code>	130
4.23	Algorithm <code>optimal_area_under_delay_constraint</code>	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 <code>generate_nand_trees</code>	153
5.1	DES Equations.	173
5.2	MIS technology library for the MSU library.	181
5.3	BDS description for <code>onestage</code>	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.	33
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.	170
5.3	Run-time for each step in the DES design.	180
A.1	Packages in MIS-III.	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 $f(x_1, \dots, x_n)$ is a decomposition into the form $F(y_1, \dots, y_s, \phi)$ and $\phi(z_1, \dots, z_{n-s})$. 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 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 Ashenurst 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.

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

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

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 $i = 1, \dots, n$, 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, $S = X_1^{S_1} X_2^{S_2} \dots X_n^{S_n}$ and $T = X_1^{T_1} X_2^{T_2} \dots X_n^{T_n}$ represent product terms.

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

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

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

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

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

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 $X_1^{S_1 \cap T_1} \dots X_i^{S_i \cup T_i} \dots X_n^{S_n \cap T_n}$. 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

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

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:

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

where $c_j^i = 0$ 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 \bar{I} :

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

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:

$$\begin{aligned} E &= \{c \in P \mid c \not\subseteq (P - c)\} \\ R_t &= \{c \in (P - E) \mid c \subseteq (E \cup D)\} \\ R_p &= P - (E \cup R_t) \end{aligned}$$

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 $A \cdot x^T \geq (1, 1, \dots, 1)^T$ and $\sum_{j=1}^m x_j c_j$ is minimum.

The constraint $A \cdot x^T \geq (1, 1, \dots, 1)^T$ 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:

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

where c_1 and c_2 are single variables with a cost c^* , $S_i, i = 1 \dots n$ and $T_j, j = 1 \dots m$ 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 $T = \prod_{j=1}^m T_j$.

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 \bar{M}_1 ; otherwise, remove c_2 to create \bar{M}_1 . (Either c_1 or c_2 must be in M .) The variable set \bar{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 . \square

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:

$$\begin{array}{ccc} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{array}$$

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 theunate 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
<i>trivial</i>	9	9
<i>noncyclic</i>	56	56
<i>cyclic-solved</i>	49	49
<i>cyclic-unsolved</i>	3	0
<i>too many primes</i>	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
<i>trivial</i>	9	9	9
<i>noncyclic</i>	56	56	56
<i>cyclic-solved</i>	49	49	21
<i>cyclic-unsolved</i>	3	0	0
<i>too many primes</i>	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 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
col4	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	* 140	indust	noncyclic	558	82	108
exp	8/18	* 89	indust	noncyclic	238	30	50
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
ml	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

2.11. PLA TEST SET CLASSIFICATION

39

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	10/10	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	50
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