

Copyright © 1990, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

**COMBINATIONAL LOGIC OPTIMIZATION
TECHNIQUES IN SEQUENTIAL LOGIC
SYNTHESIS**

by

Sharad Malik

Memorandum No. UCB/ERL M90/115

28 November 1990

**COMBINATIONAL LOGIC OPTIMIZATION
TECHNIQUES IN SEQUENTIAL LOGIC
SYNTHESIS**

by

Sharad Malik

Memorandum No. UCB/ERL M90/115

28 November 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

Combinational Logic Optimization Techniques in Sequential Logic Synthesis

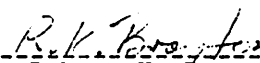
Sharad Malik

University of California
Berkeley, California

Department of Electrical Engineering
and Computer Science
Computer Science Division

Abstract

Designing an integrated circuit with over one hundred thousand components is a significantly complicated task; impossible to handle without computing aids. Computer-aided design tools are used in all aspects of the design: logical design of functional units, physical design of gates and modules, placement and interconnect routing, logical and timing verification, and management of design data. Of these, the automatic design of the logic components, referred to as logic synthesis, was the last to come about; an indication of the inherent difficulty of this task. There was a lack of sophisticated logic optimization techniques needed to generate high quality results. This prompted research in this area and as a result there are now several commercially available design aids. Thus far, logic synthesis has largely concentrated on combinational logic. This is an incomplete view, since digital circuits are, in general, sequential in nature. This thesis attempts to overcome this limitation. It presents techniques for the optimization of sequential logic circuits. In particular, it considers extensions of known combinational logic optimization techniques that are applicable in sequential logic synthesis. The contributions are in two areas. In the first part it is shown how existing combinational logic optimization techniques can be directly applied in the expanded context of sequential logic synthesis. The presented approach maximally exploits combinational logic optimization techniques, i.e. it can potentially detect any logical relationships that exist between any two gates in the circuit, they need not be part of the same combinational logic block. In the second part, techniques for the optimization of multi-level circuits with multiple-valued inputs are presented. Logic optimization techniques used in multi-level circuits have been extended to handle multiple-valued inputs. In addition to being a significant result in its own right, this has direct application in the state assignment problem in sequential logic synthesis. In both these areas, theoretical results are presented and implementation issues and practical experiences discussed.



Prof. Robert K. Brayton
Thesis Committee Chairman

Acknowledgements

*And its been this way for five long years
Since we signed our souls away.*

Jethro Tull, "War Child"

The five years that I have spent as a graduate student at Berkeley have been, at some time or the other, tough, exciting, fun and depressing. If I have made it through this, it is only because of the support, encouragement and friendship of several people. I got by with a little help from my friends, nay, with lots of help from my friends.

There are two people without whom this thesis would never have been possible; Bob Brayton and Alberto Sangiovanni-Vincentelli. They introduced me to the field of logic synthesis and have taught me most of the things that I know about this subject.

Bob has been my research advisor for the last two and a half years. The time that I have spent working on research with him has been extremely exciting – Bob's enthusiasm and desire to pursue the unknown has been truly inspirational for me. From him I have also tried to learn mathematical precision, both in the formulation and solution of problems, as well as in written and oral exposition. I consider myself truly fortunate for having worked with him.

I owe a lot to Alberto. First and foremost for giving me the opportunity to be part of the excitement of the cadgroup, without which none of the work I have been involved in would have been possible. I have profited from his advice, both technical and professional. From Alberto I have learnt, (at least I think so!) the need for systematic rigor in research, and clarity and focus in presentation. He has gone through multiple drafts of my writings and several rehearsals for my presentations; painstakingly pointing out what needs to be done differently and why. Thanks Alberto.

Two other people have had a significant impact on my research career. Kurt Keutzer introduced me to intensity in research, with his work hard and play hard attitude. He has also shown me the planning and organization that goes into a successful research career¹. I am indebted to Rick Rudell for guiding me in my early years in graduate school; for being forever willing to help me in my work. This support was very important for me. From him I have tried to learn the importance of practical concerns in the application of research ideas.

¹This is no way implies that I support his views on any matter!

I would like to thank Randy Katz for support during the initial stages of my graduate student life. Randy encouraged me to pursue the path I thought best for my research, I am thankful for that. Thanks are also due to him for serving on my qualifying exam committee. I would also like to thank Prof. Dorit Hochbaum for sparing the time to serve on both my qualifying exam as well as thesis committees.

Several people have made a direct contribution to the work in this thesis. All the work was done with Bob and Alberto; it is impossible to separate out their contributions. Luciano Lavagno was responsible for the implementation of MIS-MV, without him it may not have become a reality. The initial work in retiming and resynthesis was done with the assistance of Ellen Sentovich. The work on the performance optimization of pipelined circuits was done with the assistance of Kanwar Jit Singh.

Cadgroup provided the unique environment for most of my work. Profs. Brayton, Newton, Pederson and Sangiovanni-Vincentelli deserve special thanks for their efforts in establishing this environment for us, the students in the group. The funding for my research was provided by DARPA, NSF and industrial grants from AT&T, Bell Northern, California Micro, IBM, Intel and Motorola. I gratefully acknowledge that. Thanks also to Digital Equipment Corporation for the equipment used. Brad Krebs and Beorn Johnson provided ready assistance with any hardware and software problems. Rick Spickelmier provided ample assistance with tools he did and didn't write. I had many useful and enjoyable interactions with other members in the group. In alphabetical order they are: Pranav Ashar, Wendell Baker, Srinivas Devadas, Tim Kam, Luciano Lavagno, Bill Lin, Rick McGeer, Cho Moon, Rajeev Murgai, Alex Saldanha, Hamid Savoj, Ellen Sentovich, Narendra Shenoy, Kanwar Jit Singh, Arvind Srinivasan, Hervé Touati, Tiziano Villa, and Albert Wang. Rick McGeer deserves special mention. His firm belief that research must be fun helped keep the right perspective on things. Thanks to Ellen for carefully reading through this manuscript and in clarifying the ideas presented in the appendix. Thanks also to Umakanta Choudhury for letting me commandeer his LaTeX book during the writing of this thesis.

There are several researchers outside of Berkeley that I wish to thank. Al Dunlop provided me with the opportunity to spend a fruitful summer at AT&T Bell Labs., Murray Hill. Len Berman and Larry Carter arranged for me to spend some time at IBM Research Division, Yorktown Heights. I thank Tzvi Ben-Tzur, Len Berman, Randy Bryant, Giovanni De Micheli, Gary Hachtel, Bob Kurshan, Naotaka Maeda, and Louise Trevillyan for time spent in technical interactions.

Kathryn Crabtree was a wonderful source of support, providing encouragement when things were not going right. And without her, I couldn't have figured out the Berkeley bureaucracy in a million years. Flora Oviedo provided valuable assistance with things administrative as well as a ready smile to brighten up the cloudiest of days. Ted Goode was always extremely helpful at the foreign students office, never letting me feel that I was a "non-resident alien".

Ashi and Jean Malik provided me with the home away from home, giving me love, support and good food, as and when needed. Thanks a lot guys. Thanks also to the Becks; Fred and Polly, Dan and Annette, and Louie for making me feel part of the family. Ruth Brayton was wonderful as my local Mom, giving affection and good advice in equal measure.

Life in Berkeley would not have been the same without my buddies, who made life enjoyable even when it was trying hard to be otherwise. Tons of thanks to Jhingu, Khedkar, Huzur, K. J., Munnu, Savita, Madhu, Mots, Asha, Tarun, Shashi, Keshav, Nandu, Vedant, Ajay Amar, Murgai, Ashok Singhal, and Jaggu for providing the much needed support mechanism. Thanks also to my roommates, Tom Chen, Huzur and D. G. for putting up with my idiosyncrasies.

In its own way Berkeley made life enjoyable and interesting. Telegraph Avenue chipped in with Café Med and Moe's. Sam and Andy and the rest of the crew at Coffee Connection made the afternoon bianco ritual a pleasure. The 49er's and the A's kept the morale up with their spectacular efforts.

My aunt and uncle, Mrs. and Mr. V. N. Bakshi provided me with encouragement and the environment to pursue my academics in Delhi. My sisters Madhuri and Payal contributed in their own little way by putting up with me at home, hoping that some day I would be normal. It is not possible to thank the next three people enough with mere words. Nonetheless, thanks to Aarti for all her love and willingness to undertake a difficult journey with me. Thanks to Mom for her love and sacrifices that have made all this possible. And last but not the least, thanks to Darshan Uncle, for being more than a father to me. It is to him that I dedicate this thesis.

Contents

Table of Contents	v
List of Figures	ix
List of Tables	xi
1 Introduction	1
1.1 VLSI Design and Logic Synthesis	1
1.2 Sequential Logic Synthesis	4
1.2.1 Problem Classification	5
1.2.2 Previous Work	10
1.3 Thesis Overview	15
 I Retiming and Resynthesis	 17
2 Basic Ideas	19
2.1 Clocking Methodology	20
2.2 Overview	21
2.3 Theoretical Formulation	22
2.3.1 Retiming: An Overview	23
2.3.2 Extensions to Retiming	26
2.3.3 Conditions for Peripheral Retiming	31
2.3.4 Computing the Path Weight Matrix	38
2.3.5 Solving the Path Weight Matrix	42
2.3.6 Legal Resynthesis Operations	42
2.4 Peripherally Retimable Circuits: General Topology	46
2.5 Optimizing Sequential Circuits	49
2.6 Computing Equivalent States Across Optimizations	52
 3 Implications and Applications	 57
3.1 Relationship to Logic Testing	57
3.2 Relationship to State Assignment	60
3.3 Performance Optimization	67

3.3.1	Two Problems in Performance Optimization	68
3.3.2	Main Results	71
4	Practical Experiences	79
4.1	Implementation Issues	79
4.1.1	Growing Pipelined Sub-Circuits from a Seed	79
4.1.2	Clustering Combinational Logic Blocks	84
4.2	Experimental Results: Area Optimization	87
4.2.1	Experimental Circuits	87
4.2.2	Experimental Procedure and Results	88
4.2.3	Analysis of Experimental Results	88
4.3	Experimental Results: Performance Optimization	96
4.3.1	Example Circuits and Experimental Procedure	96
II	Handling Symbolic Inputs	99
5	Multi-Level Logic Minimization	101
5.1	Multi-Level Optimization Techniques	102
5.2	Overview	103
5.3	Circuit and Function Representation	103
5.4	Circuit Decomposition Using Kernels	106
5.4.1	Kernels and Kernel Intersections	106
5.4.2	Kernels and Multiple-Valued Variables	109
5.5	Circuit Decomposition Using Common Cubes	120
5.5.1	Common Cube Extraction with Binary Variables	120
5.5.2	Common Cube Extraction with Multiple-Valued Variables	121
5.6	Circuit Simplification	123
5.7	Logic Verification	125
6	Practical Experiences	127
6.1	Implementation Issues	127
6.1.1	Size Estimation in Algebraic Decomposition	128
6.1.2	Incompletely Specified Literals	129
6.1.3	Satisfiable Constraint Matrices	132
6.1.4	The Encoding Problem	136
6.2	Experimental Results	137
6.2.1	An Example MIS-MV Run	142
7	Conclusions	149
A	Discrete Mappings and Boolean Functions	151
A.1	Introduction	151
A.2	Boolean Functions: A Review	152
A.2.1	Boolean Algebras	152
A.2.2	Boolean Formulas	154

CONTENTS

vii

A.2.3 Boolean Functions	154
A.3 Discrete Mappings	154
A.4 Compact Representations of Boolean Functions	156
A.5 Expressions Representing Discrete Mappings	157
A.6 Conclusions	161
Bibliography	163

(1)	Introduction	1
(2)	General Principles of the Theory of the	2
(3)	Structure of the Matter	3
(4)	Properties of the Matter	4
(5)	Conclusions	5
(6)	References	6
(7)	Appendix	7
(8)	Index	8
(9)	Bibliography	9
(10)	Summary	10
(11)	Notes	11
(12)	Tables	12
(13)	Figures	13
(14)	Plots	14
(15)	Diagrams	15
(16)	Equations	16
(17)	Formulas	17
(18)	Derivations	18
(19)	Proofs	19
(20)	Conclusions	20
(21)	References	21
(22)	Appendix	22
(23)	Index	23
(24)	Bibliography	24
(25)	Summary	25
(26)	Notes	26
(27)	Tables	27
(28)	Figures	28
(29)	Plots	29
(30)	Diagrams	30
(31)	Equations	31
(32)	Formulas	32
(33)	Derivations	33
(34)	Proofs	34
(35)	Conclusions	35
(36)	References	36
(37)	Appendix	37
(38)	Index	38
(39)	Bibliography	39
(40)	Summary	40
(41)	Notes	41
(42)	Tables	42
(43)	Figures	43
(44)	Plots	44
(45)	Diagrams	45
(46)	Equations	46
(47)	Formulas	47
(48)	Derivations	48
(49)	Proofs	49
(50)	Conclusions	50
(51)	References	51
(52)	Appendix	52
(53)	Index	53
(54)	Bibliography	54
(55)	Summary	55
(56)	Notes	56
(57)	Tables	57
(58)	Figures	58
(59)	Plots	59
(60)	Diagrams	60
(61)	Equations	61
(62)	Formulas	62
(63)	Derivations	63
(64)	Proofs	64
(65)	Conclusions	65
(66)	References	66
(67)	Appendix	67
(68)	Index	68
(69)	Bibliography	69
(70)	Summary	70
(71)	Notes	71
(72)	Tables	72
(73)	Figures	73
(74)	Plots	74
(75)	Diagrams	75
(76)	Equations	76
(77)	Formulas	77
(78)	Derivations	78
(79)	Proofs	79
(80)	Conclusions	80
(81)	References	81
(82)	Appendix	82
(83)	Index	83
(84)	Bibliography	84
(85)	Summary	85
(86)	Notes	86
(87)	Tables	87
(88)	Figures	88
(89)	Plots	89
(90)	Diagrams	90
(91)	Equations	91
(92)	Formulas	92
(93)	Derivations	93
(94)	Proofs	94
(95)	Conclusions	95
(96)	References	96
(97)	Appendix	97
(98)	Index	98
(99)	Bibliography	99
(100)	Summary	100

List of Figures

1.1	General Sequential Circuit	5
1.2	Symbolic Specification of Logic: An Example	9
1.3	State Transition Tables and Graphs	13
2.1	Edge Triggered and Transparent Latches	20
2.2	Sequential Circuits and Communication Graphs: An Example	24
2.3	Retiming: An Example	25
2.4	Legal Retiming: An Example	26
2.5	Example: Use of a Negative Latch	27
2.6	Peripheral Retiming	29
2.7	Peripheral Retiming: Example	29
2.8	Circuit with no peripheral retiming : Example 1	31
2.9	Circuit with no peripheral retiming : Example 2	32
2.10	Path weight matrix: Example	33
2.11	Computing the Path Weight Matrix	39
2.12	Operators “+” and “&”	40
2.13	Computing the Path Weight Matrix: Example	41
2.14	Solving the Path Weight Matrix	43
2.15	Introducing Pseudo-dependencies with Negative Path Weight	47
2.16	Pipelined Circuits and their Retiming	48
2.17	Acyclic Circuit with no Peripheral Retiming	50
2.18	Handling Cyclic Circuits	51
2.19	Example FSM Optimization	53
2.20	The Equivalent State Problem: An Example	55
3.1	Impact on Transition Behavior	61
3.2	Obtaining Equivalent FSM Implementations	62
3.3	2-way split and merge	63
3.4	Switch	64
3.5	Switch using 2-way merge and split	64
3.6	Labelled Cycle of Equivalent States	65
3.7	Non-CP Transformations	66
3.8	Obtaining Equivalent FSM Implementations	67

3.9	Peripheral Retiming of Pipelined Circuits	70
3.10	Blocked Latch Motion	73
4.1	Growing Pipelined Circuits from a Seed	80
4.2	Adding node to included	82
4.3	Adding an Edge with $w \neq 0$ to included	82
4.4	Clustering Combinational Logic Blocks: I	85
4.5	Clustering Combinational Logic Blocks: II	86
4.6	Summary of the Experimental Procedure	89
4.7	Register Outputs Form a High Fanout Cutset	90
4.8	Example Circuit: <i>add_comp</i>	92
4.9	Experimental Results for <i>add_comp</i>	94
4.10	Example From a Datapath	95
5.1	Representing Circuits as MV-networks	105
6.1	Algorithm <i>sa_encoding</i>	138
6.2	<i>keyb</i> : Initial Circuit	143
6.3	<i>keyb</i> : Circuit after Node Simplification	144
6.4	<i>keyb</i> : Circuit after Algebraic Decomposition	145
6.5	<i>keyb</i> : Circuit after Encoding	147

List of Tables

4.1	Experimental Results: Performance Optimization of Pipelined Circuits . . .	97
6.1	Input Encoding Comparison	141

TABLE I

TABLE I. The results of the calculations of the α and β components of the

TABLE I. The results of the calculations of the α and β components of the

Chapter 1

Introduction

*Imperious Prima flashes forth
Her edict "to begin it":
In gentler tones Secunda hopes
"There will be nonsense in it."*

– Lewis Carroll, "Alice in Wonderland"

This thesis examines the problem of automatically synthesizing digital logic circuits. In particular, logic circuits with memory elements are considered; i.e. circuits that exhibit sequential behavior. This introductory chapter is organized as follows. First, the role of logic synthesis in the design of VLSI (Very Large Scale Integration) circuits is explained. Next, the problem domain of sequential logic synthesis is introduced. A classification of problems in this area is presented and previous work done for these problems is described. Then the scope of this thesis is defined with respect to these problems. This chapter concludes with the organization of the rest of the thesis.

1.1 VLSI Design and Logic Synthesis

The design of a digital logic system goes through several stages. The typical design flow is as follows:

Design Specification The desired behavior of the system is specified at some level of abstraction. The exact level of detail may vary depending on the designers and the specific system being designed.

Design Partition Typically systems being designed are sufficiently complex to merit being broken up into smaller sub-systems in order to make the design task more tractable.

Logic Design Here the sub-system specifications are given structure. They are converted to interconnected logic elements such as gates, logic modules (e.g. adders) and memory elements. Part of the design specification typically includes some constraints that the final design must meet, such as chip area and delay through the logic. Since the design may not be ready for physical layout, the area and delay are approximated at this level.

Physical Design After the individual components and their interconnections for each integrated circuit (IC) in the system have been specified, they have to be mapped to a physical layout that specifies the individual transistors and their interconnection. In addition, all the IC's in the system need to be placed and interconnected on a collection of printed circuit boards.

Several iterations through one or more of these stages may be needed before the design meets its specification.

While the above design flow has remained largely unchanged, integrated circuits have seen a rapid increase in complexity over the last two decades. Crossing the 100,000 mark in the number of transistors per IC (or chip) marked the entry into the era of very large scale integration (VLSI). Since then it has been possible to design and manufacture IC's with a few million transistors. The complexity presented in the design of circuits involving such a large number of components cannot be managed without computing aids of some sort. As a result a wide variety of Computer Aided Design (CAD) tools have been developed for helping designers with various aspects of the design. These tools perform two kinds of tasks:

1. **Routine Tasks:** Several tasks in the design process are routine (e.g. design rule checking). These can be easily and efficiently automated making the task at hand faster and less error prone.
2. **Searching Large Design Spaces:** The large number of components result in a combinatorial explosion when we consider the possibilities at several stages in the design process. Physical placement and routing is a classic example of this. Design tools per-

form an efficient search of the solution space which would not be possible for human designers.

In either case, these tools result in vastly reduced design time. This translates into cheaper design costs as well as faster time to market a product. These advantages have made CAD tools an integral part of VLSI design. The following areas have been addressed by design tools. They are stated here in rough chronological order of development.

Verification At several stages during the design, parts of the circuit need to be checked to assure that they meet the timing and/or logical requirements for the design. Traditionally, this has been done by simulating the circuit. Logic and timing simulators have been developed that are capable of handling significantly complex circuits. In recent years, formal verification techniques are gradually replacing simulation for logic verification.

Physical Design The circuit designer's view of the integrated circuit is a geometric view of overlapping polygons representing transistors and their interconnects. Physical design tools enable design of gates comprising of several transistors, modules consisting of several gates, the placement of these modules and gates in a two dimensional plane and the routing of interconnections between these for each IC, and placement and routing of IC's on printed circuit boards.

Design Management and Tool Integration Efficient management of the large amount of data needed to store the different parts of the design in its various stages is a formidable task. Design management tools handle the large databases needed for this purpose. These databases also permit various tools working on a design to communicate with each other since all of them now interface with the same database.

Synthesis The logic design phase was the last to see some degree of automation. This is an indication of its inherent difficulty and complexity. Currently, this is also the most time consuming part of the design process. Synthesis efforts can be classified into two categories based on their starting and ending points.

Behavioral Synthesis A description of the input-output behavior of the system is converted to structure in the form of interconnected blocks of combinational logic and memory elements. The blocks of combinational logic may have some known

functionality, e.g. a 16-bit adder, or may be specified as logic equations. It has been difficult for design tools to achieve design quality comparable to that attainable by human designers. As such these design tools are mainly of research interest and this phase is still dependent on the skill of designers. (In [53] a review of this subject is presented.)

Logic Synthesis The translation of memory elements and combinational logic blocks (described as equations) into a set of interconnected primitive elements such as gates and latches is termed logic synthesis. These primitive elements may be part of a pre-designed library that is used in conjunction with a particular design style (such as standard cell, sea of gates etc.). Since logic synthesis tools must produce results that are comparable with those produced by human designers, design optimization is a very important part of any tool. The metrics used to evaluate the result are the size of the resulting circuit (which will impact the final area), its delay (which determines the throughput or performance) and its testability. These metrics will be examined in Section 1.2.1.

1.2 Sequential Logic Synthesis

The work presented in this thesis is concerned with design aids for synthesis. In particular logic synthesis involving memory elements is considered, which is termed *sequential logic synthesis*. Figure 1.1 shows a general schematic of a sequential digital logic circuit. This has two parts; combinational logic and memory elements. The blocks of *combinational logic* each compute an arbitrary Boolean logic function. Each block consists of logic gates, such as AND, OR, NOT, connected to implement this function. Typically the interconnection of these gates within a combinational logic block is acyclic. The memory elements are used to store data between successive computations of the logic blocks. Thus their introduction results in the circuit storing the past history of inputs. As a result they are thought of as exhibiting sequential behavior, i.e., the circuit operates on input sequences and produces output sequences. This is in contrast to a combinational logic circuit, which has no memory and therefore produces a single output for a single input. The memory elements are referred to as *latches* since they latch in the data present at their inputs. (A note on the drawing conventions used in the sequel. Combinational logic is drawn using conventional gate symbols or shaded ovals. Latches are depicted by rectangles.) This research focuses on

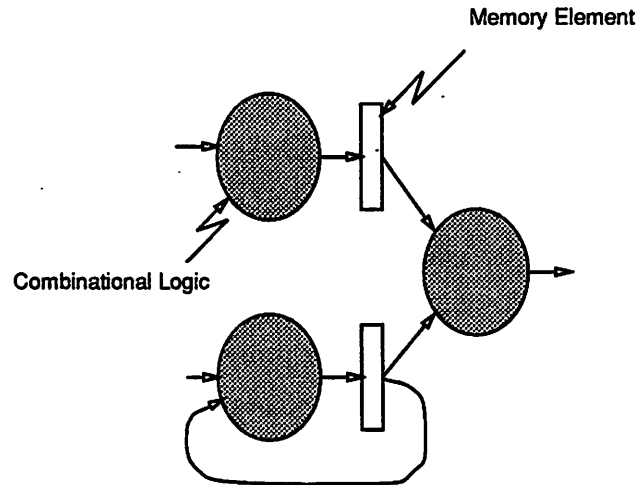


Figure 1.1: General Sequential Circuit

a special class of digital circuits, *viz.* synchronous digital circuits. These circuits have the property that all memory elements latch in their data synchronously with respect to a clock signal that is common to the circuit. A large percentage of all circuits designed fall into this category. Most existing research efforts in this area may be classified as *combinational logic synthesis* since they deal with only the combinational parts of the design. The memory elements are considered as if sacred and are not modified after their initial introduction. This has traditionally been the case because combinational logic optimization has been well-studied in the past and those results can be exploited here. Very little work that exploits the ability to modify the memory elements has been done. The techniques proposed in this thesis go beyond combinational logic optimization inasmuch as they consider altering the combinational parts as well as the memory elements during design optimization.

1.2.1 Problem Classification

This section examines the various problems that arise in design optimization during sequential logic synthesis. These may be classified along three orthogonal axes.

1: Optimization Criterion

Traditionally, the most important optimization criterion during IC design has been minimizing the size of the resulting circuit. The size is measured in terms of the area occupied by the layout. The reason for this pre-occupation with minimum area circuits has been the correlation between the area and the cost of the final circuit. In IC manufacture, the *yield* (or the percentage of non-faulty components) is an exponentially decreasing function of the size of the IC [54]. Thus, larger circuits tend to have smaller yields which results in higher cost per working component. However, in recent years the increasing maturity in IC fabrication has resulted in more stable processes which have increased the size of circuits capable of being manufactured with acceptable yields. As a result, area optimization has become less important. However, to say that area optimization is not important any longer is inaccurate. With increasing silicon real estate being available, the demands for it have also increased. Designers are putting more and more functionality on a single chip, and would like to have additional area available for adding resources, such as on-chip memory.

There has been an increasing demand for higher performance from circuits in the past decade. This arises due to higher computational needs for complex computations as well as increased volume of information being processed. In the context of synchronous sequential circuits, the performance of a circuit is measured in terms of the cycle time of its system clock. This determines the throughput of the circuit. If a synthesis tool does not address the issue of meeting the performance constraint on a design, then it is counter-productive to the very use of synthesis. Designers would (and do) spend a significant amount of time correcting the output of synthesis tools in order to meet the timing constraints. Currently, for most applications it seems that performance is the paramount optimization metric.

Once a circuit is manufactured, it needs to be tested to ensure that it has no manufacturing defects. These tests are input stimuli that distinguish between good and faulty circuits. Traditionally, testing has been considered to be a post design activity, i.e. tests are determined only after the circuit design is complete. Recent research has shown that testability considerations can be included as part of the design process. The resulting circuits have higher coverage of potential faults as well as shorter test sequences in comparison with those designed without these considerations. The final design is evaluated in terms of its testability, which is measured as some function of the fault coverage and

length of tests. Thus, in addition to area and performance, testability has emerged as an important optimization criterion.

It should be pointed out that in a typical design scenario the desired metric is some combination of performance, area and testability. The testability requirements are specified in terms of the minimum acceptable coverage under some fault model. Typically the performance constraints come from the system specification and it does not pay to do any better than what is required. The optimization goal then is to minimize the area, maximize the testability (providing at least the minimum acceptable coverage) and meet the specified performance constraint.

A final caveat: The word optimization is used in logic synthesis much in the same way as it is in the context of optimizing compilers than in the strict mathematical sense, i.e., the quality of the output is improved with respect to some metric as opposed to finding some global maximum or minimum.

2: Input Specification

Inputs may be specified in several different ways to a logic synthesis system. The most common way is to specify a set of Boolean equations that describe the combinational logic blocks and the memory elements connecting the logic blocks. Equivalently, the combinational logic blocks may be specified as an interconnection of logic gates. These two specifications are considered equivalent since there always exists a trivial mapping that converts one to the other. This form of the specification is referred to as a *Boolean specification*. The specification is said to be *mapped* if the gates and memory elements in the specification refer to specific members of some cell library. In general, mapped specifications are outputs of synthesis systems; however they may also be inputs. An instance of the mapped specification occurs in technology re-mapping of a design. Here, an already existing design needs to be re-implemented in a new technology. Logic synthesis may be used to improve the absolute quality of the previous design or to modify the implementation to exploit/suit the new technology.

The system being described typically captures some real life situation. In addition to variables that have binary or Boolean values, the specification may include symbolic variables that represent the real-life variables. As an example, Figure 1.2 has a description

of the classic Mead Conway traffic light controller [55] in the BDS language taken from [69]¹. The actual syntax and description is not significant here, what should be noted is the use of non-binary-valued variables. Here the state variable representing the state of the traffic lights is represented in symbolic form and can take on four possible values. Similarly, the output variables representing the highway and farm lights can take on three values. Since the logic is explicitly specified, this description is considered to be at the logic level even though it involves symbolic variables. This form of specification is referred to as a *symbolic specification*.

Since signals in a digital circuit can have only binary values² the symbolic variables need to be *encoded* using binary-valued variables. The process of encoding replaces a symbolic variable with a set of binary-valued variables known as encoding variables. Each value of the symbolic variable is mapped to some binary pattern of the encoding variables under the encoding. For example, in the case of the traffic light controller, the four state values HG, HY, FG, FY may be represented as the bit patterns 00, 01, 10, 11 on two binary-valued encoding variables. The resulting Boolean logic depends on the choice of encoding. Thus, the area, performance and testability of the circuit may depend on the choice of encoding. This gives rise to the *encoding problem* in logic synthesis wherein an encoding needs to be determined for a symbolic variable such that the resulting logic is optimal under some metric. The versions of the problem where the symbolic variables are inputs or outputs of the combinational logic are referred to as the input and output encoding problems respectively. When the symbolic variable is the state of a finite state machine, then this variable is both an input as well as an output of the finite state machine combinational logic. The additional constraint on the encoding is that the same encoding be selected both for the input as well as the output variable. In this case the encoding problem is referred to as the input-output encoding, or the state assignment problem. This taxonomy was first introduced in [56].

3: Structure of Target Logic

Combinational logic is implemented as a set of interconnected logic gates. The depth of the logic circuit is the maximum number of gates along any path from an input

¹This description has been slightly modified to highlight the symbolic nature of the variables.

²While circuits using multi-valued logic have been proposed, they have not yet become a practical reality. Thus, it is fair to say that signals in digital circuits are binary-valued.

```

MODEL traffic_light
    hl, fl          ! control for highway and farm lights
    st<0>,          ! to start the interval timer
    nextState =
    c<0>,           ! indicating a car on the farm road
    ts<0>, tl<0>    ! timeout of short and long interval timers
    presentState ;
ROUTINE traffic_light_controller;
    nextState = presentState; st = 0;
    SELECT presentState FROM
        [HG]: BEGIN
            hl = GREEN;  fl = RED;
            IF c AND tl THEN BEGIN
                nextState = HY; st = 1;
            END;
        END;
        [HY]: BEGIN
            hl = YELLOW; fl = RED;
            IF ts THEN BEGIN
                nextState = FG; st = 1;
            END;
        END;
        [FG]: BEGIN
            hl = RED;    fl = GREEN;
            IF NOT c or tl THEN BEGIN
                nextState = FY; st = 1;
            END;
        END;
        [FY]: BEGIN
            hl = RED;    fl = YELLOW;
            IF ts THEN BEGIN
                nextState = HG; st = 1;
            END;
        END;
    ENDSELECT;
ENDROUTINE;
ENDMODEL;

```

Figure 1.2: Symbolic Specification of Logic: An Example

to an output of that circuit. Circuits of depth two are treated rather specially. These *two-level* circuits can be implemented easily as programmable logic arrays (PLA's) which have a very regular and compact layout. This feature made this form of logic a popular choice in the early days of CAD tools since they could do a reasonably good job of generating the mask layout automatically. In addition, work had already been done in understanding and optimizing two-level logic (e.g. [61]).

However, there exist some logic descriptions such as adders and parity trees which have no compact two-level representation. These must be implemented as circuits of depth greater than two, referred to as *multi-level logic*. For most circuits, permitting multiple levels in the logic results in smaller circuits. In addition, large PLA's tend to be slow because of long diffusion lines that need to be discharged³. Even though multi-level logic typically has more gates from an input to output than two-level logic, yet it may be faster since it does not have the problem of long diffusion lines. Finally, it is noted that two-level logic is always an option even with multi-level logic, since it is just a special case. These factors result in multi-level logic being preferred to two-level logic.

The Problem Space

The three issues described above, *viz.* optimization criterion, input specification and structure of target logic, form orthogonal axes that help define the space of problems in sequential logic synthesis. Since each of the three axes permits several possibilities, the complete problem space is their cartesian product. For example, one point in this space is the area optimization problem for two-level logic implementation with Boolean input specification.

In the next section, the previous work done in this general area is described, as well as how it relates to these problems. This will establish the open problems and provide the motivation for the work presented in this thesis.

1.2.2 Previous Work

There has been a substantial amount of work in the various problems described in Section 1.2.1. Rather than describe the previous work separately for each of the individual problems, the main approaches that have been used are examined, and it is shown how

³This problem is partially solved in pre-charged PLA's with metal lines; however at the expense of more complex clocking schemes and/or additional area.

they apply for the various problems. Where the volume of literature in a particular area is extensive, only a few representative works have been cited.

Combinational Logic Optimization

Combinational logic optimization is used in sequential logic synthesis as follows. The combinational logic blocks are first separated from the memory units, optimized individually and then reconnected. The largest volume of work done in logic synthesis is in this area, perhaps because this has the largest impact on the quality of the final results.

The earliest work in combinational logic optimization can be found in the work of Quine [61] in minimizing the product terms in two-level representations of logic functions. Two-level logic optimization for minimum area has since then been very well-studied. Both exact [61, 52, 20, 64] as well as heuristic solutions [9] have been presented. The results in [63] show that exact solutions can be obtained for significantly large examples by using practically efficient algorithms. In addition, the heuristic algorithms produce results that match or are close to the exact solutions. For all practical purposes this problem is considered to be solved.

Multi-level combinational logic optimization has also gained significant maturity in the past decade. The problem here is significantly more complicated than in two-level logic since the possibilities of restructuring the logic are limitless in comparison. Nonetheless, algorithms and programs that handle both area [21, 12, 8, 32, 60, 7] and performance optimization [77, 73, 57, 6, 21, 32] have been developed. However, unlike two-level minimization, the exact algorithms [62] work only on very small circuits, so it is not known how close the state of the art is to the global optimum.

Symbolic Minimization and Encoding

The state assignment problem has been well-studied since the 60's [39, 37, 1]. However, the first attempt to relate the problem with the final logic implementation was made by DeMicheli *et al.* [59] in 1984 for two-level logic. Actually, the solution presented there was the input encoding approximation to the state assignment problem. This was then generalized in [56] to include output constraints for two-level logic. The general paradigm followed there was to first perform a symbolic minimization of the logic description, and then use this to generate constraints that the encoding must satisfy. The approach used

in [79] is similar. Both of these techniques are heuristic in nature. In [30] exact solutions to this problem have been provided, however these are not practical for any but the smallest of circuits.

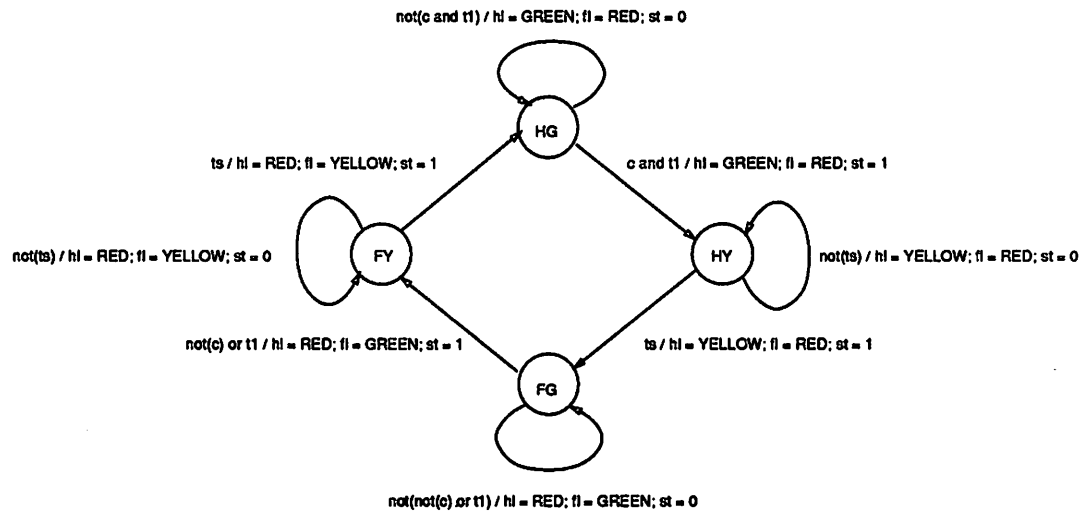
In the case of multi-level logic the approaches have not been as rigorous as those for the two-level case. Again this can be attributed to the greater flexibility permitted by multi-level logic. The approaches to the encoding problems here tend to be predictive inasmuch as they select an encoding that is likely to result in smaller logic [25, 34, 45]. Symbolic minimization for multi-level logic has not been proposed thus far.

Retiming

Retiming was introduced as a technique to improve the performance of systems at the micro-architectural level [44, 43]. This approach exploits the ability to change the positions of latches in the circuit. If we restrict ourselves to edge-triggered latches, then the cycle time of the system is the longest combinational delay between latches. Thus, it is possible to minimize the cycle time by finding positions of latches that minimize the longest combinational path between any two latches. It has been only recently that this work has been used at the logic level [58]. Retiming does not modify any combinational logic in the circuit. Thus, with respect to sequential circuit optimization the two techniques of combinational logic optimization and retiming may be viewed as duals of each other. Combinational logic optimization considers the latch positions to be fixed and modifies the combinational logic; retiming considers the combinational logic to be fixed and modifies the positions of latches.

Using State Transition Behavior

Sequential systems may be described by specifying their transition behavior in the form of state transition graphs (STG's) or equivalently as state transition tables. Figure 1.3 shows the state transition table and graph for the traffic light controller described in Figure 1.2. In the state transition table, each row describes a transition in the underlying finite state machine. The transition is from present state PS under the input vector IN to next state NS and the output produced is OUT. In the STG, the vertices represent the states. The arcs represent the transitions. The labels of the form 'input/output' on each arc represent the input vector that causes the transition and the output value produced. This is the



State Transition Graph: Example

PS	IN	NS	OUT
HG	not(c and t1)	HG	hl = GREEN; fl = RED; st = 0
HG	c and t1	HY	hl = GREEN; fl = RED; st = 1
HY	not(ts)	HY	hl = YELLOW; fl = RED; st = 0
HY	ts	FG	hl = YELLOW; fl = RED; st = 1
FG	not(not(c) or t1)	FG	hl = RED; fl = GREEN; st = 0
FG	not(c) or t1	FY	hl = RED; fl = GREEN; st = 1
FY	not(ts)	FY	hl = RED; fl = YELLOW; st = 0
FY	ts	HG	hl = RED; fl = YELLOW; st = 1

State Transition Table: Example

Figure 1.3: State Transition Tables and Graphs

same as a symbolic description of the system, since the state is represented symbolically (unencoded) at this level. It is possible to use information about the transition behavior to optimize the final circuit implementation. For example, information about equivalent states may be exploited by modifying a transition to a state s to go to an equivalent state s' [46] or by merging equivalent states into a single state as is done in state minimization. Alternatively, the finite state machine (FSM) may be decomposed into a set of interacting FSM's. There has been a lot of work done in FSM decomposition [38, 29, 2].

There is potential for exploiting more information at this level than would be possible at the gate level. For example, information about equivalent states may be very difficult to extract at the logic level. However, the general problem with this approach is that it is not possible to accurately predict the impact of modifications made at this level on the gate-level implementation. Researchers have proposed using different criteria such as the number of edges [29] or the number of states [22] in the STG as a measure of the complexity. However, none of these is a consistent reflection of the gate-level complexity. The technique presented in [2] is an exception to this. Here a reasonable estimate of the area and/or delay is included in the decomposition technique. However, this is restricted to two-level implementations.

It may seem that even if the input specification is in the form of a gate-level netlist, it may be advantageous to extract its transition behavior and use that in addition to any other techniques that can be exploited at the gate level. The problem in doing that is the combinatorial explosion involved in the extraction. Extracting the STG for any sequential circuit with more than a few latches and inputs is generally considered infeasible.

Synthesis for Testability

Research in synthesis for testability started with work that related area optimization of combinational circuits with the *single-stuck-at* fault model in testing [3]. Here it was shown that a *prime and redundant* circuit is fully testable under the single-stuck-at fault model. Later in [36] a synthesis method for fully testable circuits under the *multiple-stuck-at* fault model was developed. Synthesis techniques for fully testable sequential circuits have been presented in [27, 28, 23]. The relationship between circuit performance and testability was established in [40]. Here it was shown that testability need not be sacrificed for higher performance. The problem of delay-fault testability was tackled in [24]. Here, synthesis

techniques for robust-path and gate-delay-fault testable circuits were presented.

There are two aspects of improving the testability properties of a circuit. The first is increasing its testability; this has been described in the previous paragraph. The second is making it more easily testable. This implies reducing both the computing time it takes to derive these tests (e.g. [19]) as well as the time it takes to run these tests on the circuit (e.g. [27]).

1.3 Thesis Overview

Since multi-level logic is the most prevalent target implementation, the work presented in this thesis will focus on this form of logic implementation. An attempt is made to build on previous work done in combinational logic synthesis and see the natural extension of well-understood ideas there in the expanded context of multi-level sequential logic synthesis.

The thesis is divided into two parts. In the first part, it is demonstrated how the dual ideas of retiming and combinational logic optimization can be combined in such a way as to maximize the use of combinational logic optimization. The results here apply to both area and performance optimization. While synthesis for testability is not directly considered, the relationship between the ideas presented here and the testability of the resulting circuits is examined.

The second part presents techniques for the symbolic minimization of multi-level circuits with symbolic inputs. This has direct application in the state assignment problem for multi-level logic.

Chapter 2 describes the theoretical results developed in combining retiming and combinational logic optimization. The procedure that utilizes both these techniques is called *retiming and resynthesis*. Sequential sub-circuits for which all the latches can effectively be ignored are considered. This enables them to be considered as combinational circuits. For these circuits all the latches can be migrated to the periphery of the circuit by a procedure that is an extension of retiming. One of the main results of this chapter are the necessary and sufficient conditions on sub-circuits for which this is possible. This result enables us to use combinational logic optimization beyond latch boundaries; in fact it pushes combinational optimization to its limits in the context of sequential circuits.

In Chapter 3 the implications and applications of these ideas are described. The

relationship of retiming and resynthesis to logical testing and state assignment is examined. Then it is shown how these ideas can be applied towards the performance optimization of sequential circuits. A special class of sequential circuits is considered, *viz.* pipelined circuits. Here the equivalence between the performance optimization problem for pipelined circuits and that for combinational circuits is established.

Chapter 4 discusses the issues involved in the practical implementation of retiming and resynthesis as part of a sequential logic optimization system, SIS [70]. Specific algorithms are presented as well as the experiences with using these on some real designs.

The second part of the thesis considers the symbolic minimization of multi-level circuits with multiple-valued inputs. Chapter 5 presents extensions of the various multi-level optimization techniques used with Boolean circuits to handle multiple-valued inputs. The main contribution of this chapter is the technique for factorization of logic expressions with multiple-valued variables. This was the missing link in the multi-level optimization of circuits with multiple-valued inputs. It is then shown how the results of symbolic minimization can be used to tackle the input encoding problem. This can be used to approximate the state assignment problem; an approximation that is valid when the primary output logic dominates the next state logic.

The implementation of these issues uncovers some interesting problems. These practical issues are considered in Chapter 6 along with experimental results for the input encoding of some real designs.

Finally, Chapter 7 summarizes what has been learned from this work and considers future directions in this area.

Part I

Retiming and Resynthesis

Chapter 2

Basic Ideas

“You can draw water out of a water-well,” said the Hatter; “so I should think you could draw treacle out of a treacle-well – eh, stupid?”

– Lewis Carroll, “Alice in Wonderland”

Over the last decade combinational logic optimization has attained a significant level of maturity. (Some of the work done in this area was reviewed in Chapter 1.) The problems and approaches there are well understood: almost fully in the two-level logic case, and to a lesser extent in the multi-level logic case. However, in sequential logic optimization their utility is restricted to individual portions of combinational logic. Logical relationships are not exploited between gates that are separated by latch boundaries. What is desirable is the ability to use the ideas in combinational logic optimization beyond latch boundaries. In this direction we would like to push combinational logic optimization to its limits, i.e., capture all the logical relationships that exist between gates in a sequential circuit even though they may not belong to the same block of combinational logic. This thought direction is a very natural one; it stems from the desire to build on what is already known and forms the motivation behind the work presented in this part of the thesis. First the application domain of these ideas is specified as a class of sequential circuits with specific clocking methodologies. Then the suggested approach is described, which is termed *retiming and resynthesis* since it combines retiming with resynthesis of combinational logic. (Most of the work presented in this chapter was first reported in [49]).

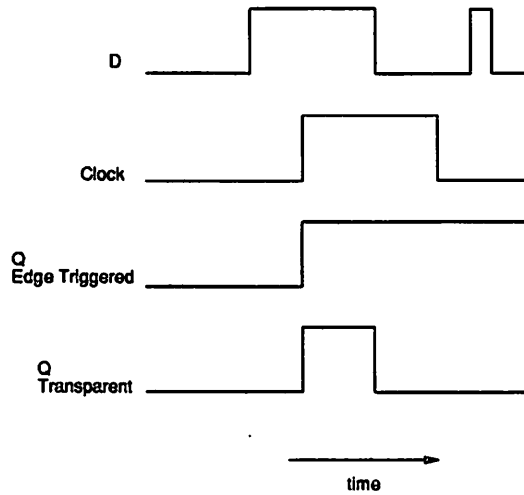


Figure 2.1: Edge Triggered and Transparent Latches

2.1 Clocking Methodology

As mentioned in Chapter 1, only synchronous sequential circuits are considered, i.e., all memory elements latch their data synchronously with respect to a clock signal common to the circuit. Within synchronous circuits there is flexibility as to when the data is latched with respect to the clock edge. This is illustrated in Figure 2.1. Let D be the input and Q the output of the latch. In edge-triggered latches the data present at the clock edge (the rising edge in this case) is latched and available at Q . In transparent or level-sensitive latches, the latch is transparent during the time the clock is high, i.e. the data input is available at the output. The data value at the end of the clock high period is latched.

This work is restricted to an edge-triggered clocking methodology. Thus, the word “latch” in the following exposition refers to an edge-triggered latch or a flip-flop. It may seem that this restriction is a fairly strong one given the designers’ wisdom that transparent latches result in higher performance. This fact is illustrated through examples in [65], where it is shown that by appropriately selecting the clocking parameters it may be possible to clock a synchronous system with transparent latches significantly faster than one with edge-

triggered ones. While that may sometimes be the case, a large fraction of ASIC (Application Specific Integrated Circuit) designs are done using edge-triggered latches [5]. The principal reason for this is the ease of analysis of circuits with edge-triggered latches. There are no complex “time borrowing” scenarios and designers do not have to worry about “short paths”. Since ASIC’s are the application domain where synthesis is the predominant design methodology, this does not seem too restrictive.

Retiming algorithms have this restriction ¹ and since this work uses retiming, it is inherited. Current developments in exploiting retiming techniques with transparent latches [71] could enable retiming and resynthesis to be applied to circuits with transparent latches.

2.2 Overview

Sub-circuits in a sequential circuit are characterized for which the latches can effectively be ignored and thus the sub-circuit can be considered as a combinational block. This permits existing combinational logic optimization techniques to be used on it. This approach is more powerful than combinational logic optimization, since it examines interactions between portions of logic separated by latches. As a result, the optimization process makes full use of dependencies between gates. Moreover, it is guaranteed that it is complete, i.e., the largest sub-circuit for which this can be done is determined. This ensures that no optimization that can be obtained by considering interactions between gates is missed. Converting this sub-circuit to a combinational logic block can be viewed as a retiming process in which all the latches are pushed to the periphery of the sub-network. However, this technique is more powerful than conventional retiming in that it permits *negative latches* to be pushed to the periphery. This is equivalent to temporarily “borrowing” latches from the environment, and is a legitimate operation as long as these latches are “returned” to the environment at the end of the optimization process. This additional allowance is more powerful since it permits a larger portion of the logic to be viewed as a single block than is permitted by conventional latch movements using retiming. Next, this combinational logic block may be resynthesized according to a specified cost function. This could be minimizing the area, the delay or meeting a particular area/delay tradeoff. Conditions are specified

¹Retiming algorithms need this restriction in order to compute the cycle time of the circuit in polynomial time.

for the legal redistribution of latches in this circuit, i.e., conditions under which the latches borrowed from the environment can be returned. The redistribution can be done while satisfying constraints such as minimizing the number of latches subject to a specified cycle time (if these constraints are satisfiable) by using the algorithms described in [43]. Since the optimization algorithms work directly on the gate-level netlist, they use the gate-level complexity as their cost function, unlike algorithms that work on state transition graphs.

2.3 Theoretical Formulation

Let us first focus our attention on sequential circuits whose underlying topology is acyclic. (These are also referred to as *feed-forward circuits*.) These circuits are modeled by a directed acyclic graph called a *communication graph*² where each vertex v represents either

- a) an input/output pin or
- b) a combinational logic block.

The input/output pins correspond to the primary inputs and primary outputs of the circuit. The granularity of the combinational logic block may vary: it may be a single gate or a larger module such as an adder. The vertices in the graph are connected by directed edges. A restriction is placed that each input pin has no incoming edges and exactly one outgoing edge (a single-output source), and that an output pin has no outgoing edges and exactly one incoming edge (a single-input sink). If a primary input is used in more than one place in the circuit then this is captured by introducing a dummy vertex in the graph that handles the multiple out-edges. (The out-edges are also referred to as fanout.) An *internal edge* connects vertex u to vertex v if both u and v represent combinational logic blocks, and the logic represented by v explicitly depends on the value computed at u . A *peripheral edge* connects either an input pin to the logic block that uses that input or connects a logic block that computes the value of an output to the corresponding output pin. Each edge e has a corresponding weight $w(e)$ representing the number of latches between the two vertices it connects. An example of a sequential circuit and its communication graph is shown in Figure 2.2. Note that the multiple fanout of primary input a is handled through the internal vertex a' in the graph. For simplicity in the figure, if the edge weight is 0 then

²This is related to the definition of a communication graph presented in [44].

the edge label is omitted. A sequential circuit is alternatively referred to as a sequential network. The terms circuit, network, and graph are used interchangeably whenever there is no ambiguity.

A *path* between two vertices v_1 and v_2 in the graph is a sequence of consecutive edges from v_1 to v_2 . The weight of a path is the sum of the weights of all the edges along the path. In Figure 2.2, the path from input b to output f has weight 0, while the path from b to e has weight 1.

2.3.1 Retiming: An Overview

The cycle time of a synchronous sequential circuit is determined by the length of the longest path between any two latches in the circuit. The concept of *retiming* exploits the ability to move the latches in the circuit in order to decrease the length of the longest path in the circuit while preserving its functional behavior. Retiming algorithms were first proposed by Leiserson *et al.* [44, 43]. To illustrate this with a small example, consider Figure 2.3. The circuit on the left is functionally equivalent to the circuit on the right since delaying the output of gate g by a cycle is equivalent to delaying each of its inputs by a cycle. The movement of latches during retiming is quantified by an integer $L(v)$ (called the lag of v) for each vertex v , which represents the number of latches that are to be moved in the circuit from each out-edge of vertex v to each of its in-edges. Thus, in Figure 2.3 the circuit on the right is obtained from the circuit on the left by retiming g by $+1$. Similarly, in obtaining the circuit on the left from that on the right g has been retimed by -1 . For input and output pins the lag is 0. Consider an edge $e(u, v)$ in the circuit. Let $w(e)$ be the weight of the edge in the graph before retiming and $w_r(e)$ be the weight after retiming. w_r is determined from w and the lags by the following equation:

$$w_r(e) = w(e) + L(v) - L(u)$$

This is used to prove a simple result that will be used frequently in the sequel.

Lemma 2.3.1 (Leiserson and Saxe) *Let p be a path between input i and output j . Let $W(p)$ be the weight of this path. Let r be a retiming and $W_r(p)$ be the weight of this path after retiming. Then, $W(p) = W_r(p)$.*

Proof.

$$W(p) = \sum_{\text{path } i_i \rightarrow o_j} w(e)$$

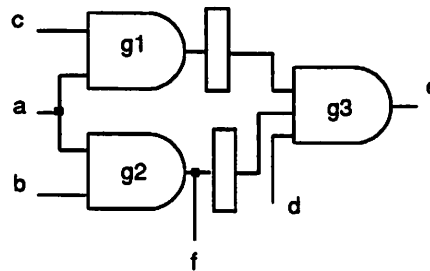
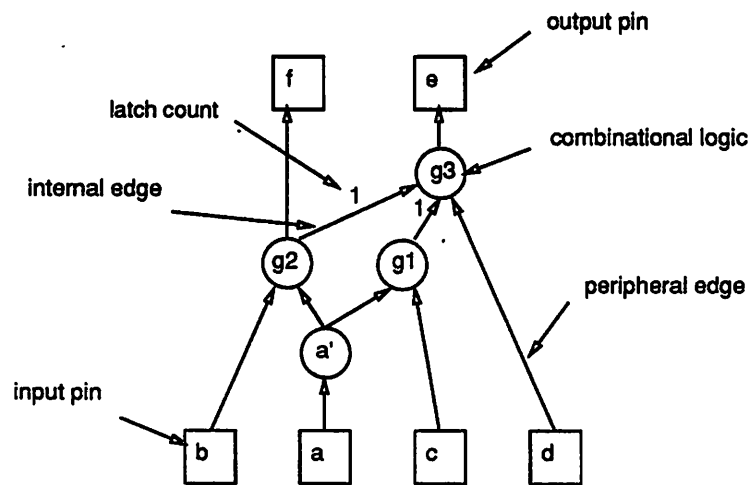
**Sequential Circuit****Communication Graph**

Figure 2.2: Sequential Circuits and Communication Graphs: An Example

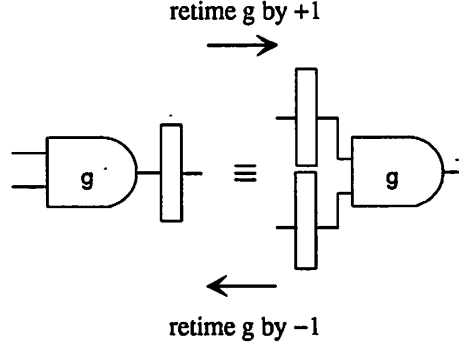


Figure 2.3: Retiming: An Example

$$W_r(p) = \sum_{\text{path } i_i \rightarrow o_j} w_r(e) = \sum_{\text{path } i_i \rightarrow o_j} (w(e) + L(v) - L(u))$$

This sum telescopes:

$$W_r(p) = \sum_{\text{path } i_i \rightarrow o_j} w(e) + L(o_j) - L(i_i)$$

Since $L(o_j) = L(i_i) = 0$,

$$W_r(p) = \sum_{\text{path } i_i \rightarrow o_j} w(e) = W(p)$$

■

The following definition is from [44].

Definition 2.3.1 A legal retiming is the assignment of an integer $L(v)$ to each vertex in the communication graph such that for each edge e , $w_r(e) \geq 0$.

For a legal retiming, the edge weights of the retimed graph must be non-negative; indicating a non-negative number of latches on each edge. Thus, there exists a real physical circuit corresponding to this graph. This is not possible with negative edge weights in the retimed circuit since there is no physical circuit component corresponding to a negative latch. A legal retiming has been shown [44] to generate a circuit that is functionally equivalent to the original circuit. Figure 2.4 shows a legal retiming on the communication graph of Figure 2.2. Here, the lag of $g1$ is $+1$ and the lag for all other vertices is 0 .

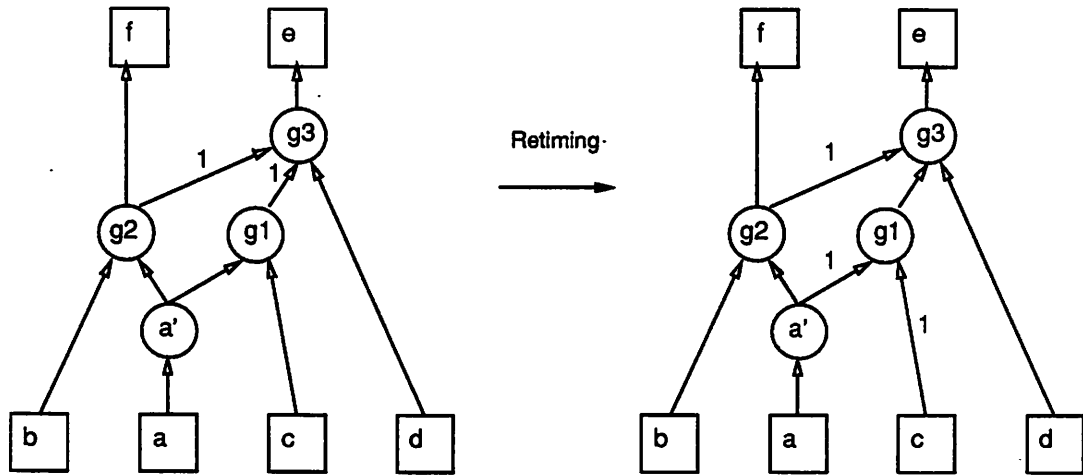


Figure 2.4: Legal Retiming: An Example

2.3.2 Extensions to Retiming

It is now shown how retiming can be extended by introducing the concept of a “negative” latch, i.e., an edge weight in the graph that is negative. Negative edge weights are permitted on peripheral edges only. Allowing a negative edge weight n on a peripheral edge is equivalent to “borrowing” n latches from the environment. The latches may be “returned” by a subsequent retiming step, whereby n latches are forced to each edge with weight $-n$. The observation that the peripheral edge weights can temporarily take on negative values allows retiming operations and subsequent optimizations that would otherwise not be possible. This is illustrated with the circuit in Figure 2.5(a). Consider the latch on the connection between $g2$ and $g3$. In order to move this latch from its present position either $g3$ is retimed by -1 (for forward motion) or $g2$ retimed by $+1$ (backward motion). If $g3$ is retimed by -1 , this will result in an edge weight of -1 at input d . If $g2$ is retimed by $+1$ this will result in an edge weight of -1 at output f . Thus, neither of these retimings is legal. However, if this “illegal” retiming were permitted temporarily, then it is possible to gain additional advantage over what is permitted by just legal retimings. Figure 2.5(b) shows the circuit after $g3$ has been retimed by -1 . The edge weight of -1 on input d is

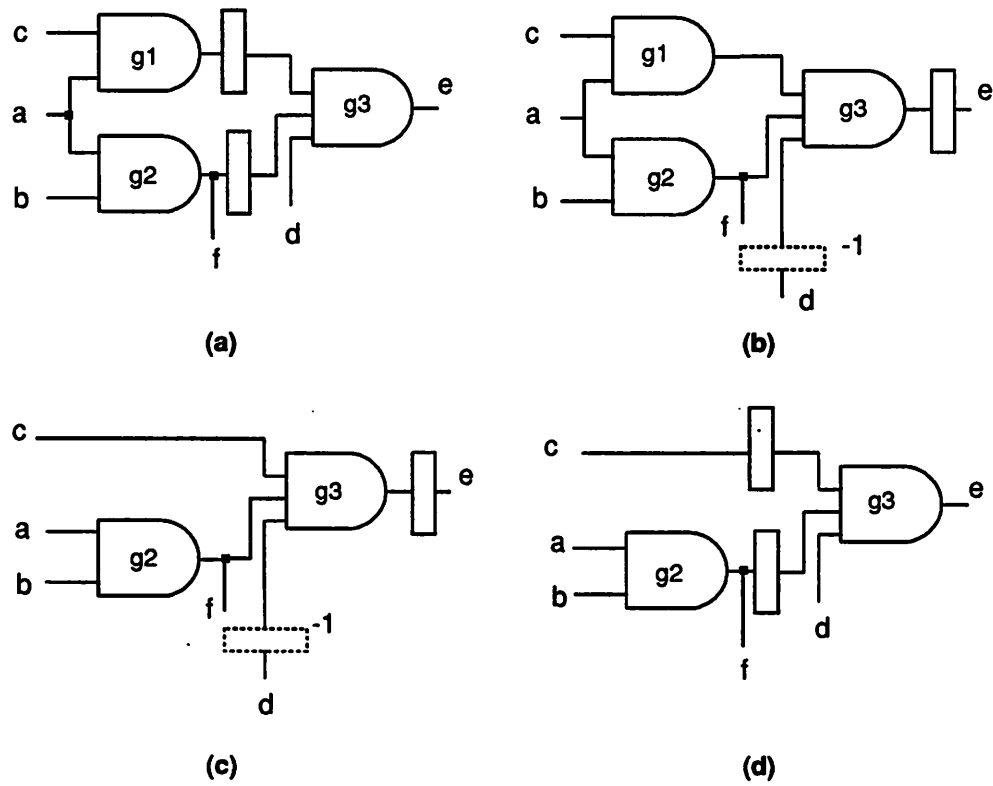


Figure 2.5: Example: Use of a Negative Latch

represented by the latch in dotted lines and with the label -1 on it. At this point, all the gates in the circuit are part of a single combinational logic block. For this logic block, it can be shown that the functionality remains unchanged if the connection from a to $g1$ is deleted, i.e., this connection is redundant or in testing parlance, untestable for a stuck-at-1 fault in the context of this combinational logic block. Thus, this connection may be deleted and $g1$ replaced by a wire. This simplified circuit is shown in Figure 2.5(c). Note that this connection (from a to $g1$) is not redundant in the context of the original combinational block (consisting of $g1$ and $g2$) defined by the original position of latches. Only after we could view all the gates as part of a single combinational logic block was this redundancy exposed. Of course, this circuit still is not realizable since there is a negative latch at input d . This situation can easily be rectified by retiming $g3$ by $+1$. This annihilates the negative latch at input d resulting in the circuit in Figure 2.5(d). This example illustrates the advantage gained by permitting illegal retimings temporarily. Later in this section it is shown why this is a legitimate operation.

Let us now go back and see what enabled us to eliminate $g1$ in the previous example. Once we were able to consider all three gates in the circuit as part of a larger combinational block, we could use the combinational optimization technique of redundancy removal to detect and delete the redundant connection. This is precisely what we were looking for, i.e., a way to consider and exploit logical relationships between gates that extend beyond latch boundaries. Ideally, we would like to push out all the latches in the circuit to the peripheral edges. This results in no latches on any of the internal edges and thus all the gates are part of the same combinational logic block. This permits the use of any combinational logic optimization technique on this larger combinational block. The notion of a *peripheral retiming* does precisely this.

Definition 2.3.2 A peripheral retiming is a retiming such that for each internal edge e , $w_r(e) = 0$.

This is graphically shown in Figure 2.6. After peripheral retiming, there are α_i latches at input pin i , β_j latches at output pin j , and no latches on any internal edge.

The circuit in Figure 2.5(b) is a peripheral retiming of the one in Figure 2.5(a). The same peripheral retiming is shown in terms of the communication graph in Figure 2.7.

The condition that $w_r(e)$ is 0 for all internal edges forces all latches to the pe-

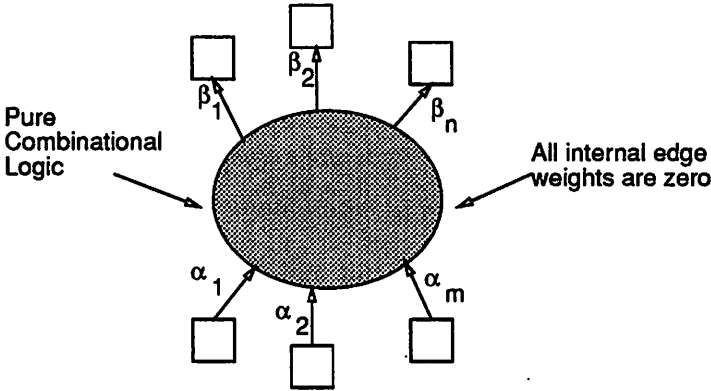


Figure 2.6: Peripheral Retiming

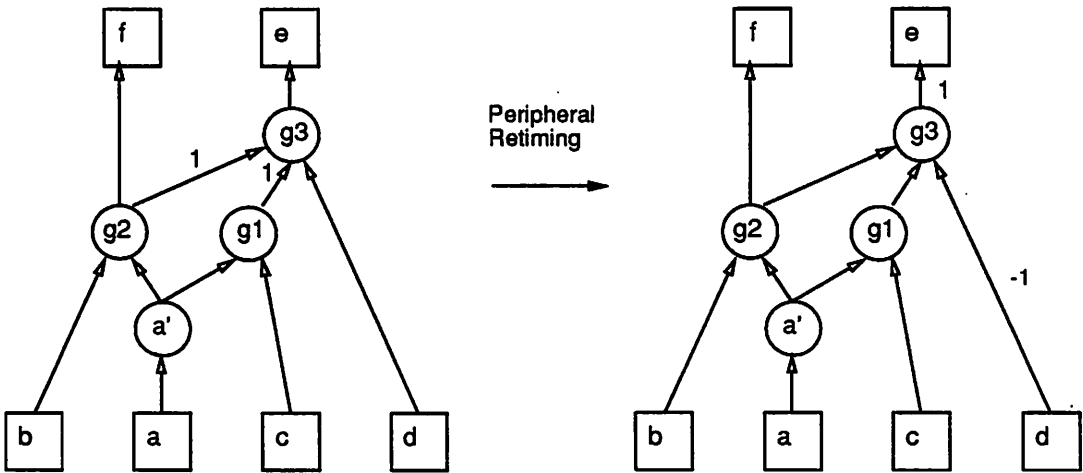


Figure 2.7: Peripheral Retiming: Example

peripheral edges. Note that the definition permits negative weights on the peripheral edges, which corresponds to the negative latch concept presented earlier. Permitting negative latches temporarily on peripheral edges is a legitimate operation as shown by the following theorem. Functional equivalence here refers to the equivalence of the finite automata corresponding to the initial and final circuits.

Theorem 2.3.1 *A circuit that undergoes a peripheral retiming, combinational optimization and a subsequent legal retiming is functionally equivalent to the original circuit.*

Proof. Let C_1 be the original circuit and C_2 be the peripherally-retimed circuit obtained with retiming r . Let α_i and β_j be the number of latches at the i^{th} input and j^{th} output pin in C_2 . Let C_3 be the circuit obtained after combinational resynthesis, \mathcal{R} , on the interior combinational logic and let C_4 be the circuit obtained after a legal retiming l on C_3 .

Let $\alpha_{min} = |\min(0, \alpha_i)|$ over all α_i and let $\beta_{min} = |\min(0, \beta_j)|$ over all β_j . Consider the circuit C_5 obtained from C_1 by adding α_{min} latches at each input pin and β_{min} latches at each output pin. $C_5 \equiv \text{delay}(C_1, \alpha_{min} + \beta_{min})$, i.e. given an input-output vector sequence $(\mathcal{I}, \mathcal{O})$ for C_1 , the input sequence \mathcal{I} results in the output sequence \mathcal{O} delayed by $\alpha_{min} + \beta_{min}$ cycles in C_5 . Let C_6 be the circuit obtained by retiming C_5 with r . This is a peripheral retiming of C_5 with $\alpha'_i = \alpha_i + \alpha_{min}$ and $\beta'_j = \beta_j + \beta_{min}$. Note that this is a legal retiming since there are no negative latches, as α'_i and β'_j are non-negative and there are no other latches in the circuit. Here recourse is taken to the results in [44] that show functional equivalence with legal retimings to claim that C_6 is equivalent to C_5 ³. Combinational resynthesis, \mathcal{R} , of the interior combinational logic of C_6 , resulting in the circuit C_7 , obviously does not change its functionality since none of the functions of any primary outputs or latch inputs are changed by this. Now, retiming l is applied to C_7 to result in C_8 . Note that since l was a legal retiming for C_3 it must result in at least α_{min} latches at each input pin and β_{min} latches at each output pin. Also, by transitivity, C_8 is equivalent to C_5 . Hence, $C_8 \equiv \text{delay}(C_1, \alpha_{min} + \beta_{min})$. Let C_9 be obtained from C_8 by removing α_{min} latches from each input pin and β_{min} latches from each output pin. Thus, $C_9 \equiv \text{delay}(C_1, 0)$ i.e. C_9 is equivalent to C_1 . Note that C_9 is identical to C_4 because the same resynthesis \mathcal{R} and final retiming l were applied in order to obtain them, ensuring that they have the same gate and latch netlists. Thus, C_4 is equivalent to C_1 . ■

³Functional equivalence here is subject to being able to get the two circuits in equivalent states. The problem of finding equivalent states for the original and retimed circuits has been looked at in [76] and is discussed in Section 2.6.

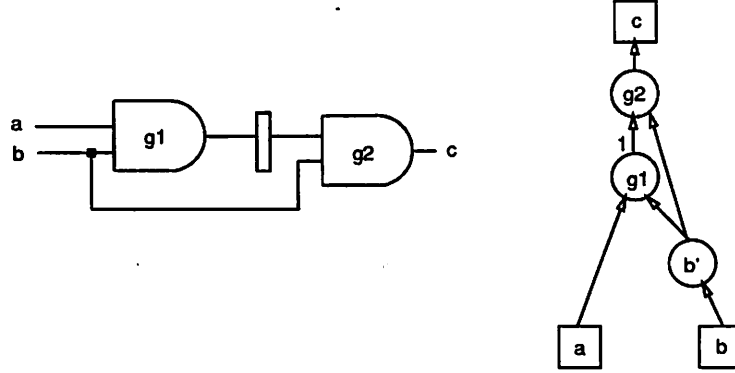


Figure 2.8: Circuit with no peripheral retiming : Example 1

2.3.3 Conditions for Peripheral Retiming

Not all circuit topologies permit a peripheral retiming. Two such topologies are now considered.

Consider the circuit in Figure 2.8 and its corresponding communication graph. All attempts to move the latch (either backward or forward) to the periphery result in a negative weight on the edge between b' and $g2$. In fact, as will be shown later in this section, this circuit cannot be peripherally retimed. Examining the circuit gives us some insight into why this is so. The output c depends on the value of input b at two different times. Let us assume that a peripheral retiming were possible. Then in the peripherally retimed circuit c would depend only on one time value of b since all paths from b to c would have the same number of latches, *viz.* $\alpha_b + \beta_c$. This would not capture the correct behavior and is the reason why no peripheral retiming exists.

Now consider the circuit and communication graph in Figure 2.9. Again, it is not possible to move the latches to the periphery without introducing a negative weight on the internal edges $(a', g2)$ and $(b', g1)$. The intuition as to why a peripheral retiming does not exist here is more complicated than in the previous case. For output c , inputs a and b are delayed by 1 and 0 cycles respectively. For output d , these inputs are delayed by 0 and 1

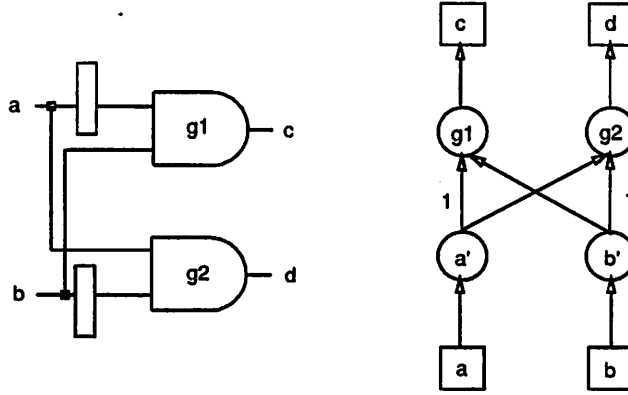


Figure 2.9: Circuit with no peripheral retiming : Example 2

cycles respectively. Let us assume a peripheral retiming were possible. Then c would see a and b delayed by $\alpha_a + \beta_c$ and $\alpha_b + \beta_c$ respectively. Similarly d would see them delayed by $\alpha_a + \beta_d$ and $\alpha_b + \beta_d$ respectively. The only difference in delays for the inputs that d sees with respect to c is due to the different number of latches on the peripheral edges at c and d , i.e. $\beta_d - \beta_c$. Thus, the input delays are the same for all the outputs except for a constant offset that depends on the output and this offset is added to each of the input delays. Clearly this is not possible for this example ⁴.

While these two examples give some insight into when a peripheral retiming may not exist, by themselves they do not provide a characterization of circuits that permit a peripheral retiming. In order to obtain such a characterization, the *path weight matrix* of a network is defined.

Definition 2.3.3 A path weight matrix, W , of a sequential network is an $m \times n$ matrix, where

- 1) m is the number of inputs
- 2) n is the number of outputs
- 3) $W_{ij} = *$ if no path exists between input i and output j

⁴Even though the circuits in Figure 2.8 and Figure 2.9 cannot be peripherally retimed, sub-circuits of these circuits can. This is demonstrated later in Section 2.5.

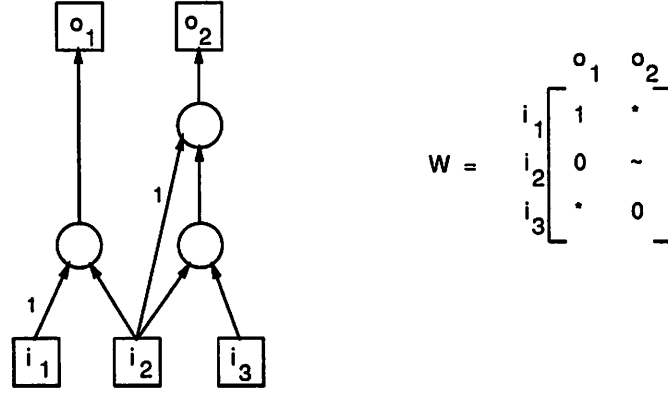


Figure 2.10: Path weight matrix: Example

4) $W_{ij} = \sim$ if two paths between input i and output j have different weights

5) $W_{ij} = \sum_{\text{path } i_i \rightarrow o_j} w(e)$ if all paths between input i and output j have the same weight.

Figure 2.10 shows a communication graph and the corresponding path weight matrix.

In addition, the satisfiability condition on the path weight matrix is defined, which is directly related to the existence of a peripheral retiming.

Definition 2.3.4 A matrix W is satisfiable if

a) $W_{ij} \neq \sim, \forall i, j$

b) $\exists \alpha_i, \exists \beta_j, 1 \leq i \leq m, 1 \leq j \leq n, \alpha_i, \beta_j \in \mathcal{I}$ such that for each $W_{ij} \neq *$, $W_{ij} = \alpha_i + \beta_j$.

These two conditions have been motivated by the two examples considered above. Cyclic circuits (assuming there is at least one latch on each path containing a cycle) will have $W_{ij} = \sim$ for each path containing a cycle. Thus, only acyclic communication graphs can have a satisfiable path weight matrix. Optimization of sequential circuits with a cyclic structure is described in Section 2.5.

The following two lemmas are used to demonstrate the relationship between a satisfiable path weight matrix and the existence of a peripheral retiming.

Lemma 2.3.2 *Let W be a satisfiable path weight matrix for a communication graph G . Let $\alpha_1, \alpha_2, \dots, \alpha_i, \dots, \alpha_m$ and $\beta_1, \beta_2, \dots, \beta_j, \dots, \beta_n$ be integers that satisfy W . Let v be a non-I/O vertex in the communication graph G and let $f_i(v) = \alpha_i - \sum_{\text{path } i_i \rightarrow v} w(e)$, where i_i is some i^{th} input pin and there is path from this pin to v . Then, $f_i(v)$ is independent of i .*

Proof.

Let i_1 and i_2 be two inputs each with a path to vertex v . Then:

$$f_1(v) = \alpha_1 - \sum_{\text{path } i_1 \rightarrow v} w(e)$$

$$f_2(v) = \alpha_2 - \sum_{\text{path } i_2 \rightarrow v} w(e)$$

Let v have a path to some output o_1 (it must have a path to some output pin). The weight along the path from v to o_1 is

$$\sum_{\text{path } v \rightarrow o_1} w(e)$$

The path from i_1 to o_1 has weight

$$\sum_{\text{path } i_1 \rightarrow v} w(e) + \sum_{\text{path } v \rightarrow o_1} w(e) = \alpha_1 + \beta_1 \quad (2.1)$$

The path from i_2 to o_1 has weight

$$\sum_{\text{path } i_2 \rightarrow v} w(e) + \sum_{\text{path } v \rightarrow o_1} w(e) = \alpha_2 + \beta_1 \quad (2.2)$$

Subtracting Equation 2.1 from Equation 2.2 yields

$$\sum_{\text{path } i_2 \rightarrow v} w(e) - \sum_{\text{path } i_1 \rightarrow v} w(e) = \alpha_2 - \alpha_1$$

Rearranging, we obtain

$$\alpha_1 - \sum_{\text{path } i_1 \rightarrow v} w(e) = \alpha_2 - \sum_{\text{path } i_2 \rightarrow v} w(e)$$

$$f_1(v) = f_2(v)$$

■

Lemma 2.3.3 *Let W be a satisfiable path weight matrix for a communication graph G . Let $\alpha_1, \alpha_2, \dots, \alpha_i, \dots, \alpha_m$ and $\beta_1, \beta_2, \dots, \beta_j, \dots, \beta_n$ be integers that satisfy it. The following lag function, $L_p(v)$, results in a peripheral retiming:*

- a) $L_p(v) = \alpha_i - \sum_{\text{path } i_i \rightarrow v} w(e)$ for each internal vertex
- b) $L_p(v) = 0$ for each I/O pin

where α_i is the α associated with the input i which has a path to vertex v , and $\sum_{\text{path } i_i \rightarrow v} w(e)$ is the weight of any path from input i to v .

Proof.

A peripheral retiming requires $w_r(e)$ to be 0 for each internal edge e . Each edge weight in the retimed circuit is:

$$w_r(e_{uv}) = w(e_{uv}) + L(v) - L(u)$$

where $w(e_{uv})$ is the weight of the edge from u to v . $L(u)$ can be expressed in terms of any α_i such that there is a path from the i^{th} input to vertex u . Any such i suffices because $L(u)$ is independent of i by Lemma 2.3.2.

$$L(u) = \alpha_i - \sum_{\text{path } i_i \rightarrow u} w(e)$$

If a path exists from input i to vertex u , then there is a path from input i to vertex v , and $L(v)$ can be expressed in terms of input i :

$$L(v) = \alpha_i - \sum_{\text{path } i_i \rightarrow v} w(e)$$

Hence,

$$\begin{aligned} w_r(e_{uv}) &= w(e_{uv}) + \alpha_i - \sum_{\text{path } i_i \rightarrow v} w(e) - \left(\alpha_i - \sum_{\text{path } i_i \rightarrow u} w(e) \right) \\ &= w(e_{uv}) - w(e_{uv}) = 0 \end{aligned}$$

For an input edge, vertex u is an input pin, so $L(u) = 0$. In addition, each input edge has the property $w(e_{uv}) = \sum_{\text{path } i_i \rightarrow v} w(e)$, yielding

$$w_r(e_{uv}) = w(e_{uv}) + \alpha_i - \sum_{\text{path } i_i \rightarrow v} w(e)$$

$$w_r(e_{uv}) = \alpha_i$$

Similarly, for the output edges of the network, v is an output pin, so $L(v) = 0$. Thus:

$$\begin{aligned} w_r(e_{uv}) &= w(e_{uv}) + L(v) - L(u) \\ &= w(e_{uv}) - \left(\alpha_i - \sum_{\text{path } i_i \rightarrow u} w(e) \right) \end{aligned}$$

For each output edge e between vertex u and I/O pin v ,

$$\sum_{\text{path } i_i \rightarrow u} w(e) = \alpha_i + \beta_j - w(e_{uv})$$

so the weight of the edge in the retimed circuit can be expressed as follows:

$$w_r(e_{uv}) = w(e_{uv}) - [\alpha_i - (\alpha_i + \beta_j - w(e_{uv}))]$$

$$w_r(e_{uv}) = \beta_j$$

Thus, the specified lag function results in the desired edge weights for the internal and peripheral edges that are required by the peripheral retiming. ■

Theorem 2.3.2 *A sequential network has a peripheral retiming if and only if its path weight matrix is satisfiable.*

Proof.

If part:

Follows directly from Lemma 2.3.3.

Only if part:

It suffices to show that a circuit with a peripheral retiming has a satisfiable path weight matrix. A circuit with a peripheral retiming has an integral (possibly negative) number of latches, α_i , at the i^{th} input and an integral (possibly negative) number of latches, β_j , at the j^{th} output, and no latches on any of the internal edges. Regardless of the path chosen, the number of latches between the i^{th} input and the j^{th} output is $\alpha_i + \beta_j$ (if a path exists) in the retimed circuit. By Lemma 2.3.1 this is the (i, j) th entry of the path weight matrix for the original circuit. Thus the path weight matrix is satisfied by these α 's and β 's. ■

Note the significance of this result: it gives a complete characterization of the class of sequential circuits for which all the latches can be pushed to the periphery, i.e., it specifies the necessary as well as the sufficient conditions on the circuit topology.

A peripheral retiming involves finding a set of α 's and β 's that satisfy the path weight matrix. These α 's and β 's specify the peripheral retiming. In the retimed circuit there are α_i latches at the i^{th} input pin and β_j latches at the j^{th} output pin. A matrix that is satisfiable has no \sim entries, and has at least one set of α_i 's and β_j 's such that $\alpha_i + \beta_j = W_{ij}$.

For the communication graph in Figure 2.7, the path weight matrix is as follows:

$$\begin{array}{cc} & f \quad e \\ b & 0 \quad 1 \\ a & 0 \quad 1 \\ c & * \quad 1 \\ d & * \quad 0 \end{array}$$

and can be satisfied by choosing, for example, $\alpha_b = 0$, $\alpha_a = 0$, $\alpha_c = 0$, $\alpha_d = -1$, $\beta_f = 0$, $\beta_e = 1$, resulting in the circuit shown on the right in that figure.

The path weight matrix for the circuit in Figure 2.8, which had no peripheral retiming, is as follows:

$$\begin{array}{cc} & c \\ a & 1 \\ b & \sim \end{array}$$

The \sim in the second row rules out the possibility of a peripheral retiming. For the circuit in Figure 2.9, the path weight matrix is as follows:

$$\begin{array}{cc} & c \quad d \\ a & 1 \quad 0 \\ b & 0 \quad 1 \end{array}$$

It is easily checked that no α_i , β_j exist by applying the conditions necessary to satisfy the matrix. This yields

$$\alpha_1 + \beta_1 = 1 \tag{2.3}$$

$$\alpha_1 + \beta_2 = 0 \tag{2.4}$$

$$\alpha_2 + \beta_1 = 0 \tag{2.5}$$

$$\alpha_2 + \beta_2 = 1 \quad (2.6)$$

Subtracting Equation 2.3 from Equation 2.4:

$$\beta_2 - \beta_1 = -1$$

Subtracting Equation 2.5 from Equation 2.6 yields

$$\beta_2 - \beta_1 = 1$$

This contradiction implies that the path weight matrix is not satisfiable.

2.3.4 Computing the Path Weight Matrix

It is now shown how the path weight matrix, W , is computed for a given communication graph. Each column of the path weight matrix contains information about the number of latches between that output (corresponding to that column) and each of the inputs. Thus, the column is a *path weight vector* for that output. The notion of a path weight vector, V , can be extended to any vertex in the communication graph. V captures information about the number of latches between that vertex and each of the inputs. As with the outputs,

1. a $*$ entry in the vector indicates that there is no path between the input corresponding to that entry and this vertex.
2. a \sim entry in the vector indicates that there are differing number of latches along different paths to the input corresponding to that entry.

With this a recursive procedure for computing W is now outlined. W is computed by computing V for each of the outputs. For a vertex v in the graph, $V(v)$ is computed by first computing $V(u)$ for each u such that the edge (u, v) exists in the graph and then composing these as described in Figure 2.11. The recursion stops at the inputs of the graph. For input i , V has a 0 in the i^{th} position and a $*$ in all the other positions. The pseudo-code for this algorithm is given in Figure 2.11. Some clarification is needed for the operators “+” and “&” used in procedure `compute_V`. “+” is a binary operator whose first argument is a path weight vector and the second is an integer. The result is a path weight vector. The effect of “+” is to add on the additional number of latches along that edge to the previously computed path weight vector for the source vertex of the edge. Each

```

/* inputs:    communication graph, graph
  outputs:    path weight matrix, W
*/

compute_path_weight_matrix(graph){

    /* compute V for each output and concatenate */
    foreach_output(graph){
        compute_V(output);

        /* W is just the concatenation of these V's */
        concatenate(W, V(output));
    }

    return W;
}

/* inputs:    vertex, v
  outputs:    path weight vector, V(v)
*/

compute_V(v){

    /* termination condition */
    if(v is input){

        /* for ith input vertex return vector with 0
           at ith position and * at all others */

        return input_V(v);
    }

    /* V initialized to vector with all *'s */
    initialize(V(v));

    /* recursive computation */
    foreach_edge(e(u,v)){
        compute_V(u);

        V(v) = V(v) & (V(u) + w(e(u,v)));
    }

    return V(v);
}

```

Figure 2.11: Computing the Path Weight Matrix

+	k2	&	~	*	k1
~	~	~	~	~	~
*	*	*	~	*	k1
k1	k1 + k2	k2	~	k2	if(k1=k2) then k1 else ~

(a)
(b)

Figure 2.12: Operators “+” and “&”

entry of this is computed according to the rules given in Figure 2.12(a). Here $k1$ and $k2$ are integers. $k2$ is the second argument of the “+” operator. “&” composes the path weight vectors produced after the “+” operation for all incoming edges. “&” operates on each individual entry according to the rules specified in Figure 2.12(b). The aim is to obtain the path weight vector for this vertex from the path weight vector of the source vertices for each incoming edge. The execution of this algorithm has been shown in Figure 2.13 for the graph in Figure 2.10. The dotted boxes show the computation needed to determine the path weight vectors for vertices v_1 and v_2 .

The algorithm in Figure 2.11 has complexity $O(e \cdot m)$ where e is the number of edges in the graph and m is the number of inputs. Each edge in the graph is visited exactly once and “+” and “&” are applied exactly once for each edge. “+” and “&” have complexity $O(m)$ since they need to examine each entry of an m -entry vector. Thus `compute_V` has complexity $O(e \cdot m)$. Of course, `compute_V(v)` is executed only once for each vertex and the result cached for any subsequent calls. The cache retrieval is assumed to be constant time. This has not been explicitly stated in the algorithm. Note that the algorithm is symmetric in terms of the inputs and outputs. The algorithm is described in terms of the path weight vectors for the outputs which captures information about the number of latches along paths from each of the m inputs. Alternatively, it can be described in terms of the

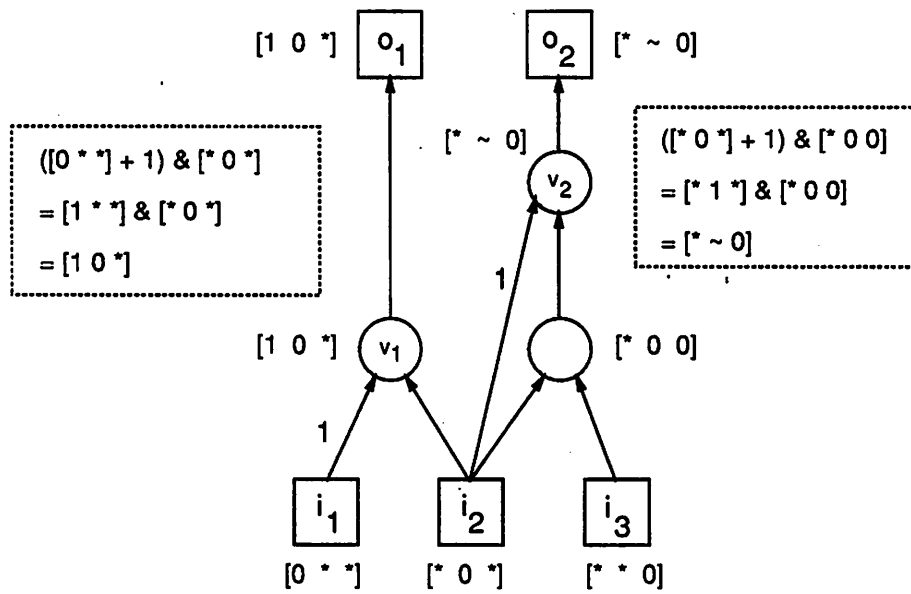


Figure 2.13: Computing the Path Weight Matrix: Example

path weight vectors for the inputs of the graph (these correspond to the rows of the path weight matrix). Their computation proceeds recursively forward from the inputs. In this case the complexity of the algorithm is $O(e \cdot n)$, where n is the number of outputs. Thus the algorithm can be modified to select the direction of recursion based on the number of inputs and outputs resulting in a complexity of $O(e \cdot \min(m, n))$.

2.3.5 Solving the Path Weight Matrix

Once the path weight matrix, W , has been computed it needs to be solved to find a satisfying assignment of α 's and β 's (or determine that none exists). If W has any \sim entries then it obviously is not satisfiable. For the rest of this section it is assumed that W has no \sim entries. Note that W is either unsatisfiable or has an infinite number of solutions. Given a particular solution, another valid solution can be obtained by adding an integer j to the α 's and subtracting j from the β 's. Any solution can be used to obtain a peripheral retiming. For simplicity, $\alpha_1 = 0$ is selected. This choice forces $\beta_i = W_{1i}$, which in turn forces $\alpha_i = W_{i1} - \beta_1$. Each entry in the matrix is then checked to ensure that $\alpha_i + \beta_j = W_{ij}$; if a violation occurs, the matrix is not satisfiable and no peripheral retiming exists for the circuit. The pseudo-code for this algorithm is given in Figure 2.14. The complexity of `solve_pwm(W)` is $O(k)$, where k is the number of integral (non-*) entries in W , since the consistency check needs to examine each integral entry in the matrix. The arbitrary selection $\alpha_1 = 0$ may not be the only arbitrary assignment necessary to compute a complete set of α 's and β 's. The circuit may have sub-circuits which are disjoint, leading to a corresponding disjoint matrix with many * entries. In this case, an arbitrary assignment to an α or a β must be made for each disjoint submatrix. The algorithm in Figure 2.14 assumes only one connected component in the graph. It needs to be applied repeatedly to each connected component in the graph. Since the complexity of finding the connected components is $O(m + n)$ the overall complexity is $O(\max(k, m + n))$.

2.3.6 Legal Resynthesis Operations

Permitting negative latches on the peripheral edges is a legitimate operation as long as the resynthesized circuit has a legal retiming. This leads to the following question:

Can we guarantee that the resynthesized circuit always has a legal retiming?

```
/* inputs:    path weight matrix, W
   outputs:   status, solution vectors  $\alpha$  and  $\beta$ 
*/

solve_pwm(W){

    /* initialize  $\alpha$  and  $\beta$  vectors */

     $\alpha = \beta = 0$ ;

     $\alpha_1 = 0$ ;

    /* compute  $\beta$ 's */
    for j from 1 to n
         $\beta_j = W(1,j)$ ;

    /* compute  $\alpha$ 's */
    for i from 2 to n
         $\alpha_i = W(i,1) - \beta_1$ ;

    /* check consistency of solution */
    for i from 2 to n
        for j from 2 to n
            if( $W(i,j) \neq \alpha_i + \beta_j$ )
                return("no solution exists",  $\alpha$ ,  $\beta$ );

    return("valid solution exists",  $\alpha$ ,  $\beta$ );
}
```

Figure 2.14: Solving the Path Weight Matrix

To examine this further we need to define a synchronous communication graph⁵.

Definition 2.3.5 *A synchronous communication graph is one in which each path between an input pin and an output pin has a non-negative path weight.*

The following lemma states how a synchronous communication graph can be legally retimed.

Lemma 2.3.4 *The following lag function results in a legal retiming in a synchronous communication graph:*

- a) $L_l(v) = sp(v)$ for each internal vertex. $sp(v)$ is the weight of the shortest path to the outputs, i.e. the path with the least weight between vertex v and an output pin.
- b) $L_l(v) = 0$ for each I/O pin

Proof.

The edge weight in a retimed circuit is given by

$$w_r(e_{uv}) = w(e_{uv}) + L(v) - L(u)$$

where edge e_{uv} is from vertex u to vertex v and L is the lag function.

Consider an internal edge and the lag function given in the statement of the lemma.

$$w_r(e_{uv}) = w(e_{uv}) + sp(v) - sp(u)$$

For $w_r(e_{uv})$ to be non-negative, $w(e_{uv}) + sp(v) - sp(u) \geq 0$ or $sp(u) \leq w(e_{uv}) + sp(v)$. This is obviously true since the weight of the shortest path from u to an output pin cannot be more than $w(e_{uv}) + sp(v)$, or the shortest path would be the edge (u, v) followed by the shortest path from v to an output pin. Thus this retiming results in all internal edges having non-negative weights.

Now consider an output edge. For an output edge, $L_l(v) = 0$.

$$w_r(e_{uv}) = w(e_{uv}) + L_l(v) - L_l(u) = w(e_{uv}) + 0 - sp(u)$$

Now, $w(e_{uv}) - sp(u) \geq 0$ since $sp(u) \leq w(e_{uv})$. Thus, $w_r(e_{uv}) \geq 0$.

Finally, consider an input edge. For an input edge, $L_l(u) = 0$. Thus,

$$w_r(e_{uv}) = w(e_{uv}) + L_l(v) - 0 = w(e_{uv}) + sp(v)$$

⁵This is related to the definition of a synchronous circuit presented in [44].

Since each input pin has exactly one out-edge, $w(e_{uv}) + sp(v)$ is the shortest path between the input pin u and the outputs. We know that all path weights between an input pin and an output pin are non-negative. Thus, $w(e_{uv}) + sp(v)$ is non-negative and $w_r(e_{uv}) \geq 0$ for an input edge.

Thus, the specified retiming is legal since all resulting edge weights are non-negative. ■

This result is used to prove the following theorem which precisely states the conditions under which a legal retiming exists.

Theorem 2.3.3 *A communication graph has a legal retiming if and only if it is synchronous.*

Proof.

If part:

Follows directly from Lemma 2.3.4.

Only if part:

If the resulting retiming is legal, then each edge weight in the retimed graph is non-negative. Thus, the path weight between an input pin and an output pin must be non-negative. By Lemma 2.3.1, retiming cannot change the path weight between an input pin and an output pin. Thus, the path weight between an input pin and an output pin must have been non-negative before the retiming. Therefore, the communication graph was (and is) synchronous. ■

Note that since the initial communication graph has no negative edges (it represents a real circuit) it is synchronous. Peripheral retiming preserves the synchronous property since retiming does not change the path weight between an input and an output pin. However, resynthesis can change the communication graph and hence it may destroy the synchronous property.

Let us see how this can happen. Let G_1 be the communication graph before resynthesis and G_2 be the graph after resynthesis. If there was a path between input i and output j in G_1 and there is a path between them in G_2 , then the path weight for this path in G_2 is $\alpha_i + \beta_j$. This is the same as the path weight W_{ij} in G_1 . Since G_1 is synchronous, this path weight is non-negative. Now consider the case in which no path existed in G_1

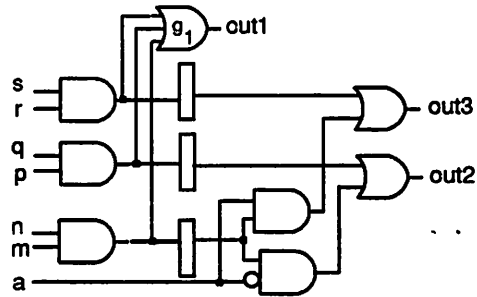
between input i and output j and resynthesis creates a path. The path weight for this path in G_2 is $\alpha_i + \beta_j$. Since α_i and β_j may be negative and G_1 did not force a non-negativity constraint on $\alpha_i + \beta_j$ (since no path existed between input i and output j), it is possible that $\alpha_i + \beta_j$ may be negative, thus destroying the synchronous property. Note that output j does not actually depend on input i ; however, resynthesis created a *pseudo-dependency* between the two.

An example is shown in Figure 2.15(a). This circuit has a peripheral retiming shown in Figure 2.15(b). Resynthesis discovers that the three-input OR gate g_1 , can be replaced by a two-input OR gate g_2 (Figure 2.15(c)). The communication graph for this circuit is not synchronous since there exists a path of negative weight (-1) between input a and output $out1$. By Theorem 2.3.3, this circuit has no legal retiming. Let us see what went wrong in terms of the logical functionality of the circuit. The circuit can be viewed as a two-stage pipeline with the latches separating the two stages. a is an input to the second stage of the pipeline and $out1$ an output of the first stage. Combinational resynthesis makes a first stage output depend on a second stage input. As a result no position of latches can be found in the circuit that will retain the original functional behavior.

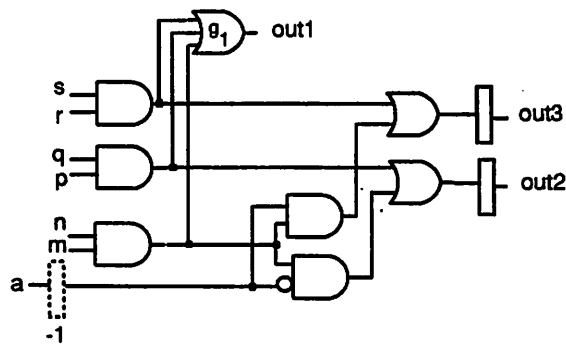
Thus resynthesis must ensure that it does not introduce a pseudo-dependency with a negative path weight; this is the only condition that the resynthesis must satisfy. There are two possible ways of dealing with this situation. We can check each resynthesis operation to ensure that it does not lead to such a topology. This method has the limitation that we need introduce these checks as part of the combinational resynthesis procedure, a move that will not permit us to use existing combinational optimization programs unaltered. Alternatively, a checkpointing approach can be adopted, where at various points in the resynthesis procedure we check that the current graph is synchronous. If not, we revert to the last synchronous graph and reject the resynthesis steps that follow. The checking involves just computing the path weight matrix, which is a fast ($O(e \cdot \min(m, n))$) operation. If the resynthesis techniques used are unlikely to destroy the synchronous property then the second method is preferable.

2.4 Peripherally Retimable Circuits: General Topology

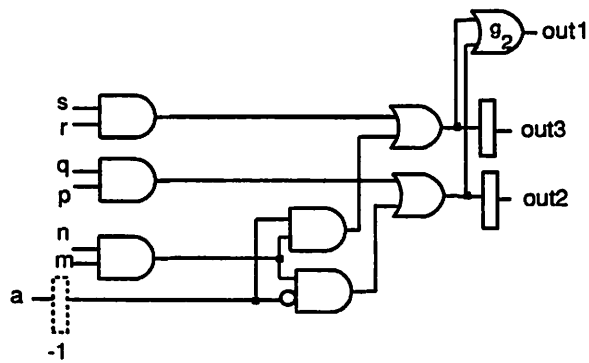
Theorem 2.3.2 states the necessary and sufficient conditions under which a circuit has a peripheral retiming. However, it gives no feel for the general topology of these circuits.



(a)



(b)



(c)

Figure 2.15: Introducing Pseudo-dependencies with Negative Path Weight

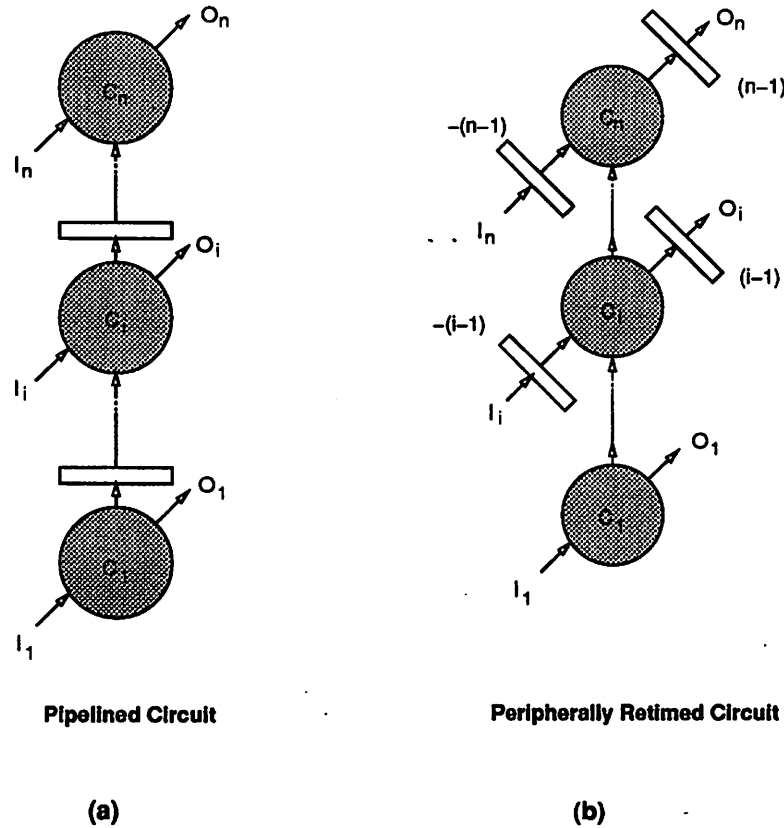


Figure 2.16: Pipelined Circuits and their Retiming

In order to better understand them we would like to characterize them in terms of their general topology.

Figure 2.16(a) shows the general topology of circuits that permit a peripheral retiming. A peripherally-retimed circuit is shown in Figure 2.16(b). This has been obtained by moving the latches forward through the circuit, borrowing latches at the inputs when required. Circuits satisfying this topology are called *balanced* or *pipelined* circuits. Note that inputs and outputs are permitted to and from each stage of the pipeline. This is more general than simple pipelines, where data enters the first stage and the result leaves the last stage. Of course any of the input or output vectors I_i or O_j may be empty.

It is now shown that this is exactly the topology corresponding to circuits that

can be peripherally-retimed. Let C be a circuit that can be peripherally-retimed and C_r be the peripherally-retimed circuit. We need to show the following:

1. In C_r , α_i is non-positive for all i and β_j is non-negative for each j .
2. There is no path from input i to output j such that $\alpha_i + \beta_j < 0$.

Let α_i latches be at input i and β_j latches be at output j in C_r . This can be modified to obtain another solution as follows. Let α_{max} be the value of the largest α_i . Subtract α_{max} from each α_i and add it to each β_j . The resulting values of α 's and β 's also satisfy the path weight matrix. As in Figure 2.16(b), C_r has non-positive α 's and non-negative β 's. To see that the β 's must be non-negative, assume that this were not true, i.e. some β_j were negative. Let i be an input from which there is a path to output j . Then the path weight in C_r from i to j must be negative since α_i is non-positive. This cannot happen since the initial graph, and hence the retimed graph, must be synchronous. Thus, all β 's are non-negative. Another consequence of the initial graph being synchronous is that there cannot be a path from input i to output j such that $\alpha_i + \beta_j < 0$ in the retimed circuit. Thus, C_r satisfies both the conditions listed above and its topology is precisely that of Figure 2.16(b). Therefore C must have the topology of Figure 2.16(a).

The circuit in Figure 2.16(a) naturally suggests that the latches are pipeline latches and that there is an underlying combinational logic block. Peripheral retiming exposes this by moving the latches to the periphery. It may seem that the combinational logic can be exposed by just ignoring the latches (replacing them by wires). This certainly is true. However, once this circuit has been resynthesized, the latches need to be placed back in the circuit. This needs to be done while guaranteeing that each input and output is in the correct stage of the pipeline. Peripheral retiming handles this elegantly. The latches at the periphery are place holders for this information. When the circuit is retimed after resynthesis, retiming ensures that each input and output lies in the correct stage in the pipeline. The significance of negative latches is reiterated; without them it would not be possible to handle inputs and outputs from each stage.

2.5 Optimizing Sequential Circuits

Let us now see how the techniques discussed in Section 2.3 are applied to general sequential circuits. For those circuits that can be peripherally retimed, the entire interior

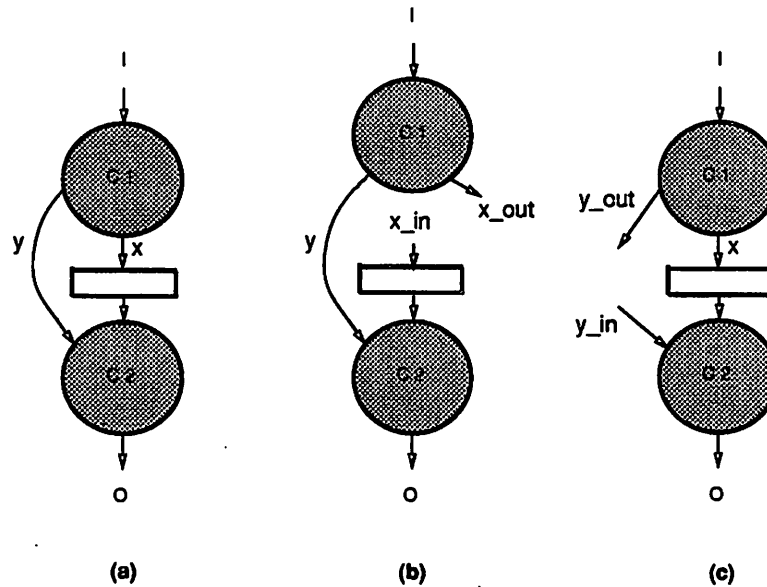


Figure 2.17: Acyclic Circuit with no Peripheral Retiming

logic block can be optimized and the latches replaced in the circuit. This section examines those that cannot be peripherally retimed.

As is illustrated by the circuit in Figure 2.17(a), an acyclic circuit may not have a satisfiable path weight matrix, and hence no peripheral retiming exists. Here, the two paths from I to O have differing number of latches along them. In this case, satisfiable sub-circuits (sub-circuits whose path weight matrices are satisfiable) are identified and created by breaking the appropriate nets. Each sub-circuit is optimized separately, and the sub-circuits are then reconnected. Consider the circuit in Figure 2.17(a). Breaking net x yields the sub-circuit shown in Figure 2.17(b). x_{out} represents additional outputs of the sub-circuit, and x_{in} represents additional inputs. This sub-circuit is satisfiable and the results of Section 2.3 can be directly applied. Finally, a circuit equivalent to the original circuit can be obtained by reconnecting the net x . Alternately, net y can be broken and the corresponding sub-circuit, Figure 2.17(c) similarly optimized. Note that the optimization of sub-circuits (b) and (c) can lead to very different results. It is not possible to predict a

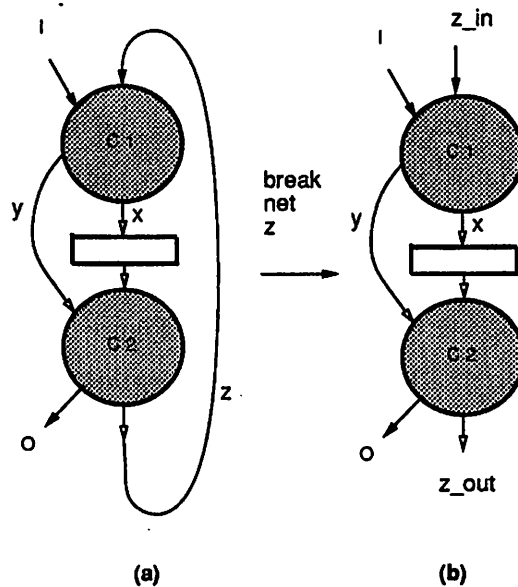


Figure 2.18: Handling Cyclic Circuits

priori which starting point leads to a better solution.

In the case of sequential circuits that have cycles in them, they first need to be made acyclic. Therefore, the first step is to choose a set of nets to cut such that all cycles are broken. However, this may not be sufficient: the resulting acyclic circuit may still have a path weight matrix that is not satisfiable. For example, breaking net z of the circuit in Figure 2.18 will break the cycle, but as in Figure 2.17(a), net x or net y still must be broken to make the path weight matrix satisfiable.

In most cases, there will be several choices of where to make cuts in the logic to create a satisfiable path weight matrix. While it is not known *a priori* which cut will yield the best results after optimization, it is simple enough to provide an interactive environment to allow the designer to experiment with several different cuts.

Consider an example of a sequential circuit that has a cyclic structure. Figure 2.19(a) shows a gate level schematic of an FSM implementation. This circuit is optimal with respect to conventional logic minimization of the combinational logic between the

latches: there are no redundant gates or connections. The cycles are broken by cutting the nets $p1$ and $p2$. This results in pseudo-inputs $p1_in$ and $p2_in$ and pseudo-outputs $p1_out$ and $p2_out$ in the circuit. The circuit is then redrawn with the signal flow unidirectional (Figure 2.19(b)). A peripheral retiming of this circuit is shown in Figure 2.19(c). An optimization of the combinational block simplifies the logic part by observing that the output of the NOR gate (signal x) may be replaced by the constant value 0 without changing the functionality of the circuit. In testing terms, the output of the x is said to be untestable for a stuck-at-0 fault.

This simplified circuit is shown in Figure 2.19(d). The circuit is retimed with a legal retiming (Figure 2.19(e)). The feedback connections are re-created and the final circuit is shown in Figure 2.19(f). This circuit has three fewer gates than the initial circuit, which represents a significant gain.

2.6 Computing Equivalent States Across Optimizations

The migration of latches raises concern about the starting state of FSM's in the minds of circuit designers and testers. The question that needs to be answered here is as follows:

Given the starting state of the initial circuit, how is the the starting state of the final circuit, obtained by applying retiming and resynthesis techniques on the original circuit, determined?

This is contained in the more general problem of determining a state in the final circuit that is equivalent to a known state in the initial circuit. An application for this arises in performing verification using simulation. If a set of simulation vectors and their responses have already been developed for the original circuit, how can these be used for the modified circuit? In order to use them again, each state in the original circuit needs to be replaced with the corresponding state in the new circuit. In [76] a procedure is provided that handles this problem for retimed circuits. Since combinational resynthesis does not migrate any latches or change the function of the input gate of a latch, there is no change in the state information in this step. Thus the use of the procedure described in [76] at each retiming step is sufficient to tackle this problem.

Related to this is the issue of initialization sequences. Typically, the design of a state machine is accompanied by the determination of an initialization sequence, i.e. a

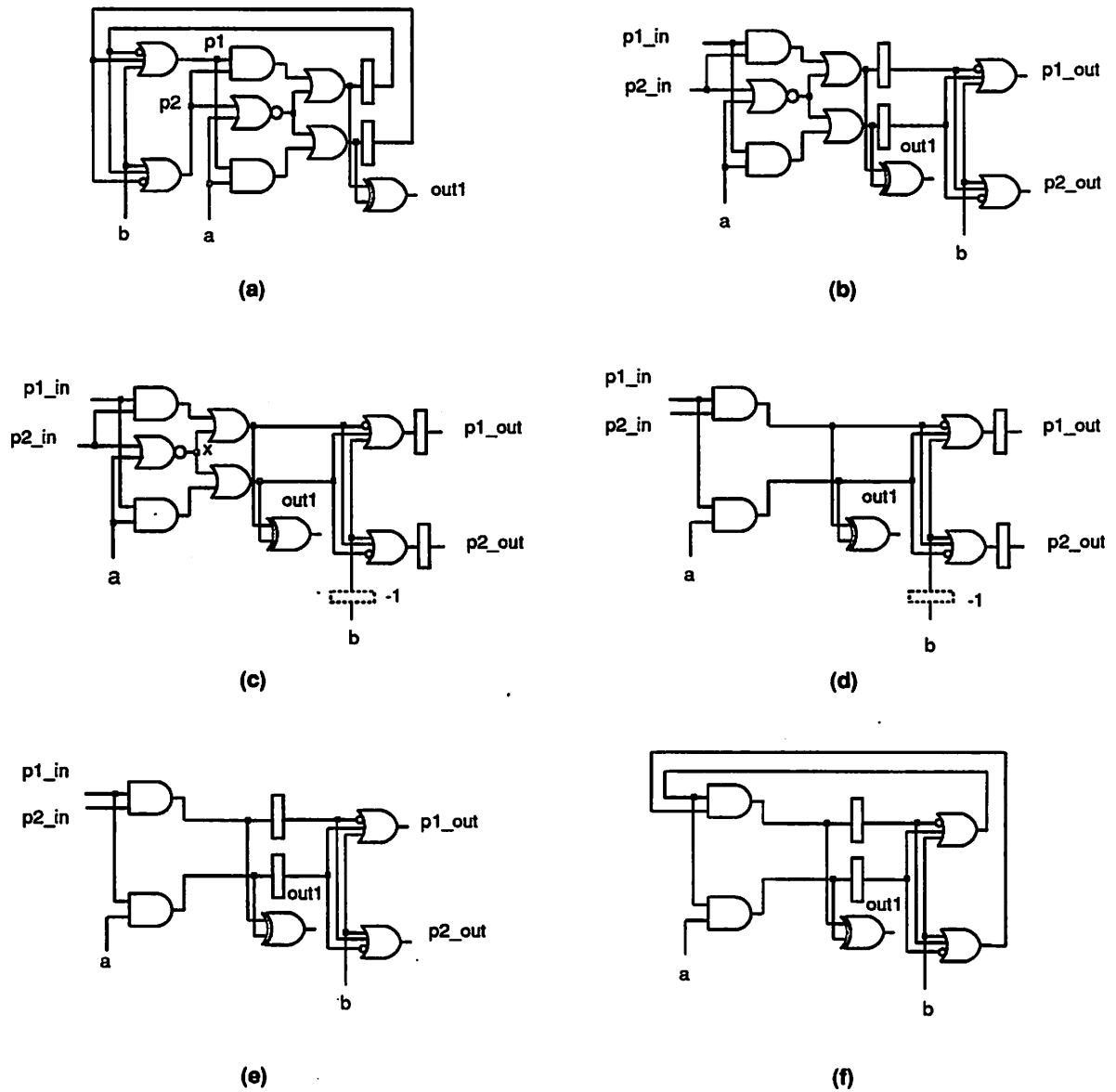
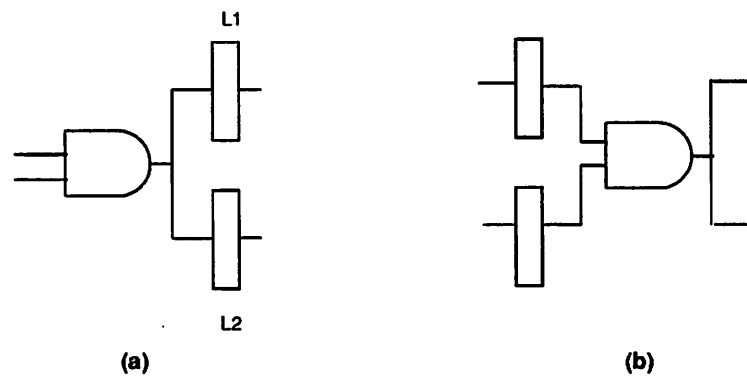


Figure 2.19: Example FSM Optimization

sequence of input vectors that is guaranteed to bring the machine to some known state (referred to as the starting state) independent of the state it is currently in. Thus, the machine may start in any possible state when it is powered on and going through the initialization sequence brings it to the known starting state. The initializing sequence may be as simple as a single input on a reset line. In [76] it is described how the initialization sequence for a retimed circuit is determined given the initialization sequence for the initial circuit. By an argument similar to that given for determining equivalent states, this procedure is applicable when both retiming and resynthesis are used.

It should be noted that in general it is possible that we may not be able to find a state equivalent to the starting state in a retimed circuit. After retiming, some or all of the latches could possibly have been replaced with wires. The only values that these wires can have are those that are consistent with the logical structure of the combinational network (i.e. no value that is part of the satisfiability don't care set [3] for the combinational circuit). If the initial state is inconsistent with this then it will not be possible to find an equivalent state in the retimed circuit. For example consider the circuit in Figure 2.20(a). Consider the state ($L1 = 1, L2 = 0$). This circuit is retimed to move the latches to the inputs of this gate as in Figure 2.20(b). There is no state in this new circuit that is equivalent to ($L1 = 1, L2 = 0$) in the original circuit. Let us consider the state transition tables for the two circuits given in Figure 2.20. A “-” in the PS column represents any possible state in the machine. State 11 in the final circuit is equivalent to state 11 in the initial circuit. States 00, 10, 01 in the final circuit are all equivalent to state 00 in the initial circuit. However, there are no states in the final circuit equivalent to either 01 or 10 in the initial circuit.

Note that the way in which the state ($L1 = 1, L2 = 0$) was reached was never specified. The only way the machine can enter this state is if there is some implicit reset circuitry in the latches. However, if all the logic associated with the reset circuitry is made explicit then this problem can never occur. The only way the latches can load data now is through the primary inputs of the circuit. As a result, the values in the latches are always consistent with the combinational circuit. Note that making the reset circuitry explicit changes the circuit that is to be retimed and thus forces the resulting retimings to be consistent with this new circuit. This results in less flexibility for the retiming algorithms. In addition, since latches will migrate during retiming, the reset logic may no longer be associated with the latches. Thus the reset logic will need to be implemented separately from the latches.



State Transition Tables

PS	IN	NS
-	00	00
-	01	00
-	10	00
-	11	11

(a)

PS	IN	NS
-	00	00
-	01	01
-	10	10
-	11	11

(b)

Figure 2.20: The Equivalent State Problem: An Example

Making this logic explicit is only needed to guarantee that we will always be able to find an equivalent state after retiming. Thus the reset logic needs to be put in only if the algorithm given in [76] determines that no equivalent state exists in the retimed circuit.

Chapter 3

Implications and Applications

There was a large mushroom growing near her, about the same height as herself: and, when she had looked under it, and on both sides of it, and behind it, it occurred to her that she might as well look and see what was on the top of it.

– Lewis Carroll, “Alice in Wonderland”

The main theoretical ideas behind retiming and resynthesis were presented in Chapter 2. In this chapter, the implications and applications of these ideas are considered. First the implications in logical testing are examined; here it is shown how the sequential testing problem for pipelined circuits is equivalent to combinational testing. Next, the relationship with state assignment is explored and the space of equivalent circuits that can be obtained using retiming and resynthesis examined. Finally, the application of retiming and resynthesis in performance optimization is presented. Here the equivalence of the performance optimization problem for pipelined circuits with the corresponding problem for combinational circuits is demonstrated.

3.1 Relationship to Logic Testing

Section 1.2.1 introduced the need for testing circuits in order to detect manufacturing defects. These defects are modeled in terms of modifications in the circuit behavior in the presence of these defects. The most common fault model is the single stuck-at fault model where a single signal or a gate output is assumed to be stuck at a constant value, 0 or 1, in the presence of the fault. A test for a fault in a combinational circuit is an input vector that distinguishes, at the outputs of the circuit, between a good and a faulty

circuit. For sequential circuits the testing problem is more complicated. A test for a fault may require the circuit to be in a particular state. Thus, a sequence of input vectors is needed that will drive the machine to this state. This sequence is known as the *justification sequence*. Next, the fault needs to be activated or excited by an *excitation vector*. However, this may only propagate the fault effect to the latches. In order to be observed, this needs to be propagated to the true outputs. This is done by applying another sequence of input vectors, called the *differentiation sequence*, that propagate the effect to the true outputs. Thus, the test for a fault in a sequential circuit has three parts: justification, excitation and differentiation. The above is a very brief introduction to testing. Details may be found in standard texts such as [15].

Work in logic optimization has established the relationship between logic optimization and logical testing under the stuck-at fault model [26]. If a stuck-at fault cannot be tested for a particular value, then the signal corresponding to the fault site can be permanently set to that value without affecting the functional behavior of the circuit. Thus, that signal or gate can be removed and the circuit simplified. In fact, in the example in Figure 2.19, the NOR gate was removed because its output was untestable for a stuck-at-0 fault in that combinational circuit.

Since combinational optimization techniques are being applied to sequential circuits, the obvious question that this raises in relation to testing is whether combinational testing techniques can be applied to test faults in sequential circuits. The answer to this is in the positive for a special class of sequential circuits, *viz.* pipelined circuits as described in Section 2.4. This is the class of circuits that permit a peripheral retiming. The remainder of this section is devoted to proving this assertion, as well as providing a method that generates tests for the sequential circuit from the combinational tests.

In the following discussion the description of pipelined circuits presented in Figure 2.16(a) is used. Let C be the pipelined circuit, C_r be the peripherally-retimed circuit, and C_c be the combinational circuit exposed by peripheral retiming. The level of a latch in C is defined as the index of the combinational logic block that it follows and the set of latches at level i is denoted as L_i . Corresponding to each latch in C , there is a signal connection in C_c . The signals corresponding to L_i are denoted as S_i . The following lemma that aids the proof of the main theorem can now be stated.

Lemma 3.1.1 *Let C be in some state and let v_k be the state vector for L_k , i.e., v_k is a*

vector of the values stored in the latches in L_k . It is possible to observe the value v_k on S_k by selecting an appropriate input vector for C_c .

Proof. The proof is by induction on the level of the latches.

Induction Hypothesis: The statement in the lemma is true for all $i < k$.

Induction Basis: Let $i = 1$. v_1 is the result of some input vector v_{I_1} applied in the previous clock cycle. In C_c , v_{I_1} will result in v_1 on S_1 .

Induction Step: v_k is the result of some input vector v_{I_k} and previous state v_{k-1} of L_k . By the induction hypothesis we know that v_{k-1} can be observed on S_k by applying some vector $(v_{I_1} \cdot v_{I_2} \cdot \dots \cdot v_{I_{k-1}})$ in C_c . Here “ \cdot ” indicates a concatenation of the vectors. Thus, v_k is observed by the input vector $(v_{I_1} \cdot v_{I_2} \cdot \dots \cdot v_{I_k})$. Note that since v_{I_k} does not depend on the inputs I_1, I_2, \dots, I_{k-1} , v_{I_k} does not cause any conflict with the previous assignments in $v_{I_1} \cdot v_{I_2} \cdot \dots \cdot v_{I_{k-1}}$. ■

With this the main result can now be proven.

Theorem 3.1.1 *Let x be a fault in C . x is testable in C_c if and only if it is testable in C .*

Proof.

If part:

Let x be testable in C . The justification sequence brings the circuit into the state needed to test the fault. If x is in combinational block C_{k+1} in the pipeline then only the state in L_k is of any concern. (If x is in C_0 then no justification sequence is needed.) Let the justification sequence result in v_k on L_k . By Lemma 3.1.1 we know that we can observe v_k on S_k by applying $(v_{I_1} \cdot v_{I_2} \cdot \dots \cdot v_{I_k})$ in C_c . Let the effect of the test be observable at outputs O_l in C . Let v_{l-1} be the state in L_{l-1} when the fault effect is observed at O_l . Thus, the excitation vector and the differentiation sequence need to load v_{l-1} in L_{l-1} . Again using Lemma 3.1.1 we know that v_{l-1} can be observed on S_{l-1} by applying $v_{I_1} \cdot v_{I_2} \cdot \dots \cdot v_{I_{l-1}}$. Note that this vector still results in v_k on S_k . Finally, all we need to observe the test at O_l in C_c is v_{I_l} . Thus, the test for the fault in C_c is the input vector $(v_{I_1} \cdot v_{I_2} \cdot \dots \cdot v_{I_l})$.

Only if part:

The proof here traces the reverse path of the proof for the if part. Let x be testable in C_c and $(v_{I_1} \cdot v_{I_2} \cdot \dots \cdot v_{I_l})$ be a test. Then the sequence $v_{I_1}, v_{I_2}, \dots, v_{I_l}$ tests the fault in C . ■

We would like to use the combinational test in C_c to generate a test for the fault in C . This is easily accomplished by observing that in Lemma 3.1.1 v_{I_i} is applied only in cycle i . Thus, the test in C is $(v_{I_1}, v_{I_2}, \dots, v_{I_l})$. Here the “,” separating the vectors indicates that they are applied in sequence.

It should be pointed out that in terms of obtaining test vectors these results are practically not very significant since pipelined circuits are relatively easy to test. However, this has an interesting application in testing using the scan approach¹. If we identify portions of the circuit that are pipelined then we know that the pipeline latches are not a problem and we can determine tests for this part using combinational techniques. Thus, the pipeline latches need not be scan latches and no additional testing penalty need be paid for this. This application has been suggested in an independent analysis in [33].

3.2 Relationship to State Assignment

Figure 2.19 illustrates an example of applying retiming and resynthesis to a given FSM implementation. Figure 3.1 shows the transition behavior of the initial and final circuits in terms of their state transition tables. The only difference is in the third row in the table. Here in the final circuit the transition is to the state 00 instead of 11. While it may seem that this is a modification of the behavior of the state machine, in fact this is not so. States 00 and 11 are equivalent in the original machine and thus the switch in the transition from 00 to 11 preserved the original behavior of the circuit. Retiming and resynthesis exploited this equivalence in simplifying the circuit. In this example, retiming did not change the number or final position of the latches. However, it could potentially change that too. Thus, the final circuit may correspond to a different but equivalent STG with some other state assignment. This leads to the following question:

Given a circuit implementation with some state assignment, is it possible, using only retiming and resynthesis, to obtain any equivalent implementation with any other state assignment?

The following result partially answers this question.

¹In the scan approach, the latches can be observed as true outputs. This reduces sequential testing to combinational testing. Typically scan latches are more expensive in terms of area and testing time than normal latches.

Initial Circuit				Final Circuit			
PS	IN	NS	OUT	PS	IN	NS	OUT
00	0-	10	1	00	0-	10	1
00	1-	11	0	00	1-	11	0
01	00	11	0	01	00	00	0
01	10	01	1	01	10	01	1
01	01	10	1	01	01	10	1
01	11	11	0	01	11	11	0
10	01	10	1	10	01	10	1
10	11	11	0	10	11	11	0
10	-0	00	0	10	-0	00	0
11	0-	10	1	11	0-	10	1
11	1-	11	0	11	1-	11	0

Figure 3.1: Impact on Transition Behavior

Theorem 3.2.1 *Given a machine implementation M_1 corresponding to a state transition graph G , with a state assignment S_1 , it is always possible to derive a machine M_2 corresponding to the same state transition graph G , and a state assignment S_2 by applying only a series of resynthesis and retiming operations on M_1 .*

Proof.

Given M_1 we would like to obtain M_2 using only a series of resynthesis and retiming steps. Figure 3.2(a) shows the schematic for M_1 . N is the combinational logic that computes the next state and output functions. Since there is a one-to-one mapping between the states of M_1 and M_2 , it is possible to construct a circuit C such that given the code for a state of M_1 as input, the output is the code for the corresponding state in M_2 . Similarly, the inverse circuit C^{-1} can be constructed that takes a state of M_2 as input, and outputs the code for the corresponding state in M_1 . Note that C followed by C^{-1} is the identity circuit, i.e., for this circuit, the output is the same as the input. This construction is shown in Figure 3.2(b). The inputs to the state latches are resynthesized as C followed by C^{-1} . Now, the state latches may be moved to between C and C^{-1} by retiming as shown in Figure 3.2(c). This circuit corresponds to the state assignment S_2 . Any other circuit corresponding to state

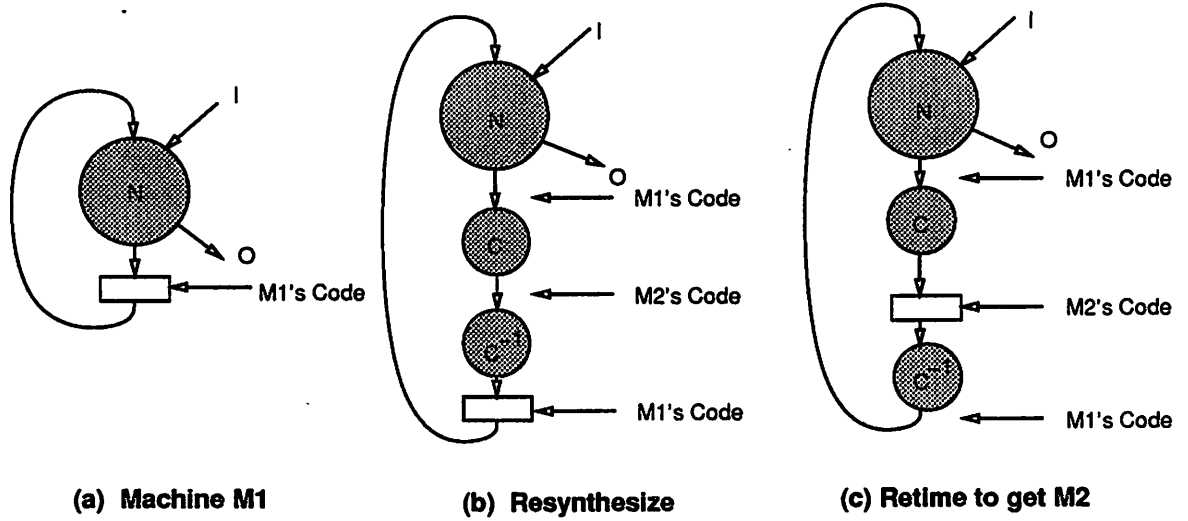


Figure 3.2: Obtaining Equivalent FSM Implementations

assignment S_2 may be obtained by the resynthesis of the combinational logic. ■

Theorem 3.2.1 does not consider the possibility of M_2 having an arbitrary STG. Let us now examine the limitations of retiming and resynthesis when M_2 may have an arbitrary STG, G_2 , that is equivalent to G_1 for M_1 .

G_1 may be modified to obtain G_2 through a series of transformations. These transformations can create states that are equivalent to existing states, merge states that are equivalent and modify state transitions to go to states equivalent to the original destinations. Let us consider three such transformations.

2-way split A state s_1 in G_1 is equivalent to two states in G_2 . This is illustrated in Figure 3.3. Here s_1 in G_1 is equivalent to states s_{11} and s_{12} in G_2 . The transitions that go to s_1 are split between s_{11} and s_{12} . Besides this, G_1 and G_2 are identical.

2-way merge This is the opposite of a two-way split, here two equivalent states s_{11} and s_{12} in G_1 are merged to a single state s_1 in G_2 . (See Figure 3.3.)

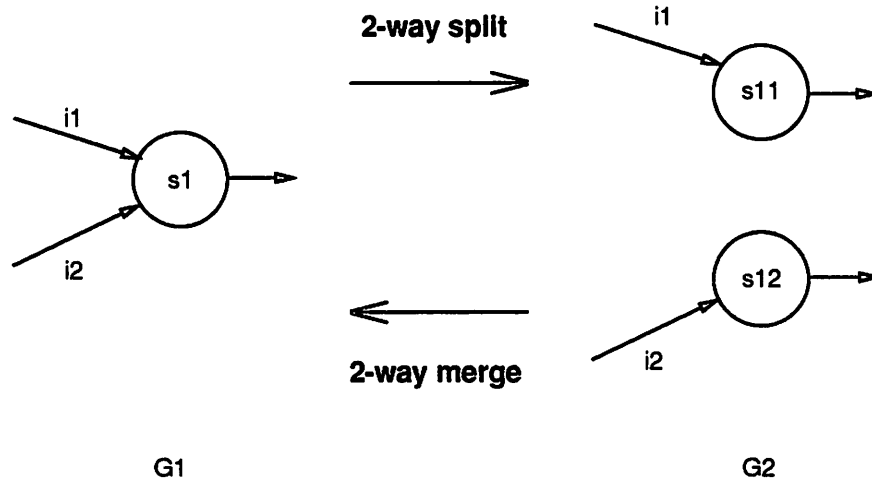


Figure 3.3: 2-way split and merge

switch Here a transition in G_1 to a state s_{11} is modified to go to an equivalent state s_{12} in G_2 . (See Figure 3.4.)

Here the assertion is made that all valid transformations are some sequence or combination of splits, merges and switches. A formal proof of this assertion is beyond the scope of this work. The following lemma states the primitive operations that realize all splits, merges and switches.

Lemma 3.2.1 *2-way split and 2-way merge are the primitive transformations. Other transformations can be built using a sequence of these.*

Proof. A switch can be achieved by first applying a 2-way merge on the states involved in the switch and then applying a 2-way split to effect the switch. This is illustrated in Figure 3.5.

Multi-way splits and merges can be accomplished by a sequence of 2-way splits and merges. ■

Thus, the task at hand has been reduced to showing that retiming and resynthesis can handle 2-way splits and merges. Unfortunately, even this is difficult to handle. A few

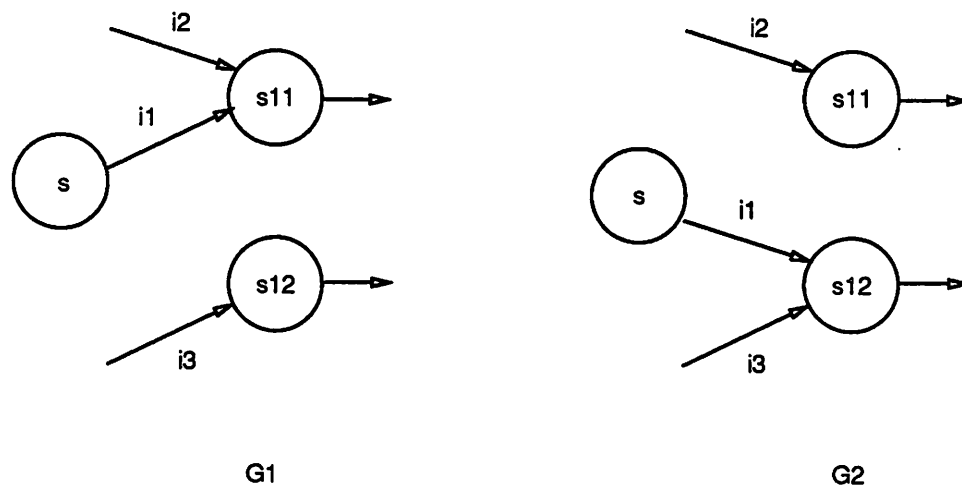


Figure 3.4: Switch

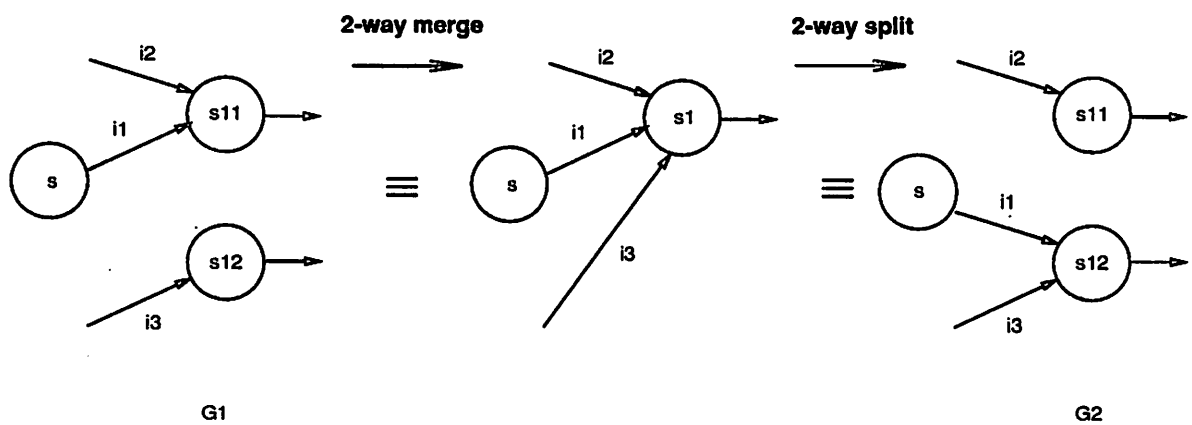


Figure 3.5: Switch using 2-way merge and split

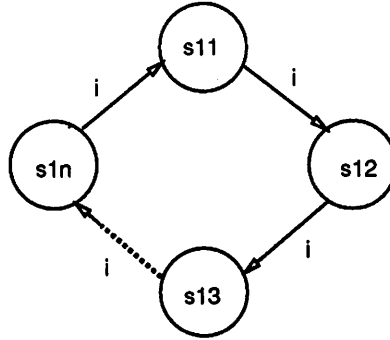


Figure 3.6: Labelled Cycle of Equivalent States

more definitions are needed in order to examine the restricted class of transformations that can be handled.

Definition 3.2.1 *A labelled cycle of equivalent states in an STG is a directed cycle such that all state vertices in the cycle are equivalent and all transition predicate vectors on the edges in the cycle have the same label.*

Figure 3.6 is an illustration of this definition. Here the states $s_{11}, s_{12}, \dots, s_{1n}$ are all equivalent. The input i is the same transition predicate for each edge of the cycle.

Definition 3.2.2 *A cycle preserving (CP) transformation does not create or destroy a labelled cycle of equivalent states.*

Figure 3.7 illustrates the non-CP versions of the three transformations described earlier. The following lemma is the restriction of Lemma 3.2.1 to CP transformations.

Lemma 3.2.2 *CP 2-way split and 2-way merge are the primitive CP transformations. Other CP transformations can be built using a sequence of these.*

Proof. Similar to proof for Lemma 3.2.1. ■

Finally it can be shown that retiming and resynthesis can handle CP transformations.

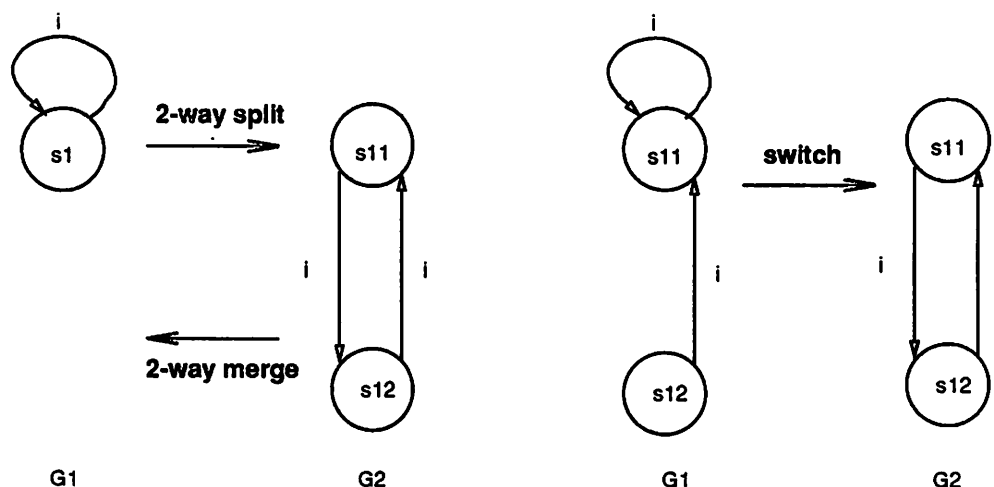


Figure 3.7: Non-CP Transformations

Theorem 3.2.2 *Let M_1 be an implementation corresponding to state assignment S_1 and STG G_1 and M_2 be an implementation corresponding to state assignment S_2 and STG G_2 . If G_2 is obtained from G_1 using only CP transformations then M_2 can be obtained from M_1 using only a sequence of retiming and resynthesis operations.*

Proof. Lemma 3.2.2 permits us to restrict ourselves to CP 2-way splits and merges.

First consider G_2 to contain a CP 2-way split of some state s_1 in G_1 . A transition to s_1 in G_1 corresponds to a transition to either s_{11} or s_{12} in M_2 depending on the primary input vector. Since the transformations are CP, the primary input vector and state s_1 uniquely determine which of s_{11} or s_{12} is the destination state in M_2 . This is not possible for a non-CP 2-way split. Thus, the one-to-many mapping between the codes for M_1 and the codes for M_2 is actually a one-to-one mapping between the M_1 codes plus the primary inputs and M_2 codes. This is accomplished through circuit C in Figure 3.8(b). Circuit C^{-1} performs the inverse mapping which is a many-to-one mapping between M_2 codes and M_1 codes and does not need I as input. Finally, Figure 3.8(c) shows how this circuit may be retimed resulting in a circuit that corresponds to G_2 . As in Theorem 3.2.1, this may be further resynthesized to any circuit M_2 that corresponds to state assignment S_2 .

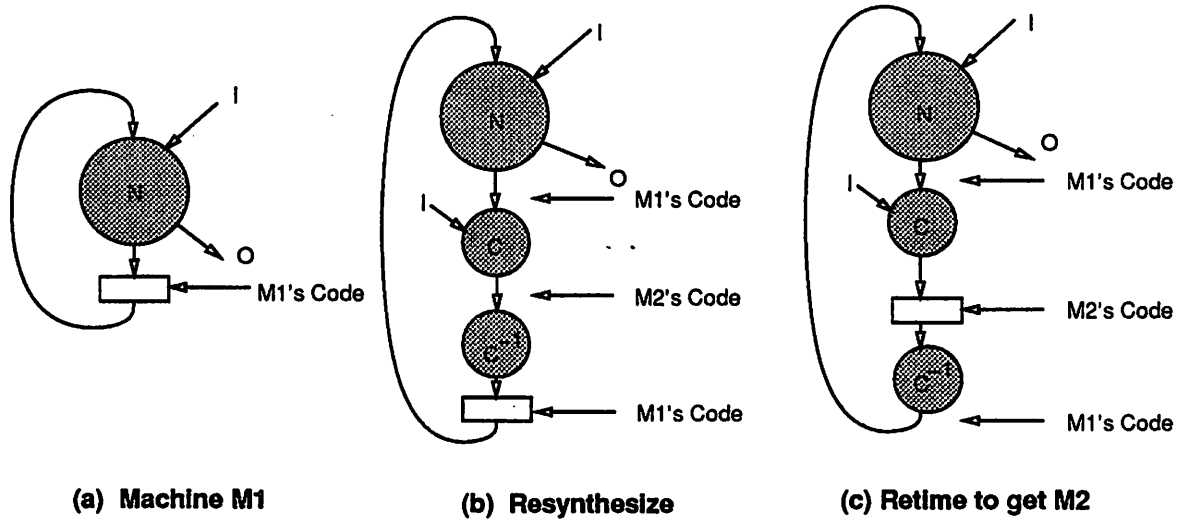


Figure 3.8: Obtaining Equivalent FSM Implementations

Since each step in retiming and resynthesis is reversible, we can obtain M_1 given M_2 using only retiming and resynthesis. But, M_1 is obtained from M_2 by applying a 2-way merge. Thus 2-way merges can be handled using retiming and resynthesis. ■

This result shows that using retiming and resynthesis does not restrict you to a small part of the solution space, but rather enables a very large space of functionally equivalent circuits to be explored. However, it does not give any insight into how this space may be searched for circuits that are optimal with respect to any given criterion. Also, it should be noted that the circuits here are completely specified, i.e., there are no don't care conditions associated with the circuits. Additional advantage may be gained by exploiting these don't cares.

3.3 Performance Optimization

The three step procedure of peripheral retiming, combinational resynthesis and a final legal retiming, to exploit combinational optimization techniques beyond latch bound-

aries was described in Chapter 2. In all the examples presented there, combinational optimization was being used for area reduction. However, the retiming and resynthesis framework does not specify the nature of the resynthesis permitted; any combinational resynthesis is valid. This section examines the use of performance-directed resynthesis in this framework. The crucial element in using the dual techniques of retiming and resynthesis is the decision as to when, where, and in what order to apply these operations. Some preliminary work in combining them has been done in [4], but the approach there is ad-hoc with no optimality guarantees. Here a rigorous solution to this problem is considered for a special yet important class of circuits, *viz.* pipelined circuits. (The work described in this section was first presented in [50].) As seen previously, these are precisely the class of circuits which permit a peripheral retiming. The problem of speeding up a pipelined circuit is transformed into one of resynthesizing a combinational logic circuit with appropriate timing constraints. This is achieved by first using peripheral retiming. The resulting maximal combinational sub-circuit can be subject to any delay-reducing transformation in an effort to meet the timing constraints specified for it. The timing constraints are based on the cycle time that the designer desires. It is shown that if the timing constraints on the maximal combinational sub-network are met then it is always possible to retime the resynthesized circuit to meet the desired cycle time. It is also shown that any circuit that meets the cycle-time constraint can be obtained by peripheral retiming, appropriate resynthesis and then retiming.

3.3.1 Two Problems in Performance Optimization

Pipelined Performance Optimization

Pipelined circuits were introduced in Section 2.4. Recall that a pipelined circuit of n stages consists of n combinational circuits (C_1, \dots, C_n) with stage i communicating with stage $i + 1$ through some signals that are latched. (See Figure 3.9(a).) Each C_i may have inputs I_i and outputs O_i besides the latched inputs and outputs used to communicate with adjacent pipeline stages. This description of a pipeline is general since it does not restrict all inputs to come in at the first stage and outputs to leave the last stage. Allowing inputs to any stage (rather than just the first stage) in the pipeline provides the following additional flexibility:

1. The pipeline can operate on multiple streams of data which may arrive separated from each other by an arbitrary number of clock cycles.

2. The pipeline can consider control signals that arrive in cycles subsequent to the initial data and perform the remaining computation based on these.

Similarly allowing outputs from any stage permits the following:

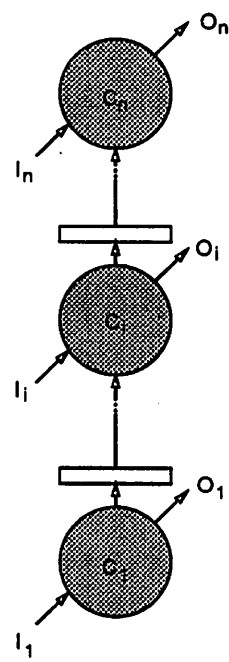
1. Multiple computations may be performed with the different results being available during different clock periods.
2. Error conditions or any status signals can be provided as outputs in the cycle they are computed.

This description of pipelined circuits is general and includes most circuits that are considered to be pipelined by digital circuit designers.

The performance optimization problem of pipelined circuits is to maximize the clocking rate or equivalently minimize the cycle time of the circuit. This determines the throughput of the circuit. Sometimes the cycle time, c , that needs to be attained is specified as a constraint, based on the requirements of the system of which this circuit is a part. In that case, the optimization problem is to meet the specified cycle time constraint. In general, not all inputs are available at the clock edge; for example, they may arrive later because of communication delays. Similarly some of the outputs may be required well before the clock edge, say, in order to satisfy setup time requirements. Thus, with each input signal an arrival time, a , is associated which is the time after the clock edge that the signal is available and with each output signal a required time, r , is associated which is the time before the clock edge that this signal must be ready. Let A and R be the sets of arrival times and required times for the inputs and outputs respectively. These capture the timing constraints due to the environment of this circuit. Thus an instance of the pipelined performance optimization problem \mathcal{P}_P , is specified as $\mathcal{P}_P = \{C, c, A, R\}$, where C is the circuit and c the desired cycle time constraint. Note that this problem statement assumes that c is provided as part of the problem. If the problem is to minimize c , this can be done by solving a number of problem instances \mathcal{P}_P , each with a known c , by performing a binary search for the least feasible c .

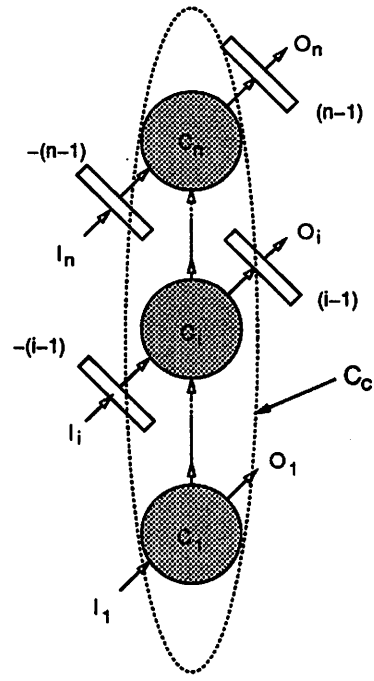
Combinational Speedup

The combinational speedup problem has been well studied in recent years. Here a given combinational circuit, C , is to be resynthesized so that it meets its timing constraints. The timing constraints are specified as required times, r , on the outputs of the combinational



Pipelined Circuit

(a)



Peripherally Retimed Circuit

(b)

Figure 3.9: Peripheral Retiming of Pipelined Circuits

circuit. In this case these times are absolute and not with reference to any clock edge. As before, the inputs may be arbitrarily delayed and an absolute arrival time a is associated with each input. Thus, an instance of this problem, \mathcal{P}_C , is specified as $\mathcal{P}_C = \{C, A, R\}$.

Problem Transformation

Given an instance of the pipelined performance optimization problem, \mathcal{P}_P it is transformed to an instance of the combinational speedup problem \mathcal{P}_C . Subsequently it is demonstrated how a solution of \mathcal{P}_C may be retimed to obtain a solution of \mathcal{P}_P .

As described in Section 2.4 we obtain a peripheral retiming for the pipelined circuit by retiming block C_i by $-(i-1)$. (See Figure 3.9(b).) This is not unique, several possible peripheral retimings exist. In terms of the circuit this implies that we sweep all the latches forward through the circuit, borrowing latches at the inputs as needed. This results in the peripherally-retimed circuit shown in the Figure 3.9(b). As before, peripheral retiming has exposed a larger combinational circuit, C_c , which is the cascade of C_1, C_2, \dots, C_n . This is the combinational circuit that we will use for the transformed problem, \mathcal{P}_C . In order to complete the definition of \mathcal{P}_C we need to specify A and R . We determine them by using c , A and R from \mathcal{P}_P as follows. For an input to stage i of C with arrival time a in \mathcal{P}_P the arrival time in \mathcal{P}_C is $a + (i-1)c$. For an output of stage i in C with required time r in \mathcal{P}_P , the required time in \mathcal{P}_C is $ic - r$. Note that it is easy to determine which stage an input or an output belongs to by looking at the number of latches in the peripherally-retimed circuit at that input or output. The intuition behind this choice of modifications of arrival and required times is that it captures the fact that the input arrives only after $i-1$ cycles and the output is required only before the i^{th} cycle. In the next section we will see how this choice of A and R results in an interesting relationship between problems \mathcal{P}_P and \mathcal{P}_C .

3.3.2 Main Results

We would like to obtain a solution of \mathcal{P}_P from a solution of \mathcal{P}_C . It is not immediately obvious that this is possible, in fact it seems almost unlikely to happen as the analysis below shows. However, we need a few simple definitions first.

Definition 3.3.1 *A path in a circuit is an alternating sequence of consecutive connections (possibly latched) and gates.*

As in [43, 44] a propagation delay, d_g , is associated with each gate in the circuit.

Definition 3.3.2 *A segment of a path is a path from an input or a latch to an output or a latch.*

Definition 3.3.3 *The delay of a segment, s , is the sum of the gate delays along the segment and is denoted by D_s .*

Definition 3.3.4 *The lumped delay of a segment s , denoted by λ_s , is the sum of the combinational logic delays associated with the gates and the times associated with late arrival of an input or early requirement of an output. Thus, $\lambda_s = a + D_s + r$.*

Of course, a or r are equal to 0 if the segment has no input or output respectively.

Definition 3.3.5 *A latch is said to be forward blocked if it cannot be legally moved forward across its output gate, g , without violating a cycle time constraint.*

This may happen for two reasons:

1. There exists a path from an input to g which does not have any latch on it. Thus, it is never possible to have a latch at each input of g which is needed for the legal forward motion of the latch. (See Figure 3.10(a).)
2. There exists a path from a critical latch to g . Informally a latch is critical if it cannot be moved forward without violating the cycle time constraint. (The formal definition of a critical latch is deferred till later in this section.) Thus it is not possible to move a latch from each input of g to its output. (See Figure 3.10(b)).

Definition 3.3.6 *A latch is said to be backward blocked if it cannot be legally moved backwards through a gate g .*

For this to happen, there must be a path from g to an output which does not have a latch on it. Thus, it is never possible to have a latch at each output of g , which is needed to move this latch backwards. (See Figure 3.10(c).)

Consider a path, p , from an input of stage i to an output of stage $i + j$ in \mathcal{P}_C . Assume that the solution to \mathcal{P}_C just meets the delay constraints for this path, i.e. the path is critical for \mathcal{P}_C . Therefore the combinational logic delay, $\text{delay}(p)$, along this critical path is given by:

$$\text{delay}(p) = ((i + j)c - r) - (a + (i - 1)c) - \epsilon$$

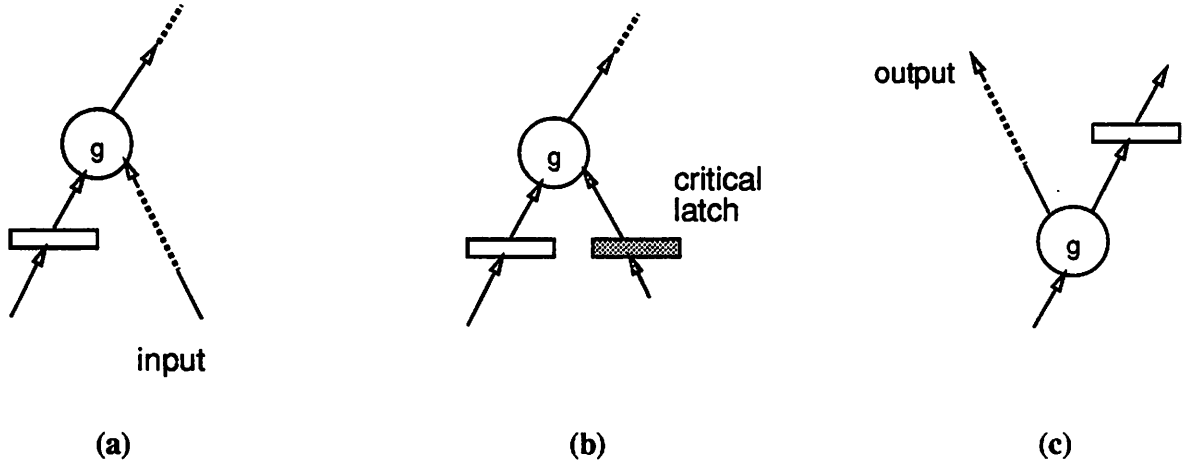


Figure 3.10: Blocked Latch Motion

where ϵ is an arbitrarily small positive number. Simplifying this results in:

$$\text{delay}(p) = (j + 1)c - r - a - \epsilon \quad (3.1)$$

Now suppose that we do find a final retiming that meets the cycle time constraint c along this path. Since retiming does not change the number of latches between any input and output, there will be j latches between the input-output pair in consideration. Thus, in the retimed circuit this path has $j + 1$ segments. Then for each segment, $\lambda_k < c$. Summing over k for this path, we see that:

$$\sum_{k=0}^j \lambda_k < (j + 1)c$$

which gives

$$a + \sum_{k=0}^j D_k + r < (j + 1)c$$

or equivalently

$$\sum_{k=0}^j D_k < (j + 1)c - r - a$$

But $\sum_{k=0}^j D_k$ is $\text{delay}(p)$ and we know from Equation 3.1 that this is equal to $(j + 1)c - r - a - \epsilon$. Thus we cannot add any delay to any segment without violating the cycle constraint.

Therefore each path segment along this path is critical in as much as moving any latch would violate the cycle constraint. It is possible that the solution to \mathcal{P}_C may actually result in the constraints along all paths in C_c to be just met. In that case, for each path there is only one available position of latches that will meet the cycle time constraint. Since different paths overlap, it is possible that the positions of latches dictated by each path may be in conflict.

To make matters worse, latches cannot be arbitrarily placed along a path as assumed by the above analysis. The latch motion may be blocked in either the forward or backward direction. This makes it even less probable that the latches can be positioned so that for all the paths they lie at the single position that meets the cycle time constraint and simultaneously avoid being blocked by the positions of inputs and outputs.

Since there are only discrete positions along the length of the path where a latch may be placed, *viz.* before and after gates, the granularity of control that we have over the latch positions is only the largest possible gate delay δ . To handle this, the following relaxed problem is defined: $\mathcal{P}_{P'} = \{C, c + \delta, A, R\}$ derived from \mathcal{P}_P . A solution to $\mathcal{P}_{P'}$ exceeds the cycle time constraint for \mathcal{P}_P by no more than δ .

To show that any solution of \mathcal{P}_C can be retimed to get a solution of $\mathcal{P}_{P'}$ a few more concepts must be considered.

Definition 3.3.7 *A segment is said to be critical if it just meets the performance constraints for that segment, i.e. the latch at the end of the segment cannot be moved forward across a gate g without violating the constraints for $\mathcal{P}_{P'}$. Thus, $c \leq \lambda_k < c + \delta$ and $\lambda_k + d_g \geq c + \delta$ for a critical segment.*

Definition 3.3.8 *A segment is said to be violating if it does not meet the timing requirements for $\mathcal{P}_{P'}$. Thus, $\lambda_k \geq c + \delta$ for a violating segment.*

Definition 3.3.9 *A path is said to be critical if all its segments are critical.*

Definition 3.3.10 *A latch is said to be critical if it terminates a critical segment.*

Definition 3.3.11 *A path is said to be violating if the last segment is violating and all the other segments are critical.*

We label each input of the circuit based on the number of latches at that input in the peripherally-retimed circuit in Figure 3.9 (b). If an input has $-(i)$ latches, then its label

is i . Each latch in a pipelined circuit can be labelled by a unique integer that specifies its level in the circuit. This labelling obeys the following two rules:

1. If there is a purely combinational path (with no latches) from an input with label i to this latch, then the label on the latch must be i .
2. If there is a path with no latches from a latch with label $i - 1$ to this latch, then its label must be i .

For a pipelined circuit, a unique labelling exists for the latches which satisfies both rules.

The following basic lemma is critical to prove the equivalence of $\mathcal{P}_{P'}$ and \mathcal{P}_C .

Lemma 3.3.1 *Let k be any integer not exceeding the largest label on any latch in C . Then, it is possible to legally retime C to C_r , such that in C_r :*

1. *There exists a violating path from an input to an output that passes through latches labelled only $< k$, OR*
2. *The following three conditions are satisfied:*
 - (a) *No segment starting at a latch or input labelled $< k$ is violating.*
 - (b) *Each latch labelled $\leq k$ is either critical or forward blocked.*
 - (c) *Each critical latch terminates a critical path from some input to that latch.*

Proof. The proof is by induction on k .

Induction Hypothesis: Assume the statement in the lemma is true for all $i < k$.

Induction Basis: The statement in the lemma is proven for $i = 0$.

Let us try and place all latches (using retiming) with label 0 so that the input segments do not violate the cycle time constraint $c + \delta$.

Case 1: It is not possible to do so. Then one of the following must be true.

1. There exists a path from an input to an output for which $a + D + r \geq c + \delta$. This path is a violating path from an input to an output and thus the first condition in the lemma is satisfied.
2. There exists a path from an input, i , to a latch, l , for which $a + D \geq c + \delta$. This latch must be backward blocked or else we would have moved it backwards to get rid of this violation. If it is backward blocked then the input gate of this latch must have a

path to an output, o , with no latch on it. Thus, for the path from i to o , the following must be true: $a + D + r \geq c + \delta$. This follows from the fact that r is non-negative and you need to go through at least the same and possibly additional logic while going from i to o instead of going from i to l . This input to output path is violating and the first condition in the lemma is satisfied.

Case 2: It is possible to do so. Then we can move each latch with label 0 forward till it is either critical or forward blocked. If a latch is critical then it must be so because it terminates a critical path from some input. Now all three parts in the second condition of the lemma statement are satisfied.

Induction Step: It is now proven for $i = k$. As in the base step we can either place all latches with label k without violating the cycle time constraint $c + \delta$, or we cannot. Consider each of these separately.

Case 1: We cannot. Then by an argument similar to that used in Case 1 of the base case, there is a segment from either a latch or an input to an output that is violating. In fact, this violating segment must be from an input or a critical latch. To see why this must be so, suppose that the violating segment is from a non-critical latch to the output. Since the non-critical latch was blocked by either a critical latch or an input, the segment from this critical latch or input to the output is violating. If it is from an input, then the path from this input to the output is violating satisfying the first condition of the lemma. If it is from a critical latch, by the induction hypothesis the critical latch terminates a critical path from an input. This path along with the violating segment starting from this latch forms a violating path from an input to an output satisfying the first condition in the lemma.

Case 2: We can. Then each latch with label k can be moved forward until it is either forward blocked or critical while ensuring that no segment is violating. For each critical latch with label k there must be a critical segment from an input or a critical latch. The reasoning as to why all critical segments to a latch cannot be from non-critical latches is the same as that used in Case 1 above. If the critical segment is from an input, then the critical latch does terminate a critical path from an input. If it is from a critical latch, then by the induction hypothesis this critical latch terminates a critical path starting at an input. The concatenation of this critical path with this critical segment gives the required critical path from an input for the critical latch with label k which satisfies the second condition in the lemma. ■

With this we can now prove the following.

Lemma 3.3.2 *If the solution to \mathcal{P}_C cannot be retimed to meet cycle time constraint $c + \delta$, then there must exist a violating path from an input to an output.*

Proof. Let k be the maximum label on a latch in any retimed circuit. Using Lemma 3.3.1 with this k , we see that either there is a violating path from an input to an output, in which case we are done, or all critical latches with label k terminate a critical path starting at an input. Note that since the cycle time constraint was not met, there must be a violating segment starting at a latch or input with label k . Actually this violating segment must be from a critical latch or input with label k by the same argument used in Case 2 of the induction step in the proof for Lemma 3.3.1. If this is from an input then this segment forms a violating path from an input to an output. If it is from a critical latch then this segment appended to the critical path from an input terminating at that latch forms a violating path from an input to an output. ■

Finally it is shown that the existence of a violating path implies that the constraint for that path in \mathcal{P}_C was not satisfied.

Lemma 3.3.3 *If there exists a violating path from an input to an output in any retimed circuit equivalent to C , then the constraint for the corresponding path was not satisfied for \mathcal{P}_C .*

Proof. By summing up the delay constraints along the violating path it is seen how the delay constraint for the corresponding path cannot be met in \mathcal{P}_C . Recall that a path is violating if the last segment is violating and all other segments are critical. Let the violating path have $j + 1$ segments. If $j = 0$, then $a + D + r \geq c + \delta$. The constraint on this path in \mathcal{P}_C is that $(kc - r) - (a + (k - 1)c) > D$ or equivalently, $a + r + D < c$ (k is the label on the input.). Thus, this constraint is violated. If $j > 0$, then the following inequalities are obtained from the fact that the path is violating.

$$\begin{aligned} a + D_0 &\geq c \\ D_i &\geq c \quad 1 < i < j \\ r + D_j &\geq c + \delta \end{aligned}$$

From these we see that:

$$a + \sum_{k=0}^j D_i + r \geq (j+1)c \quad (3.2)$$

The constraint on the same path in \mathcal{P}_C is:

$$((j+k)c - r) - (a + (k-1)c) > \sum_{k=0}^j D_i$$

This can be rewritten as:

$$(j+1)c > a + \sum_{k=0}^j D_i + r$$

which is not met as we can see from Equation 3.2 above. ■

Theorem 3.3.1 *If \mathcal{P}_C has a solution then this can be retimed to give a solution of \mathcal{P}_P .*

Proof. Follows from Lemmas 3.3.2 and 3.3.3. ■

From Theorem 3.3.1 we see that solving \mathcal{P}_C is sufficient in order to obtain a solution to \mathcal{P}_P to within a gate delay. Now it is shown that any solution to \mathcal{P}_P must be a retiming of a solution of \mathcal{P}_C .

Theorem 3.3.2 *If there exists a solution to \mathcal{P}_P then this can always be obtained by retiming some solution of \mathcal{P}_C .*

Proof. For any path from an input to an output in the retimed circuit the following inequality must be satisfied for each segment: $\lambda_k < c$. Summing over all segments:

$$a + \sum_{k=0}^j D_i + r < (j+1)c$$

The constraint on this path in \mathcal{P}_C is exactly this and is therefore satisfied. Since this is true for all paths, \mathcal{C}_c for the solution of \mathcal{P}_P is a solution of \mathcal{P}_C . The solution of \mathcal{P}_P is then obtained by moving in the peripheral latches by retiming. ■

Thus, for the pipelined problem to have a solution (within a gate delay) it is necessary and sufficient that the combinational problem has a solution. This is significant since it tells us that the problems are equivalent and therefore we need concentrate only on the relatively simpler combinational speedup problem.

Chapter 4

Practical Experiences

Let me see: four times five is twelve, and four times six is thirteen, and four times seven is – oh dear!

– Lewis Carroll, “Alice in Wonderland”

In this chapter the practical aspects of the ideas developed in Chapters 2 and 3 are discussed. Their implementation as well as practical experiences with some circuits are described. The implementation issues are considered first.

4.1 Implementation Issues

In Chapter 2 it was observed that not all circuits permit peripheral retiming; only pipelined circuits do. However, the examples shown there illustrate that even if a circuit is not pipelined, sub-circuits exist that are pipelined and retiming and resynthesis techniques can be used on these sub-circuits. Given an arbitrary circuit, there are several possible pipelined sub-circuits. (A trivial example of a pipelined sub-circuit is any single gate in the circuit. Obviously this is not a very useful sub-circuit.) Ideally we would like to determine the sequence of pipelined sub-circuits that need to be considered so as to guarantee that all logical relationships between gates are exploited. This seems to be a very difficult task. In the absence of this it is desirable to guarantee at least some locally optimum property of the pipelined sub-circuits being examined. With this in mind two different techniques were implemented to examine pipelined sub-circuits. These are now individually described.

4.1.1 Growing Pipelined Sub-Circuits from a Seed

```

/* inputs: seed, network
   outputs: network
*/

seed_network(seed, network){
    /* initialize */
    included = pending =  $\phi$ ;
    enqueue(pending, seed);
    /* add until nothing is pending */
    while(pending !=  $\phi$ ){
        node = dequeue(pending);
        add_node(node, included, pending);
    }
    return(network);
}

/* inputs: node, included, pending
   outputs: included, pending
*/

add_node(node, included, pending){
    /* initialize */
    w_list =  $\phi$ ;
    /* consider each fanin */
    foreach fanin of node{
        w = - weight(edge(fanin, node));
        if(fanin  $\in$  included){
            w_list = w_list  $\cup$  w;
        } else if(fanin  $\notin$  pending){
            enqueue(pending, fanin);
        }
    }
    /* consider each fanout */
    foreach fanout of node{
        w = weight(edge(node, fanout));
        if(fanout  $\in$  included){
            w_list = w_list  $\cup$  w;
        } else if(fanout  $\notin$  pending){
            enqueue(pending, fanout);
        }
    }
    lag = mode(w_list);
    retime(node, lag);
    included = included  $\cup$  node;
}

```

Figure 4.1: Growing Pipelined Circuits from a Seed

The least property that is desirable from a pipelined sub-circuit is that it should be maximal, i.e. no other gate in the circuit can be added to this sub-circuit and the sub-circuit still be pipelined. This property is relatively easy to guarantee if a given sub-circuit is expanded incrementally by adding a node to it along with some of its in-edges and out-edges as long as it retains the pipelined property. When no node can be added, then the sub-circuit is maximal. The starting point is some arbitrary seed node which may be specified by a designer in an interactive environment such as [70].

This is described in algorithm `seed_network` in Figure 4.1. Here the sub-circuit is peripherally retimed as it is being constructed. `network` is the communication graph representing the circuit¹. The set `included` contains those nodes already included in the sub-circuit along with all the edges between these nodes. `pending` is a queue of nodes waiting to be added to `included`. To start with, `included` is empty and `seed` is added to the front (enqueued) of `pending`. The algorithm proceeds by repeatedly taking a node from the front of `pending`, and adding it to `included` till `pending` is empty.

The addition of `node` to `included` is done as follows. Figure 4.2 shows a node, `node`, that is to be added next to `included`. Edge i from `included` to `node` has weight w_{in_i} . Similarly, edge j from `node` to `included` has weight w_{out_j} . Consider the following scenario.

$$\forall i \quad w_{in_i} = -w$$

$$\forall j \quad w_{out_j} = w$$

Now, if `node` is retimed by a lag of w , there are no latches on any edge between `node` and `included`. Combinational logic optimization can exploit the logical relationships implied by all these edges.

In general, the above scenario will not always occur. In that case, it is not possible to retime `node` so that all edges between it and `included` are latch-free. For any retiming there is at least one edge between `included` and `node` that has non-zero edge weight. (See Figure 4.3.) For combinational logic optimization the input to the latch on the edge is a primary output of the combinational circuit, and the output of the latch is a primary input to the combinational circuit. The non-zero edge weight need not be positive. A negative edge weight corresponds to negative latches on the peripheral edges of a combinational network.

¹The restriction that the communication graph be acyclic was needed in Chapter 2 for describing peripheral retiming. This is now relaxed, i.e., the communication graphs being considered here may be cyclic.

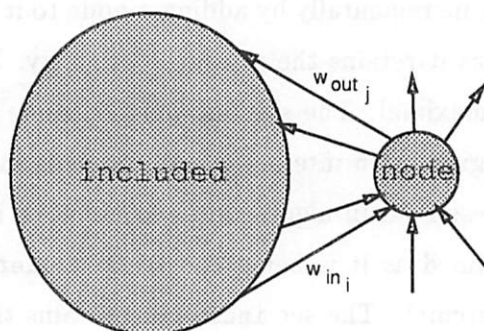
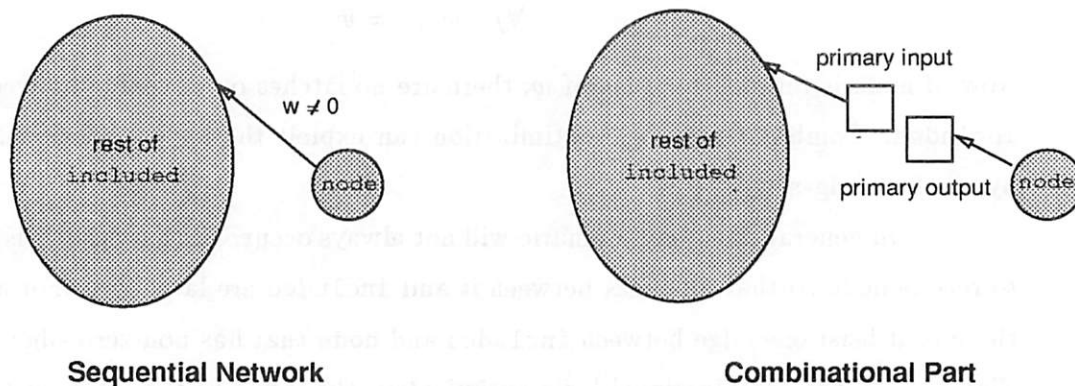


Figure 4.2: Adding node to included

Figure 4.3: Adding an Edge with $w \neq 0$ to included

Since latch-free edges permit information to be used by combinational optimization it is desirable that as many of these edges should be latch-free as possible. For an edge from `included` to `node` to be latch-free after retiming `node`, the lag for `node` should be $-w_{in_i}$. Similarly, for an edge from `node` to `included` to be latch-free after retiming `node`, the lag for `node` should be w_{out_i} . Thus, the desired lag is the modal element of the following set: $\{-w_{in_1}, \dots, -w_{in_i}, w_{out_1}, \dots, w_{out_j}\}$, i.e., the element that occurs with the most frequency.

`seed_network` will retime `node` by this lag, and add it to `included`. All other nodes connected to `node` that are not in `included` or `pending` are added to `pending`. Thus, the network consisting of nodes in `included` grows radially out from `seed`. When `seed_network` terminates, all the nodes in the circuit in the same connected component of the network as `seed` are in `included`. This follows from the following two observations. Each node that is in the same connected component as `seed` gets added to `pending` and each node added to `pending` gets added to `included`. Thus, this meets the requirement that `included` be maximal. As shown in Figure 4.3, the position of the latches determines the combinational logic circuit obtained.

A note on the data structures that are used in practice. Since inclusion checks need to be made on `included`, it is implemented as a hash table. Inclusion checks are also made on `pending`; thus it is implemented as a hash table in addition to being implemented as a queue. As the name suggests, `w_list` is implemented as a list.

The algorithm is linear in the size of the network since each node is examined once and each edge examined twice, once from each end.

Theorem 4.1.1 stated below describes the relationship between any two nodes in `included` in terms of information that can be used by a combinational logic optimizer. The following two lemmas aid its proof.

Lemma 4.1.1 *In procedure `add_node` retiming `node` by lag results in at least one edge between `node` and `included` to have weight equal to zero.*

Proof. This follows from the fact that lag is selected from `w_list` and each element of `w_list` corresponds to a lag that will result in some edge with weight 0 after retiming. ■

Lemma 4.1.2 *Let x and y be any two nodes in `included` at any point in the algorithm `seed_network`. There is an undirected path of zero weight between x and y .*

Proof.

The proof is by induction on the size of included.

Induction Hypothesis: The statement in the lemma is true for $n < k$ nodes in included.

Induction Basis: The lemma statement is true for $n = 2$ since by Lemma 4.1.1 addition of the second node to included is accompanied by the addition of a zero-weight edge between it and the first node.

Induction Step: Let $n = k - 1$. By the induction hypothesis the lemma statement is true for any pair of nodes in included. Let x be any node in included and y be the node being added. By Lemma 4.1.1 there is some node z in included such that there is a zero-weight edge from y to z after y is retimed by lag. The zero-weight path between x and z concatenated with the zero-weight edge between z and y forms the zero-weight path between x and y . ■

Theorem 4.1.1 *Let x and y be any two nodes in included at any point in the algorithm seed_network. Then one of the following must be true:*

1. *There is a directed path of zero weight from x to y .*
2. *There is a directed path of zero weight from y to x .*
3. *There is some node z in included such that there is a directed path of zero weight from both x and y to z .*
4. *There is some node z in included such that there is a directed path of zero weight from z to both x and y .*

Proof. The proof follows from Lemma 4.1.2 and the fact that if an undirected path exists between x and y then one of the conditions in the theorem statement must be true. ■

The significance of this result is that combinational logic optimization of included considers at least some interaction of any pair of nodes in included. This is certainly not true for the initial circuit.

4.1.2 Clustering Combinational Logic Blocks

There are several decisions that are made by algorithm seed_network. It determines the order in which the nodes not yet in included are added, as well as the lag for

```

/* inputs:    network
   outputs:    PI_list_array, PO_list_array, node_list_array
*/

cluster(network, PI_list_array, PO_list_array, node_list_array){
    /* initialize */
    visited =  $\phi$ ; current_block = 0;
    /* start with primary_outputs */
    foreach primary_output in network{
        if primary_output  $\notin$  visited{
            /* new cluster discovered */
            current_block++;
            PI_list = PO_list = node_list =  $\phi$ ;
            visit(primary_output, visited, PI_list, PO_list,
                node_list);
            PI_list_array[current_block] = PI_list;
            PO_list_array[current_block] = PO_list;
            node_list_array[current_block] = node_list;
        }
    }
}

```

Figure 4.4: Clustering Combinational Logic Blocks: I

retiming node. For each of these the intuitively better choice is made, but there is no guarantee regarding how good each of these decisions is. What is desirable is some more control that the designer can exercise in the choice of the resulting circuits. For example if the designer is provided with a schematic of a circuit in terms of its combinational logic blocks and latches such as that shown in Figure 2.16 and he/she can specify the lag for each individual block, then the resulting retimed circuit is completely under his/her control. The lags may be selected by the designer based on information about the relative sizes of the logic blocks, their logical functionality, etc. Note that it is not possible to determine the combinational logic clusters by looking at the initial circuit just in terms of an interconnection of a large number of gates and latches.

The connectivity information in terms of the combinational logic blocks can be obtained by clustering the nodes of the network into combinational logic blocks. Two gates

```

/* inputs: node, visited, PI_list, PO_list, node_list
   outputs: visited, PI_list, PO_list, node_list
*/

visit(node, visited, PI_list, PO_list, node_list){
    /* add node to appropriate list */
    if(node is a primary input)
        PI_list = PI_list  $\cup$  {node};
    else if(node is a primary output)
        PO_list = PO_list  $\cup$  {node};
    else
        node_list = node_list  $\cup$  {node};
    /* visit fanins in same cluster */
    foreach fanin of node{
        if(w(edge(fanin, node)) == 0){
            /* fanin in same cluster */
            if(fanin  $\notin$  visited){
                visit(fanin, visited, PI_list, PO_list, node_list);
            }
        }
    }
    /* visit fanouts in same cluster */
    foreach fanout of node{
        if(w(edge(node, fanout)) == 0){
            /* fanout in same cluster */
            if(fanout  $\notin$  visited){
                visit(fanout, visited, PI_list, PO_list, node_list);
            }
        }
    }
}

```

Figure 4.5: Clustering Combinational Logic Blocks: II

are in the same cluster if there is a path between them of weight 0. Algorithm `cluster` in Figures 4.4 and 4.5 describes the clustering process. On completion, `PI_list_array`, `PO_list_array`, `node_list_array`, will contain the lists of primary inputs, primary outputs and nodes; one for each cluster. These may be then used by a schematic generation routine or for interactive querying as discussed in the previous paragraph. Clustering begins by calling `visit` for a primary output node. `visit` recursively visits all nodes in the same cluster and updates the list of primary inputs, primary outputs and nodes for this cluster. The clustering is all done when there are no more primary outputs that have not been visited. Once clustering is complete the circuit may be retimed with the designer specifying the lag for each cluster.

4.2 Experimental Results: Area Optimization

4.2.1 Experimental Circuits

In the case of combinational logic optimization, a large suite of benchmark circuits has been accumulated by the research community and is available for distribution [47]. This is not the case for the relatively new field of sequential circuit optimization. Thus, as part of the experimental process a set of example circuits had to be collected. The circuits examined are from the following sources:

- Circuits synthesized from the FSM descriptions provided by the International Workshop on Logic Synthesis [47]. The FSM's are specified in terms of their state transition table. Circuit implementations are obtained by running the state assignment program NOVA [79].
- Circuits from speech ² and image recognition ³ chips.
- Controllers from an industrial source.
- Sequential circuits used as benchmark circuits in sequential circuit testing [16].

²These are from the circuit described in [75].

³This circuit is obtained from a design provided by G. DeMicheli from Stanford University.

4.2.2 Experimental Procedure and Results

In Section 4.1 it was observed that there are several possible sub-circuits that can be considered for retiming and resynthesis given an arbitrary sequential circuit. Since it is practically impossible to consider all possibilities, some choices need to be made to keep the search space limited. The choice of the lag for retiming node in the procedure `add_node` is an example of this. Another decision that needs to be made in procedure `seed_network` is the choice of the seed node. A desirable quality of the seed node is that the resulting sub-circuit should have nodes from across latch boundaries (in the original circuit) in the same combinational block so that logic relationships between them can be exploited. With this reasoning, a good choice for a seed is the input node to a latch. Since the latches are pushed radially outward from this point, nodes from both sides of the latch boundary get included in the same combinational logic block. Of course, there is typically more than one latch in the circuit and each latch input is a potential seed. Procedure `experiment` in Figure 4.6 outlines the experimental procedure that records the results of using each latch input as a seed. Combinational logic optimization is done using `misII` (This is version 2, release 1 of `MIS` [12]). Retiming to minimize latches is performed using the algorithm provided in [43].

The experimental results are not very encouraging. Only in one case is an area reduction of more than 5% observed in the combinational resynthesis. (This reduction is being measured with respect to a circuit in which the combinational logic and number of latches have been already minimized.) This example is an image processing circuit where a 6% reduction in the area was observed in the combinational part. The number of latches remain unchanged. The next section examines possible reasons for these results.

4.2.3 Analysis of Experimental Results

The lack of positive results in the previous section demands a look into why this is so. The following analysis aims to explain the experimental observations.

It is interesting to observe the combinational resynthesis step in terms of the topology of the peripherally-retimed circuit and the combinational optimization techniques used in `misII`.

First a look at the topology. It is common for primary inputs to fan out to a large number of gates in a combinational circuit. Since register outputs are primary inputs to the combinational logic, they share this high-fanout property. When the registers are moved to

```
/* inputs: network
   outputs: experimental results
*/

experiment(network){
    foreach_latch(network, latch){
        node = latch_input(latch);
        new_network = seed_network(network, node);

        /* optimize the area of the combinational logic */
        combinational_area_optimize(new_network);

        /* minimize latches using retiming */
        retime_min_latches(new_network);

        /* output the size of the circuit */
        output(combinational_area(new_network), num_latches(new_network));
    }
}
```

Figure 4.6: Summary of the Experimental Procedure

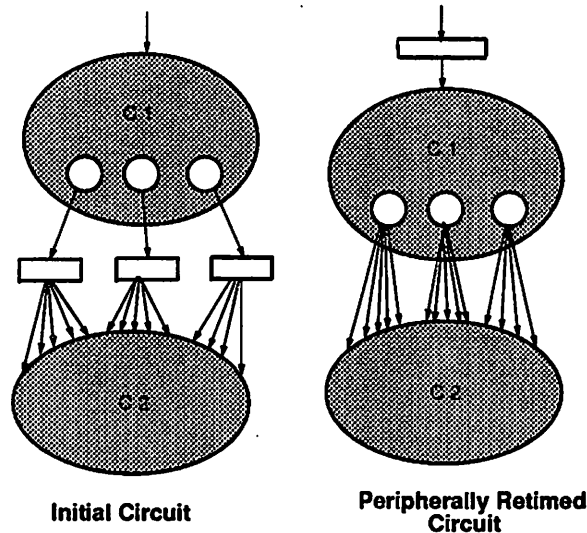


Figure 4.7: Register Outputs Form a High Fanout Cutset

the periphery, their previous input nodes have high-fanout (see Figure 4.7). Let us examine the implications of this on the optimization techniques included in *misII*.

misII has two main phases of optimization: circuit restructuring and node simplification. These are considered separately.

The circuit restructuring phase operates by collapsing (functional composition) low-fanout nodes into their fanout nodes and then restructuring these large resulting nodes using the algebraic techniques of cube and kernel extraction. Note that since the register input nodes have high-fanout in the peripherally-retimed circuits, these nodes form a high-fanout cutset of this network. This effectively restricts the restructuring to each side of the original latch boundaries. However, this restructuring has already been exploited and therefore nothing new is gained by migrating the latches. Completely collapsing the circuit, and thereby removing the barrier to restructuring, is computationally too expensive and impractical for reasonably-sized circuits.

The node simplification phase constructs the satisfiability don't care set [3] and simplifies each node using two-level minimization with this don't care set. Typically this don't care set is large and a filter is used that extracts only part of this don't care set for a

node [67]. This is determined by looking at the topology of the network. For the topology that we are working with, a network where the original register inputs form a cutset, the filtering would restrict the don't cares for a node to be generated only from other nodes that are on the same side of the cutset. Again this has already been exploited and nothing new is obtained from migrating the registers. Disabling the filter and using the complete satisfiability don't care set does not improve any of the results. This is partially to be expected since the filter has been designed so that the quality of the results is almost as good as what can be obtained with the entire satisfiability don't care set. Finally, node simplification using a subset of the observability don't cares for a node [68] is considered. Intuitively these don't cares should be very useful in this case since the observability of the nodes in sub-circuit C_1 (see Figure 4.7) is changed by adding sub-circuit C_2 and thus additional simplification is possible⁴. A possible reason as to why these are not effective is that only a subset of the observability don't cares are being used and this may not be sufficient.

Based on this it appears that the combinational optimization programs are trapped in a local minimum and are not powerful enough to exploit the additional information that is provided by the latch migration.

The following experiment helps substantiate these claims. Consider the example circuit in Figure 4.8(a). It consists of a two-stage pipeline. The first stage computes the sum of two n bit numbers, $a[n-1:0]$ and $b[n-1:0]$. The second stage checks if the $n+1$ bit sum is greater than 2^n . Figure 4.8(b) shows the peripheral retiming of this circuit. It is instructive to see how misII optimizes this circuit after it is given the additional information that the output of the adder is used only in the comparator. Two different optimization techniques are tried.

1. The standard misII script is used followed by simplification using a subset of the observability don't cares.
2. The circuit is first collapsed to two-level logic. This is then simplified using the two-level logic minimizer ESPRESSO. Technique 1 is then applied to the resulting circuit. The motivation behind collapsing the circuit to two-levels and using ESPRESSO is that two-level minimization is very powerful and can exploit the observability don't cares

⁴The nodes in C_1 were initially observable at the latches (for the combinational part). These observation points have been removed now.

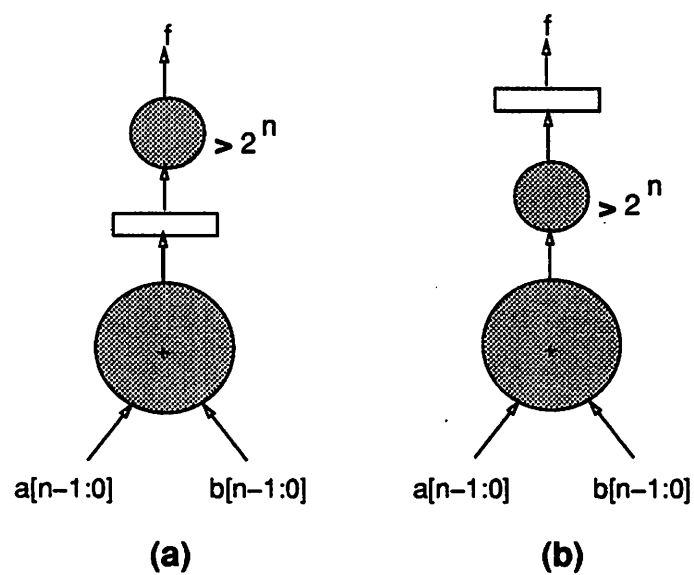


Figure 4.8: Example Circuit: *add_comp*

completely.

In order to evaluate how well the optimized circuits are doing, the circuit for the function $f = (a[n-1:0] + b[n-1:0]) > 2^n$ is separately designed. The following is a parametric description of this design.

$$\begin{aligned} x[i] &= x[i-1] + a[i] + b[i] \\ y[i] &= y[i-1] (a[i] + b[i]) + x[i-1] a[i] b[i] \\ x[-1] &= 0 \\ y[-1] &= 0 \\ f &= y[n-1] \end{aligned}$$

The semantics of $x[i]$ and $y[i]$ are as follows. Let $s[i]$ be the i^{th} sum bit and $c[i]$ be the carry out of the i^{th} bit of a ripple carry adder. $s[i]$ and $c[i]$ are given by:

$$\begin{aligned} s[i] &= a[i] \overline{b[i]} \overline{c[i-i]} + \overline{a[i]} b[i] \overline{c[i-i]} + \overline{a[i]} \overline{b[i]} c[i-i] + a[i] b[i] c[i-1] \\ c[i] &= a[i] b[i] + c[i-1] (a[i] + b[i]) \\ c[-1] &= 0 \end{aligned}$$

$x[i]$ is 1 if and only if $c[i] = 1$ or at least one $s[j] = 1$, $0 \leq j \leq i$. $y[i]$ is 1 if and only if $c[i] = 1$ and at least one $s[j] = 1$, $0 \leq j \leq i$.

This circuit is implemented using $8n - 8$ literals in factored form. This is in comparison to the adder followed by the comparator which is $\approx 13n$ literals.

Figure 4.9 compares the sizes of the circuits obtained using the different methods described here as a function of n . The curve labelled *adder + comp* represents the circuit obtained by optimizing the adder and the comparator separately and then combining them. The following observations are made from this graph:

1. Only for very small n does misII do a reasonable job when compared to the $8n - 8$ circuit. However, as n increases misII quickly runs out of steam and it does not reduce the size of the initial circuit.
2. Using ESPRESSO enables the observability don't cares to be used and as a result the quality is the same as the separately designed circuit. However, it is not practical to collapse the circuit to two levels for larger n .

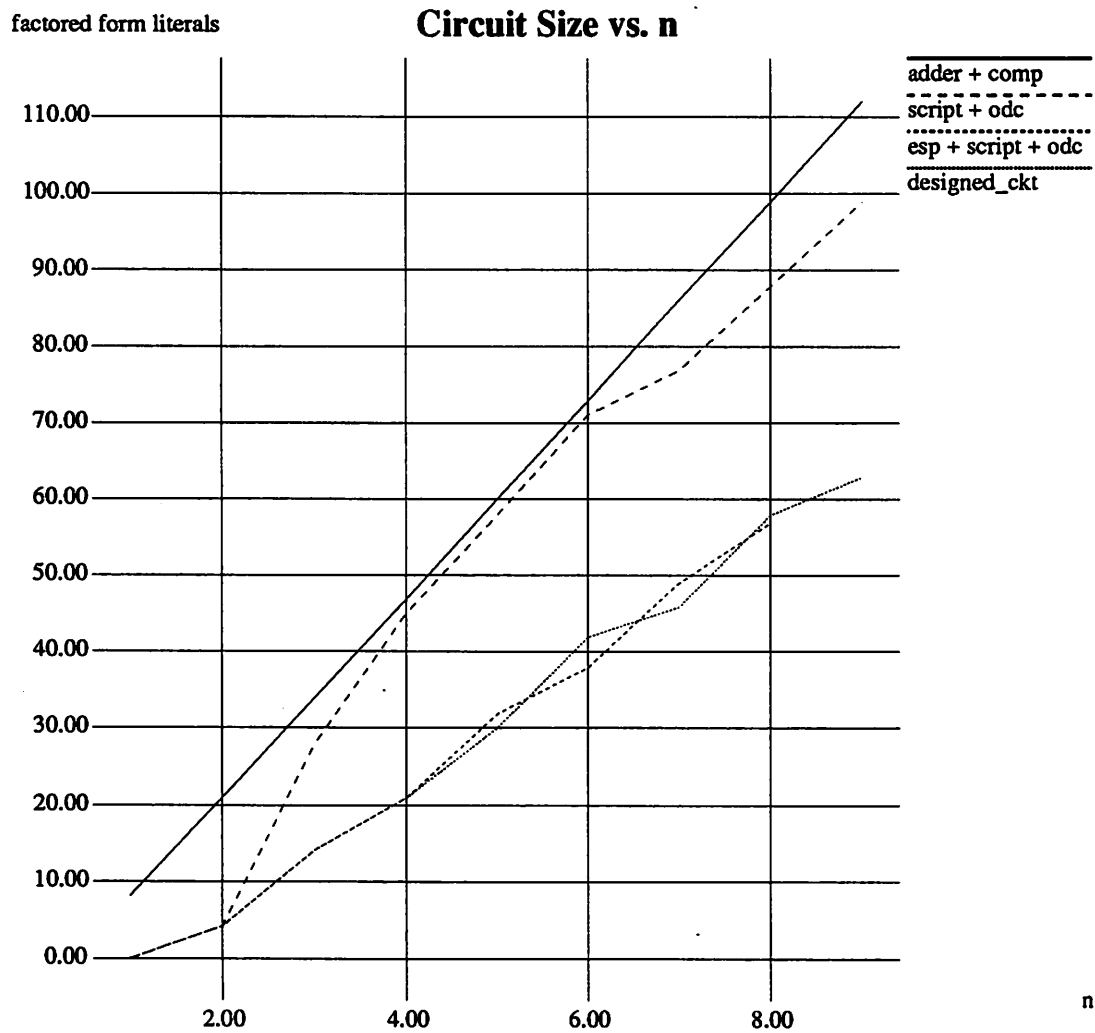


Figure 4.9: Experimental Results for *add_comp*

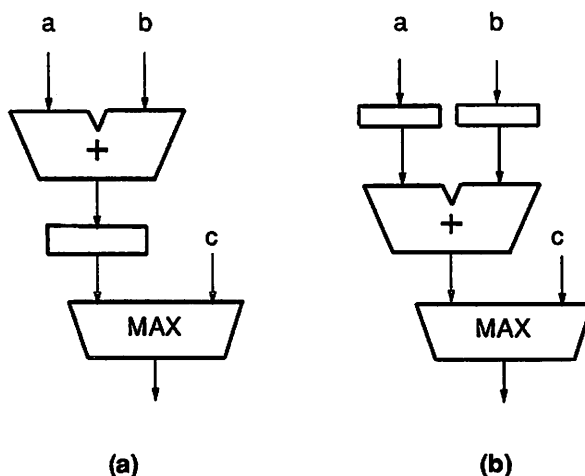


Figure 4.10: Example From a Datapath

Thus, it is seen that as the circuit size increases, *misII* is unable to exploit the additional information (that the adder is being followed only by a comparator) in any way. It should be noted that even for $n = 9$ the circuit sizes are relatively small (≈ 100 literals) in comparison to the circuits used in the area optimization experiments in this section. Thus it is likely that this limitation of *misII* is at least partially responsible for the lack of substantial improvements seen.

Current research in combinational logic optimization techniques (e.g. [60]) holds some promise in terms of discovering more powerful techniques that do circuit restructuring without collapsing and algebraic factoring. These are more likely to exploit the additional information generated by latch migration for the combinational logic optimizers.

Analysis of pipelined datapaths gives some insight as to when there is inherently no potential for further improvement and thus no use in expending any further effort. Consider the circuit in Figure 4.10(a). Here the function $\text{MAX}(a + b, c)$ is performed over two cycles, with the addition being done in the first stage and the selection of the maximum done in the second stage. No area improvement is obtained for this circuit. It is instructive to see why this is so. Figure 4.10(b) shows the same circuit with a peripheral retiming. Note that if $c = 0$, then the output of this circuit is $a + b$. Thus, even though the output of the adder

is not *explicitly* observable for this circuit, it is *implicitly* observable since it can be passed on to the output of the circuit by setting c to 0. Thus, no additional observability don't cares are generated by the cascade of the two logic blocks. We also note that the implicit observability of the adder outputs forces the adder IO-map to be a Boolean function as opposed to a Boolean relation [13], i.e. no two outputs of the adder are equivalent as far as the MAX logic block is concerned. Thus, there is no flexibility in changing the logic function of the adder block. It has been our experience that implicit observability is a general characteristic of datapath circuits and this property does not make them amenable to further area optimization using retiming and resynthesis techniques.

It is possible that for the example circuits there is no further potential for improvement using retiming and resynthesis techniques. However, since the set of examples is not fully representative of different kinds of sequential circuits, no generalizations can be made.

4.3 Experimental Results: Performance Optimization

4.3.1 Example Circuits and Experimental Procedure

Since the theoretical results in performance optimization were developed specifically for pipelined circuits, the experiments to evaluate their practical utility are also conducted on pipelined circuits. Only one of the circuits among those described in Section 4.2.1 is pipelined, thus additional circuits are needed for this experiment. The following three arithmetic circuits were designed for this purpose. Each of these is a two-stage pipelined circuit.

- ex1** A two-stage adder that adds four 8-bit numbers (A, B, C and D). The first stage computes the partial sums $A + B$ and $C + D$ and the second stage computes the final sum. Each adder is a ripple-carry adder.
- ex2** A two stage adder that adds two 16-bit numbers. The first stage computes the sum of the 8 least-significant bits and the second stage computes the final sum. Each adder is a ripple-carry adder.
- ex3** This circuit computes the parity of the sum of two 8-bit numbers (A and B). During the first stage the sum is generated using a ripple-carry adder. In the next stage the

Name	Rsyn-Ret			Ret-Rsyn			PR-Rsyn-Ret		
	area	latches	cycle	area	latches	cycle	area	latches	cycle
ex1	804	27	14.4	518	28	15.4	758	25	14.4
ex2	500	25	16.6	542	22	15.4	571	42	11.0
ex3	292	9	12.6	265	14	14.0	371	22	12.4

Table 4.1: Experimental Results: Performance Optimization of Pipelined Circuits

parity of the sum is computed using a balanced parity tree.

Three design scenarios are evaluated and the cycle time achieved by each is reported in Table 4.1. The three scenarios explore different methods to obtain a faster version of the initial pipelined circuit and are as follows:

1. resynthesis followed by retiming (Rsyn-Ret)
2. retiming followed by resynthesis (Ret-Rsyn)
3. the approach proposed in Section 3.3, i.e., peripheral retiming followed by resynthesis followed by retiming (PR-Rsyn-Ret).

For purposes of this experiment only one delay optimization routine is applied, the critical-path restructuring on a technology-independent network [73], as part of the resynthesis procedure. The delay through the circuit is measured using a two-input NAND gate representation of the circuit. Each gate contributes one unit of delay and each fanout contributes an additional delay (0.2 units in this experiment). The area of the combinational part is measured as the number of literals (gate input connections) in the same 2-input NAND representation.

From the results in Table 4.1 we make the following observations.

1. The order of retiming and resynthesis operations impacts the value of cycle time that can be achieved. Neither order can be counted on to be the best for all circuits.
2. The cycle time obtained by the proposed method matches or is better than the best result that can be obtained by any combination of a single retiming and a single

resynthesis step. This is what is expected theoretically and it is gratifying that this is achieved in practice as well. It is clear from circuit *ex2* that the additional flexibility gained from looking at the maximal combinational logic sub-network obtained by peripheral retiming provides the optimization techniques greater freedom in restructuring the circuit to reduce the cycle time.

In the current experiments there is an increase in the number of latches, over the initial number of latches, when the proposed method is used. This is due to the fact that no attempt is made to minimize the latches during retiming and also due to the particular resynthesis technique used. The critical path restructuring increases the width of the circuit and hence more latches are used.

Part II

Handling Symbolic Inputs

Chapter 5

Multi-Level Logic Minimization

“What a curious feeling!” said Alice, I must be shutting up like a telescope!”

And so it was indeed: she was now only ten inches high, and her face brightened up at the thought that she was now the right size for going through the little door into that lovely garden.

– Lewis Carroll, “Alice in Wonderland”

Symbolic variables and the encoding problems associated with them were introduced in Chapter 1. This part of the thesis considers the input encoding problem for area minimization. In addition to being a problem in its own right, it is also an approximation to the state assignment problem in sequential logic synthesis. Input encoding-based techniques can be applied to state assignment by treating the state variable as an input to the logic that computes the next state and output functions. The fact that it is also the output of this description is not taken into account. This approximation is valid when the output logic is significantly larger than the next state logic.

For the case of two-level logic, satisfactory solutions to the input encoding problem were obtained by first using multiple-valued (MV) two-level logic minimization (e.g. [64]) and then using the result of this to generate constraints that the encoding must satisfy (e.g. [59, 79]). The advantage of doing this is that multiple-valued logic minimization does not depend on any choice of encoding and captures the effects of all possible encodings. Thus, deferring the encoding until after minimization avoids restricting the minimizer to only one encoding. However, for multi-level logic as the target implementation, the approaches currently used tend to be “predictive” in that they determine encodings for which a multi-level logic optimizer such as MIS [12] is likely to find common divisors (e.g. [25, 45, 34])

and thus result in circuits with smaller size. Unlike the two-level case, the multi-level logic optimizer is used only after the encoding has been selected. This asymmetry between the approaches for two and multi-level logic arose from the fact that multi-level multiple-valued minimization techniques had not been developed. The work presented in this chapter attempts to fill this gap. It presents techniques for multi-level optimization of logic with MV input variables.

5.1 Multi-Level Optimization Techniques

Let us first examine the general paradigm used in the area optimization of multi-level circuits with Boolean (or binary-valued) inputs. Multi-level area optimization is a collection of different techniques, each of which attempts to reduce the circuit size by applying some function preserving transformation on the given circuit. This section examines the techniques that have been most successful.

Empirically it has been observed that the largest impact in terms of reducing the circuit size is by using circuit decomposition. This involves introducing new functions in the circuit and expressing other functions in the circuit in terms of them. Determining good decompositions is a very difficult task. Brayton and McMullen presented a technique [10] which determines circuit decompositions by first considering functions as algebraic polynomials and then finding common sub-expressions in these polynomials. Each sub-expression can then be implemented as a separate function and used repeatedly in each original occurrence. The common sub-expressions are also referred to as common factors or divisors. While this is an approximation to the real problem, it is fast and produces results of acceptable quality. These common sub-expressions are classified into two groups: common cubes, which are expressions consisting of a single product term, and common kernels, which give rise to common sub-expressions of more than one product term. This classification arises because different techniques are used for detecting these two types of divisors.

Since algebraic decomposition techniques do not exploit any Boolean relationships that exist between the different circuit components, further simplification is possible by using these relationships. Boolean relationships have been exploited in different ways by different programs. In MIS [12], this is done by considering each gate function in two-level form and then simplifying this using a two-level minimizer. The relationships between the different gate functions in the circuit are captured through the *implicit don't cares* extracted

from the circuit [3]. In BOLD [8] connections are added to and deleted from gates as long as this preserves logical behavior. In LSS [7] this is accomplished using a technique referred to as *global flow*. The transduction method [60] uses the notion of permissible functions towards this end.

5.2 Overview

Of the techniques introduced in Section 5.1, the one that is most difficult to extend to circuits with symbolic (or MV) inputs is circuit decomposition. The major contribution of this chapter is to describe such a technique. It demonstrates how common sub-expression extraction can be extended to circuits with symbolic inputs while guaranteeing some optimality for the final encoded circuit. This completes the missing link in multi-level multiple-valued minimization since the other techniques used in multi-level synthesis can be easily extended to handle MV inputs, as will be shown later in this chapter.

The following sections examine each of these optimization techniques. In each case these techniques are first briefly explained for binary-valued variables and then it is shown how these can be extended to handle MV variables.

5.3 Circuit and Function Representation

Let a symbolic variable v take values from $V = \{v_0, v_1, \dots, v_{n-1}\}$. v may be represented by a MV variable, X , restricted to $P = \{0, 1, \dots, n-1\}$, where each symbolic value of v maps onto a unique integer in P ¹. We can map several MV variables into a single MV variable as follows. Let v_1 and v_2 be two symbolic variables taking values from sets V_1 and V_2 respectively. These may be replaced by a single symbolic variable v taking values from $V_1 \times V_2$. This is in fact potentially better than considering v_1 and v_2 separately since the encoding for v takes into account the interactions between v_1 and v_2 . This enables us to restrict ourselves to a single symbolic variable for the rest of this chapter.

Let $B = \{0, 1\}$. A binary-valued function f , of a single MV variable X and $m-1$ binary-valued variables, is a mapping: $f : P \times B^{m-1} \rightarrow B$. Each element in the domain of the function is called a *minterm* of the function. Let $S \subseteq P$. Then X^S represents the

¹The notation presented in this section is the same as that used in two-level multiple-valued minimization [64].

binary-valued function:

$$X^S = \begin{cases} 1 & \text{if } X \in S \\ 0 & \text{otherwise} \end{cases}$$

X^S is called a *literal* of variable X . If $|S| = 1$ then this literal is also a minterm of X . For example, $X^{\{0\}}$ and $X^{\{0,1\}}$ are literals and $X^{\{0\}}$ a minterm of X . If $S = \phi$, then the value of the literal is always 0. If $S = P$ then the value of the literal is always 1. For these two cases, the value of the literal may be used to denote the literal. We note the following:

1. $X^{S_1} \subseteq X^{S_2}$ if and only if $S_1 \subseteq S_2$
2. $X^{S_1} \cup X^{S_2} = X^{S_1 \cup S_2}$
3. $X^{S_1} \cap X^{S_2} = X^{S_1 \cap S_2}$

The literal of a binary-valued variable y is defined as either the variable or its Boolean complement. A *product term* or a *cube* is a Boolean product (AND) of literals. If a cube evaluates to 1 for a given minterm, it is said to contain the minterm. A *sum-of-products* (SOP) is a Boolean sum (OR) of product terms. For example: $X^{\{0,1\}}y_1y_2$ is a cube and $X^{\{0,1\}}y_1y_2 + X^{\{3\}}y_2y_3$ is an SOP. A function f may be represented by an SOP expression f . In addition f may be represented as a factored form. A factored form is defined recursively as follows.

Definition 5.3.1 *An SOP expression is a factored form. A sum of two factored forms is a factored form. A product of two factored forms is a factored form.*

$X^{\{0,1,3\}}y_2(X^{\{0,1\}}y_1 + X^{\{3\}}y_3)$ is a factored form for the SOP expression given above.

A logic circuit with a multiple-valued input is represented as an MV-network. This is illustrated in Figure 5.1. An *MV-network* η , is a directed acyclic graph (DAG) such that for each node n_i in η there is associated a binary-valued, MV input function f_i , expressed in SOP form, and a binary-valued variable y_i which represents the output of this node. There is an edge from n_i to n_j in η if f_j explicitly depends on y_i . Further, some of the variables in η may be classified as *primary inputs* or *primary outputs*. These are the inputs and outputs (respectively) of the MV-network. The MV-network is an extension of the well-known Boolean network [12] to permit MV input variables; in fact the latter reduces to the former when all variables have binary values. Since each node in the network has a binary-valued output, the non-binary(MV) inputs to any node must be primary inputs to

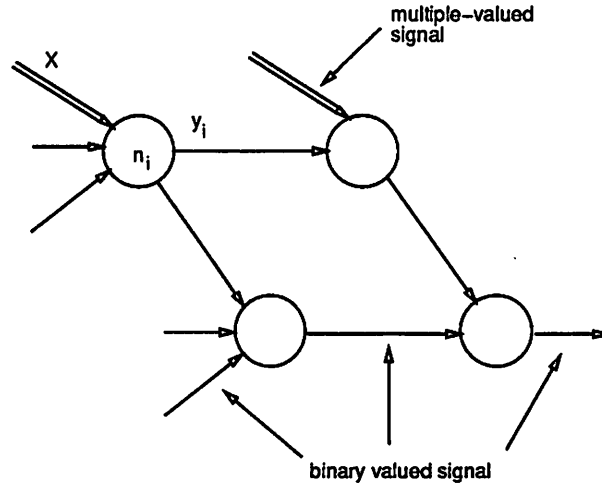


Figure 5.1: Representing Circuits as MV-networks

the network. The MV-network computes logical functions in the natural way. Each node in the DAG computes some function, the result of which is used in all the nodes to which an edge exists from this node.

In the sequel, the naming convention used in this section for functions and variables will continued. Functions names are bold lower case letters, binary-valued variables and expressions are named with regular lower case letters, and the upper case letter X is reserved for the MV variable.

It should be noted that multiple-valued functions of multiple-valued variables (referred to as discrete mappings) are not Boolean functions, nor are expressions representing them Boolean expressions. However, properties of these functions and expressions are similar to those of Boolean functions and expressions. Appendix A describes the relationship between discrete mappings and Boolean functions and explains why properties of Boolean expressions apply to expressions representing discrete mappings.

5.4 Circuit Decomposition Using Kernels

This section presents multiple-valued decomposition using kernels along with its interesting properties with respect to the final encoded circuit. In this direction, the process of common sub-expression extraction when there are no MV variables is reviewed first.

5.4.1 Kernels and Kernel Intersections

Common sub-expressions consisting of multiple cubes can be extracted from expressions of binary-valued variables using the techniques described in [10]. These techniques are referred to as *algebraic* since they treat expressions representing logic functions as multilinear monomials with unit coefficients over the variables $\{y_1, \bar{y}_1, \dots, y_n, \bar{y}_n\}$. This permits efficient factorization and decomposition, though at the cost of optimality since the Boolean identities, $y_1 \cdot y_1 = y_1$, $y_1 \cdot \bar{y}_1 = 0$ are not used. Some definitions presented in [10] are reviewed first.

Definition 5.4.1 *An expression f is said to be **cube-free** if no cube divides all the cubes in f .*

The term ‘division’ here refers to algebraic division. For example, $f_4 = y_1 y_2 + y_1 y_3$ is not cube free since the cube y_1 divides both cubes in f_4 . $f_5 = y_1 y_2 + y_3$ is cube free. By convention 0 and 1 are not cube free. In the sequel, unless otherwise mentioned, division refers to algebraic division.

Definition 5.4.2 *A kernel, k , of an expression f is a cube-free quotient of f and a cube c . A co-kernel associated with a kernel is the cube divisor used in obtaining that kernel.*

As an example, consider the expression $f_5 = y_1 y_4 + y_2 y_4 + y_3$ and the cube y_4 . The quotient of f_5 and this cube, f_5/y_4 , is $y_1 + y_2$. No other cube is a factor of $y_1 + y_2$, hence it is a kernel of f_5 . The cube y_4 is the co-kernel used to derive this kernel. In general, the co-kernel of a kernel is not unique.

SOP expressions may be alternatively viewed as sets of cubes (and vice versa). This lets us define the intersection of two kernels k_1 and k_2 in the natural way as the set of cubes present in both k_1 and k_2 . The following key result from [10] relates kernel intersections and common sub-expressions.

Theorem 5.4.1 (Brayton and McMullen) *Two expressions have a common divisor of more than one cube if and only if they have a kernel intersection of more than one cube.*

This kernel intersection can then be used to find the common divisor. Thus, we can detect all multiple-cube common sub-expressions by finding all multiple-cube kernel intersections. In [11] algorithms for detecting kernel intersections are described by defining them in terms of the rectangular covering problem. Because of the ease in understanding the concepts involved, the rectangular covering approach is used for developing the ideas in the rest of this chapter. However, they hold with any technique for kernel extraction.

The rectangular covering formulation is explained through the following example. Consider the expressions f_7 and f_8 :

$$\begin{array}{rcl}
 f_7 & = & y_1 y_5 + y_2 y_5 + y_3 \\
 & & \quad 1 \qquad \quad 2 \qquad \quad 3 \\
 f_8 & = & y_1 y_6 + y_2 y_6 + y_4 \\
 & & \quad 4 \qquad \quad 5 \qquad \quad 6
 \end{array}$$

The integer below each cube is its unique identifier used later to refer to this cube. The kernels of f_7 and f_8 are:

<i>expression</i>	<i>co - kernel</i>	<i>kernel</i>
f_7	1	$y_1 y_5 + y_2 y_5 + y_3$
f_7	y_5	$y_1 + y_2$
f_8	1	$y_1 y_6 + y_2 y_6 + y_4$
f_8	y_6	$y_1 + y_2$

The table below is the *co-kernel cube matrix* for the set of expressions. It lets us view all kernels simultaneously and detect kernel intersections. A row in the matrix corresponds to a kernel, whose co-kernel is the label for that row. Each column corresponds to a cube which is the label for that column. A non-zero entry in the matrix specifies the integer identifier of the cube represented by the entry. This is the cube in the original expressions

obtained by intersecting the row and column labels.

	y_1	y_2	$y_1 y_5$	$y_2 y_5$	$y_1 y_6$	$y_2 y_6$	y_3	y_4
1	0	0	1	2	0	0	3	0
y_5	1	2	0	0	0	0	0	0
1	0	0	0	0	4	5	0	6
y_6	4	5	0	0	0	0	0	0

A rectangle \mathcal{R} is a sub-matrix of the co-kernel cube matrix. It is defined as a set of rows $S_r = \{r_0, r_1, \dots, r_{m-1}\}$ and a set of columns $S_c = \{c_0, c_1, \dots, c_{n-1}\}$ such that for each $r_i \in S_r$ and each $c_j \in S_c$, the (r_i, c_j) entry of the co-kernel cube matrix is non-zero. \mathcal{R} covers each such entry. \mathcal{R} is denoted as: $\{R(r_0, r_1, \dots, r_{m-1}), C(c_0, c_1, \dots, c_{n-1})\}$. Observe that each rectangle that has more than one row indicates a kernel intersection between the kernels corresponding to the rows in the rectangle. The columns in the rectangle specify the cubes of the kernel intersection. For example, $\{R(2, 4), C(1, 2)\}$, indicates the kernel intersection $y_1 + y_2$ between the second and fourth kernels.

A rectangular covering of the matrix is defined as a set of rectangles that cover the non-zero integers in the matrix at least once (and do not cover a 0 entry). Once an integer is covered, all its other occurrences are don't cares (they may or may not be covered by other rectangles). A covering for the above co-kernel cube matrix is: $\{R(2, 4), C(1, 2)\}$, $\{R(1), C(7)\}$, $\{R(3), C(8)\}$. A rectangular covering suggests a factorization of the original set of expressions. If common factors are extracted and implemented separately then this is referred to as a decomposition. The resulting implementation suggested by this covering is:

$$f_7 = y_5 y_9 + y_3$$

$$f_8 = y_6 y_9 + y_4$$

$$f_9 = y_1 + y_2$$

Using the well-accepted metric of circuit size, *viz.* the total number of literals in the factored form of all the expressions [12], the above description has two fewer literals than the original description. With larger factors the size reduction is significantly higher.

5.4.2 Kernels and Multiple-Valued Variables

Now consider the case in which one of the input variables may be multiple-valued. The following example has a single MV variable X with six values and six binary-valued variables.

$$\begin{array}{rcl}
 f_7 & = & X^{\{0,1\}} y_1 y_5 + X^{\{2\}} y_2 y_5 + y_3 \\
 & & \quad \quad \quad 1 \qquad \qquad \quad 2 \qquad \qquad \quad 3 \\
 f_8 & = & X^{\{3,4\}} y_1 y_6 + X^{\{5\}} y_2 y_6 + y_4 \\
 & & \quad \quad \quad 4 \qquad \qquad \quad 5 \qquad \qquad \quad 6
 \end{array}$$

Again, the integers below each cube are unique identifiers for that cube.

The definitions and matrix representations given in Section 5.4.1 are for binary-valued expressions of binary-valued variables. These are now extended to binary-valued expressions with one MV variable. (As in Section 5.3, only one of the variables is considered to be MV, the others are binary-valued.)

Kernels and co-kernels are defined as in the case with binary-valued variables. The co-kernel cube matrix is modified as follows. Each row represents a kernel (labelled by its co-kernel cube) and each column a cube (labelled by this cube). The column cubes contain literals of binary-valued variables only. Each non-zero entry in the matrix now has two parts. The first part is the integer identifier of the cube in the original expressions (*cube part*). The second part is the MV literal (*MV part*). The MV part ANDed with the cube corresponding to its column and the co-kernel corresponding to its row forms the cube specified by the integer identifier in the cube part.

The kernels of f_7 and f_8 are:

<i>expression</i>	<i>co - kernel</i>	<i>kernel</i>
f_7	1	$X^{\{0,1\}} y_1 y_5 + X^{\{2\}} y_2 y_5 + y_3$
f_7	y_5	$X^{\{0,1\}} y_1 + X^{\{2\}} y_2$
f_8	1	$X^{\{3,4\}} y_1 y_6 + X^{\{5\}} y_2 y_6 + y_4$
f_8	y_6	$X^{\{3,4\}} y_1 + X^{\{5\}} y_2$

The corresponding co-kernel cube matrix is:

	y_1	y_2	$y_1 \ y_5$	$y_2 \ y_5$	$y_1 \ y_6$	$y_2 \ y_6$	y_3	y_4
1	0	0	1	2	0	0	3	0
	0	0	$X^{0,1}$	X^{2}	0	0	1	0
y_5	1	2	0	0	0	0	0	0
	$X^{0,1}$	X^{2}	0	0	0	0	0	0
1	0	0	0	0	4	5	0	6
	0	0	0	0	$X^{3,4}$	X^{5}	0	1
y_6	4	5	0	0	0	0	0	0
	$X^{3,4}$	X^{5}	0	0	0	0	0	0

In the co-kernel cube matrix the cube part is given above the MV part for each entry. The adjectives MV and binary will be used with co-kernel cube matrices and rectangles in order to distinguish between the multiple-valued and the binary-valued case.

A rectangle is defined as in the case for all binary variables with one modification. Now the rectangle is permitted to have zero entries also. Unlike the binary-valued case, a rectangle here does not necessarily result in a common factor. It needs to satisfy some additional conditions which are now considered.

Associated with each rectangle \mathcal{R} is a *constraint matrix* $M^{\mathcal{R}}$ whose entries are the MV parts of the entries of \mathcal{R} . For example, for the MV co-kernel cube matrix given above, M_1 , is the constraint matrix for the rectangle $\{R(2,4), C(1,2)\}$.

$$M_1 = \begin{bmatrix} X^{0,1} & X^{2} \\ X^{3,4} & X^{5} \end{bmatrix}$$

A constraint matrix is said to be *satisfiable* if:

$$M(i, j) = [\cup_i M(i, j)] \cap [\cup_j M(i, j)] \quad \forall i, j \quad (5.1)$$

i.e. if a particular value of X occurs somewhere in row i and also somewhere in column j , then it must occur in $M(i, j)$. M_1 given above is satisfiable. If a constraint matrix

is satisfiable then it can be used to determine a common factor between the expressions corresponding to its rows as follows. The union of the row entries for row i , $(\cup_j M(i, j))$, is ANDed with the co-kernel cube corresponding to row i . Similarly the union of the column entries for column j , $(\cup_i M(i, j))$, is ANDed with the kernel cube corresponding to that column. This results in the following factorization of the expressions f_7 and f_8 .

$$\begin{aligned} f_7 &= X^{\{0,1,2\}} y_5 (X^{\{0,1,3,4\}} y_1 + X^{\{2,5\}} y_2) + y_3 \\ f_8 &= X^{\{3,4,5\}} y_6 (X^{\{0,1,3,4\}} y_1 + X^{\{2,5\}} y_2) + y_4 \end{aligned}$$

Note that there is now a common factor between the two expressions which was not evident to start with. This common factor may now be implemented as a separate node in the MV-network and its output used to compute f_7 and f_8 as follows:

$$\begin{aligned} f_7 &= X^{\{0,1,2\}} y_5 y_9 + y_3 \\ f_8 &= X^{\{3,4,5\}} y_6 y_9 + y_4 \\ f_9 &= X^{\{0,1,3,4\}} y_1 + X^{\{2,5\}} y_2 \end{aligned}$$

Not all matrices are satisfiable. An example of this is as follows:

$$M_2 = \begin{bmatrix} X^{\{1\}} & X^{\{2\}} & X^{\{5\}} \\ X^{\{4\}} & X^{\{5\}} & X^{\{1\}} \end{bmatrix}$$

M_2 is not satisfiable since $X^{\{5\}}$ occurs in row 1 as well in column 2 but is not present in $M(1, 2)$.

Let $M^{\mathcal{R}}$ be a satisfiable matrix. Let ψ_{ij} be the cube indicated by the cube part of entry (r_i, c_j) in \mathcal{R} . Let Λ_i be the row label (i.e. the co-kernel) for row r_i and Γ_j be the column label for column c_j . Thus, $\psi_{ij} = \Lambda_i \Gamma_j M(i, j)$. The set of cubes $\sum_j \psi_{ij}$ is part of some expression in the original set of expressions. This may be expressed as the following factored form:

$$\sum_j \psi_{ij} = [\cup_j M(i, j)] \Lambda_i \left(\sum_j ([\cup_i M(i, j)] \Gamma_j) \right) \quad (5.2)$$

The factor $(\sum_j ([\cup_i M(i, j)] \Gamma_j))$ is independent of and thus common to all i .

Let us now see why M must be satisfiable.

Theorem 5.4.2 *The factorization specified by Equation 5.2 is valid if and only if M is satisfiable.*

Proof.

If part: This follows by multiplying the right hand side and comparing the corresponding terms on each side.

Only if part: The proof is by contradiction. Let M be a non-satisfiable matrix and suppose that Equation 5.2 is valid. Let (i, j) be an entry of M for which the condition in Equation 5.1 is not met. On expanding out the factored form on the right hand side of Equation 5.2, the term corresponding to ψ_{ij} is $\Lambda_i \Gamma_j ([\cup_i M(i, j)] \cap [\cup_j M(i, j)])$. Since $M(i, j) \neq ([\cup_i M(i, j)] \cap [\cup_j M(i, j)])$, we get $\Lambda_i \Gamma_j ([\cup_i M(i, j)] \cap [\cup_j M(i, j)]) \neq \psi_{ij}$. Thus, Equation 5.2 is not valid. ■

Satisfiable constraint matrices are not the only source of common factors. In fact the condition can be relaxed as we see below.

Definition 5.4.3 M_r is a reduced constraint matrix of M if $\forall i, j \quad M_r(i, j) \subseteq M(i, j)$.

Note that this definition includes the original constraint matrix. An example of a reduced constraint matrix for M_1 is:

$$M_3 = \begin{bmatrix} X^{\{1\}} & X^{\{2\}} \\ X^{\{3,4\}} & X^{\{5\}} \end{bmatrix}$$

A reduced constraint matrix that is satisfiable can be used to generate a common factor by covering the remaining entries separately. For example, with the original expressions and M_3 we can obtain the following factorization.

$$\begin{aligned} f_7 &= X^{\{1,2\}} y_5 (X^{\{1,3,4\}} y_1 + X^{\{2,5\}} y_2) + X^{\{0\}} y_1 y_5 + y_3 \\ f_8 &= X^{\{3,4,5\}} y_6 (X^{\{1,3,4\}} y_1 + X^{\{2,5\}} y_2) + y_4 \end{aligned}$$

Note that $X^{\{0\}} y_1 y_5$ must be covered separately.

Thus far only the way in which factors may be extracted in MV-networks has been described. Algorithms for determining kernels as well as for rectangular covering have not been discussed. These have been described in detail in [11, 63]. Typically at any time in the circuit decomposition process there is a choice to be made between several possible factors. One particular choice may restrict the choices available in the future. This is a general problem even in the case of Boolean networks. In that case a locally optimal decision is made based on the estimated gain of the factor. This is based on the size of the factor and

the number of places where it is used. The situation is more complicated in MV-networks since the size of the factor is not known until encoding. This makes it difficult to estimate its potential gain. The discussion regarding how this is handled is deferred to the next chapter. Some of the problems, that arise because of the difficulty in predicting the size of MV-literals after encoding, are handled using the concept of *incompletely specified literals*, which is now explained. However, first a few terms need to be formally defined that have thus far been used in an intuitive way.

Encodings and Encoded Implementations

Definition 5.4.4 *An encoding E , of the values of an MV variable X is a mapping of each distinct value of X to a distinct vertex in some q -dimensional Boolean space, B^q . The encoded expression of a minterm X^α , under E , denoted by $\mathcal{E}(X^\alpha, E)$, is the singleton set containing the vertex in B^q that X^α is mapped to. The encoded expression of a literal of X is the union of the encoded expressions of its constituent minterms.*

As an example, consider the following encoding, E_1 , for the values of X :

$$\begin{aligned} X^{\{0\}} : 000 & \quad X^{\{1\}} : 100 & X^{\{2\}} : 001 \\ X^{\{3\}} : 110 & \quad X^{\{4\}} : 010 & X^{\{5\}} : 011 \end{aligned}$$

The variables s_0, s_1 and s_2 are used for this 3 dimensional Boolean space. Here $\mathcal{E}(X^{\{0\}}, E_1) = \{\bar{s}_0\bar{s}_1\bar{s}_2\}$ and $\mathcal{E}(X^{\{0,1\}}, E_1) = \{\bar{s}_0\bar{s}_1\bar{s}_2, s_0\bar{s}_1\bar{s}_2\}$. A set of points in B^q may be alternatively represented as a sum-of-products expression equivalent to the vertices in this set. For example $\mathcal{E}(X^{\{0,1\}}, E_1)$ may be expressed as $\bar{s}_0\bar{s}_1\bar{s}_2 + s_0\bar{s}_1\bar{s}_2$ or equivalently as $\bar{s}_1\bar{s}_2$. Vertices in B^q that are not images of any value of X are *don't care* vertices. For example, $s_0\bar{s}_1s_2$ and $s_0s_1s_2$ are don't care vertices for E_1 . They may be included in any encoded expression for simplification.

Note that \mathcal{E} is an invertible function. Given any set of points in B^q (or an expression equivalent to these points) there is a unique literal of X that maps to them under \mathcal{E} . Thus, $\mathcal{E}^{-1} : B^q \rightarrow P$.

The notion of an encoded expression is now extended to handle sets of expressions that represent functions.

Definition 5.4.5 *An encoded implementation of a set of expressions $\{f_1, f_2, \dots, f_n\}$ under encoding E , where each f_i represents a $P \times B^{m-1}$ function, is a set of expressions*

$\{\tilde{f}_1, \tilde{f}_2, \dots, \tilde{f}_n\}$ where each \tilde{f}_i has been obtained from f_i by replacing each MV literal in f_i by its encoded expression for E . It is denoted by $\mathcal{E}(\{f_1, f_2, \dots, f_n\}, E)$.

For an MV-network η , the encoded implementation, $\mathcal{E}(\eta, E)$ for encoding E is obtained by replacing the set of node function expressions by their encoded implementation. As before \mathcal{E}^{-1} is the inverse function that can be used to obtain the original circuit from the encoded implementation.

Incompletely Specified Literals

The notion of *incompletely specified literals* is introduced through an example. Consider the following expressions in factored form:

$$\begin{aligned} f_7 &= X^{\{0,1,2\}} y_5 (X^{\{0,1,3,4\}} y_1 + X^{\{2,5\}} y_2) + y_3 \\ f_8 &= X^{\{3,4,5\}} y_6 (X^{\{0,1,3,4\}} y_1 + X^{\{2,5\}} y_2) + y_4 \\ f_9 &= X^{\{6\}} \end{aligned}$$

Note that f_7 could be modified as follows without changing the functional behavior.

$$f_7 = X^{\{0,1,2,6\}} y_5 (X^{\{0,1,3,4\}} y_1 + X^{\{2,5\}} y_2) + y_3$$

Replacing $X^{\{0,1,2\}}$ with $X^{\{0,1,2,6\}}$ leaves the function unchanged since the multiplicative factor, $(X^{\{0,1,3,4\}} y_1 + X^{\{2,5\}} y_2)$, does not have $X^{\{6\}}$ anywhere in it. Thus, using a term from two-level minimization, $X^{\{0,1,2\}}$ may be *expanded* to $X^{\{0,1,2,6\}}$. With binary variables, expansion always results in a decrease in the number of literals in the circuit size since expanding a literal y_1 or \bar{y}_1 results in the removal of this literal. However, expansion of MV-literals does not result in their removal unless the expansion is to the literal 1. Thus, it is not clear if expansion is useful in decreasing circuit size. Actually, as shown below, it depends on the encoding chosen. Consider the encoded expressions of the unexpanded and expanded literals with the following encoding, E_1 .

$$\begin{aligned} X^{\{0\}} &: 000 & X^{\{1\}} &: 100 & X^{\{2\}} &: 001 \\ X^{\{3\}} &: 110 & X^{\{4\}} &: 010 & X^{\{5\}} &: 011 \\ X^{\{6\}} &: 101 \end{aligned}$$

$$\begin{aligned} \mathcal{E}(X^{\{0,1,2\}}, E_1) &= \bar{s}_1 \bar{s}_2 + \bar{s}_1 \bar{s}_0 \\ \mathcal{E}(X^{\{0,1,2,6\}}, E_1) &= \bar{s}_1 \end{aligned}$$

With the following encoding, E_2 , we get:

$$\begin{aligned} X^{\{0\}} &: 000 & X^{\{1\}} &: 100 & X^{\{2\}} &: 001 \\ X^{\{3\}} &: 110 & X^{\{4\}} &: 010 & X^{\{5\}} &: 011 \\ X^{\{6\}} &: 111 \end{aligned}$$

$$\begin{aligned} \mathcal{E}(X^{\{0,1,2\}}, E_2) &= \bar{s}_1 \\ \mathcal{E}(X^{\{0,1,2,6\}}, E_2) &= \bar{s}_1 + s_0 s_2 \end{aligned}$$

Thus, with E_1 expansion led to a smaller encoded implementation while with E_2 it led to a larger encoded implementation.

Since it is not possible to predict the effect of expansion until encoding, it is best if the decision to expand is deferred to the encoding step. This can be captured by permitting the MV-literal to be incompletely specified. An incompletely specified MV-literal, $X^{S_1[S_2]}$, represents the incompletely specified pseudo-function ²:

$$X^{S_1[S_2]} = \begin{cases} 1 & \text{if } X \in S_1 \\ 0 & \text{if } X \notin S_1 \cup S_2 \end{cases}$$

The value of of this pseudo-function is unspecified when $X \in S_2$. This permits f_7 to be expressed as:

$$f_7 = X^{\{0,1,2\}[6]} y_5 (X^{\{0,1,3,4\}} y_1 + X^{\{2,5\}} y_2) + y_3$$

This makes it convenient to express the fact the expansion is optional and the decision whether it should be done is left till the encoding step.

The discussion as to how the incomplete specification is practically determined and used is deferred to the next chapter.

Main Results

Let us now see what the MV-factorization process gains us in terms of the final encoded circuit. We are interested in obtaining large common factors in the encoded implementation. The work reported in [48] was carried out with the underlying assumption that the common factors in the encoded implementation depend on the encoding chosen. The following theorem shows that this assumption was not entirely valid.

²This is not a true function since the mapping is not uniquely defined for all elements of the domain.

Theorem 5.4.3 *Let \tilde{f} be a factor in $\tilde{\eta} = \mathcal{E}(\eta, E)$. Then there exists a factor f in η such that $\tilde{f} = \mathcal{E}(f, E)$.*

Proof. The proof follows from the fact that \mathcal{E} is invertible. $\eta = \mathcal{E}^{-1}(\tilde{\eta}, E)$. Let, $f = \mathcal{E}^{-1}(\tilde{f}, E)$. Since \tilde{f} is a factor in $\tilde{\eta}$, f must be a factor in η . ■

This result implies that if we consider two encodings, E_1 and E_2 , then corresponding to factor \tilde{f}_1 for E_1 , there is a factor \tilde{f}_2 for E_2 . Thus there is a one-to-one correspondence between factors across different encodings. This implies that a choice of encoding can only determine the particular encoded form of the factor and not its existence. Thus, it may seem that we can always first select an encoding and then find the common factors later. However, the catch here is that the techniques for finding common factors in Boolean networks are algebraic and thus not strong enough to discover many good Boolean factors. Using conventional algebraic factorization after selecting an encoding exposes only a fraction of the common factors present. Thus, even though a common factor may exist in the Boolean network it would remain undetected. (It may also be useless in one network and valuable in the other. This is also related to the fact that for two functions to have a common factor the only condition necessary is that they have a non-zero intersection.) What is desirable is that we should at least be able to obtain factors that can be obtained using algebraic techniques and any possible encoding. The MV-factorization process, using reduced constraint matrices, aims to do exactly this. However, not all factors in all encoded implementations can be obtained by this process. The following example helps illustrate the limitations.

Consider the following expressions and encoding:

$$\begin{aligned} f_3 &= X^{\{1,2,3\}} y_1 \\ f_4 &= X^{\{4,5,6\}} y_2 \\ X^{\{1\}} &: 101 \quad X^{\{2\}} : 111 \quad X^{\{3\}} : 110 \\ X^{\{4\}} &: 001 \quad X^{\{5\}} : 011 \quad X^{\{6\}} : 010 \end{aligned}$$

With s_0, s_1, s_2 as the encoding variables we obtain the following encoded implementation:

$$\begin{aligned} \tilde{f}_3 &= y_1 s_0 s_1 + y_1 s_0 s_2 \\ \tilde{f}_4 &= y_2 \bar{s}_0 s_1 + y_2 \bar{s}_0 s_2 \end{aligned}$$

Here, $s_1 + s_2$ is a kernel common to both expressions. However it cannot be obtained by first doing MV-factorization and then selecting an encoding since no kernels exist for f_3

and f_4 . In this case all the variables involved in this kernel intersection are in the encoding space. As a result the entire kernel intersection in the encoded implementation corresponds to a single MV literal in the original circuit. However, any kernel intersection not comprised entirely of the encoding variables can be detected as the following theorem states.

Theorem 5.4.4 *Let \tilde{k} be a kernel intersection in $\tilde{\eta} = \mathcal{E}(\eta, E)$ not comprised entirely of encoding variables. Then there exists a common factor, k , extracted using the MV-factorization process for $\tilde{\eta}$ (i.e. by using a reduced satisfiable constraint matrix) such that $\tilde{k} = \mathcal{E}(k, E)$.*

Proof. A kernel intersection, \tilde{k} of $\tilde{\eta}$ implies a rectangle, \mathcal{R}_{enc} , of at least two rows, in the binary co-kernel cube matrix for $\tilde{\eta}$. Let λ_i be the co-kernel corresponding to row i of this rectangle and γ_j be the cube corresponding to column j . Let s_i be the cube in λ_i that has the variables used in the encoding of X . Similarly, let s_j be the cube in γ_j that has the variables used in the encoding of X . We construct a rectangle \mathcal{R}^- in the MV co-kernel cube matrix as follows.

For each row and column of \mathcal{R}_{enc} construct a row and column of \mathcal{R}^- . The co-kernel Λ_i , corresponding to row i , and the cube Γ_j , corresponding to column j in \mathcal{R}^- are obtained as follows.

Case 1: $\exists j \mid s_j \neq 1$

$\Gamma_j = \gamma_j / s_j$. $\Lambda_i = \lambda_i / s_i$. The MV part of entry (i, j) in \mathcal{R}^- is assigned the literal $\mathcal{E}^{-1}(s_i s_j, E)$.

Case 2: $\forall j, s_j = 1$

Here all the bits of the encoding space are in the co-kernel. Let $X^{S_i} = \mathcal{E}^{-1}(s_i, E)$. Then, $\Gamma_j = \gamma_j$ and $\Lambda_i = (\lambda_i / s_i) X^{S_i}$. The MV part of each entry of the i th row in \mathcal{R}^- is assigned the literal 1.

At this point there may exist columns in \mathcal{R}^- that correspond to the same cube and rows that correspond to the same co-kernel. Merge each such set of columns (rows) into a single column (row) corresponding to that cube (co-kernel). The MV part of each entry of this column (row) is the disjunction of the MV parts of all the columns (rows) it replaces.

Since there is at least one non-encoding variable in \tilde{k} , there is at least one $\Gamma_j \neq 1$. This ensures that there are at least two non-zero cubes for each row r , of the MV co-kernel cube matrix that is in \mathcal{R}^- . To see this let us assume this were not true, i.e. only one cube existed for some row. But then γ_j / s_j ($\gamma_j / s_j \neq 1$) would divide all the cubes for the

corresponding kernels in the binary co-kernel cube matrix. That cannot happen, otherwise these “kernels” would not be cube free. Each row r is cube free, otherwise the corresponding kernels in the binary co-kernel cube matrix would not be cube free. Thus, r corresponds to a kernel in the original set of expressions.

From this it follows that $M^{\mathcal{R}^-}$ is a reduced constraint matrix of some matrix $M^{\mathcal{R}}$ for some rectangle \mathcal{R} in the MV co-kernel cube matrix. That $M^{\mathcal{R}^-}$ is satisfiable follows from the construction of the MV co-kernel cube matrix. Each MV part entry was constructed as the product of the row and column components. Since $M^{\mathcal{R}^-}$ is satisfiable, there exists a common factor k in η corresponding to a reduction of \mathcal{R} .

We now need to show that $\tilde{k} = \mathcal{E}(k, E)$. $k = \sum_j (\Gamma_j [\cup_i X^{S_{ij}}])$, where $X^{S_{ij}} = M(i, j)$. This can be re-written as $k = \sum_j (\Gamma_j X^{\cup_i S_{ij}})$. Let $X^{S_j} = \mathcal{E}^{-1}(s_j, E)$. Now $X^{S_j - \cup_i S_{ij}}$ is a don't care for X in k , i.e. $k = \sum_j (\Gamma_j X^{\cup_i S_{ij} [S_j - \cup_i S_{ij}]})$. Thus, by expansion $k = \sum_j (\Gamma_j X^{S_j})$, giving $\mathcal{E}(k, E) = \sum_j (\Gamma_j s_j) = \tilde{k}$. ■

Theorem 5.4.4 gives the relationship between potential kernel intersections for the MV variable expressions (reduced rectangles in the MV co-kernel cube matrix) and actual kernel intersections for the encoded implementations (rectangles in the binary co-kernel cube matrix). Note the significance of the result of Theorem 5.4.4. It says that we can view all possible kernel intersections for all possible encodings as long as the kernel intersections have at least one non-encoding variable.

In fact, the MV-factorization process described is even stronger inasmuch as it can potentially discover Boolean factors in the encoded implementation that could not have been found using algebraic techniques. We illustrate this with the following example. Consider the expressions:

$$\begin{aligned} f_4 &= X^{\{1\}} y_1 + X^{\{2\}} y_2 \\ f_5 &= X^{\{1\}} y_1 + X^{\{3\}} y_3 \\ f_6 &= X^{\{4\}} y_2 + X^{\{3\}} y_3 \end{aligned}$$

Each of these has a single kernel corresponding to the co-kernel 1. The co-kernel cube matrix for these expressions is given below. For clarity, only the MV part of the matrix

has been shown since the cube part is obvious.

	y_1	y_2	y_3
1	$X^{\{1\}}$	$X^{\{2\}}$	0
1	$X^{\{1\}}$	0	$X^{\{3\}}$
1	0	$X^{\{4\}}$	$X^{\{3\}}$

Consider the rectangle $\mathcal{R} = \{R(1, 2, 3), C(1, 2, 3)\}$. $M^{\mathcal{R}}$ is satisfiable. This leads to the following decomposition:

$$\begin{aligned}
 f_4 &= X^{\{1,2\}} y_7 \\
 f_5 &= X^{\{1,3\}} y_7 \\
 f_6 &= X^{\{3,4\}} y_7 \\
 f_7 &= X^{\{1\}} y_1 + X^{\{2,4\}} y_2 + X^{\{3\}} y_3
 \end{aligned}$$

Consider the following encoding for X :

$$\begin{aligned}
 X^{\{1\}} &: 01 & X^{\{2\}} &: 11 \\
 X^{\{3\}} &: 00 & X^{\{4\}} &: 10
 \end{aligned}$$

This leads to the following encoded implementation:

$$\begin{aligned}
 \tilde{f}_4 &= s_1 y_7 \\
 \tilde{f}_5 &= \bar{s}_0 y_7 \\
 \tilde{f}_6 &= \bar{s}_1 y_7 \\
 \tilde{f}_7 &= \bar{s}_0 s_1 y_1 + s_0 y_2 + \bar{s}_0 \bar{s}_1 y_3
 \end{aligned}$$

\tilde{f}_7 is a non-algebraic or Boolean factor of \tilde{f}_4 , \tilde{f}_5 and \tilde{f}_6 , since \tilde{f}_7 cannot be obtained from $\tilde{f}_4, \tilde{f}_5, \tilde{f}_6$ by using algebraic division only. The Boolean identities, $s \cdot s = 1$ and $\bar{s} \cdot s = 0$ have been used in the process.

Thus by using a procedure that is an extension of the known algebraic factorization process, we have been able to see possible kernel intersections in all possible encoded implementations as well as additional Boolean factors.

5.5 Circuit Decomposition Using Common Cubes

5.5.1 Common Cube Extraction with Binary Variables

Sub-expressions consisting of single cubes are detected using cube extraction. This is illustrated with the following example. Consider the expressions:

$$f_6 = y_1 y_2 y_3 + y_1 y_2 y_4 + y_5$$

$$f_7 = y_1 y_2 y_5$$

The cube $y_1 y_2$ is common to three cubes in the above expressions. This can be implemented separately resulting in the following equivalent expressions.

$$f_6 = y_8 y_3 + y_8 y_4 + y_5$$

$$f_7 = y_8 y_5$$

$$f_8 = y_1 y_2$$

Cube extraction, or finding common single cube divisors, was introduced in [14]. In [63] it was shown how rectangular covering can be used for this purpose. Rectangular covering is done on the *cube-literal* matrix, which is now described. A row in this matrix corresponds to a cube in the original expressions. A column corresponds to a literal. An entry is 1 if the literal is present in the cube and 0 otherwise. For the set of expressions in the example above we have the following cube-literal matrix.

	y_1	y_2	y_3	y_4	y_5
$y_1 y_2 y_3$	1	1	1	0	0
$y_1 y_2 y_4$	1	1	0	1	0
y_5	0	0	0	0	1
$y_1 y_2 y_5$	1	1	0	0	1

As in Section 5.4.1, a rectangle, \mathcal{R} , in this matrix is a sub-matrix, $\{S_r, S_c\}$, of non-zero entries. S_r and S_c are the sets of rows and columns in this sub-matrix. A rectangle in this matrix corresponds to a cube common to the cubes in S_r . This is the cube formed by the intersection of the literals in S_c . In this example, the rectangle $\{R(1, 2, 4), C(1, 2)\}$ corresponds to the cube $y_1 y_2$ which is a common divisor of the cubes $y_1 y_2 y_3$, $y_1 y_2 y_4$ and $y_1 y_2 y_5$. As shown above this may be extracted and implemented separately.

5.5.2 Common Cube Extraction with Multiple-Valued Variables

Let us now examine cube extraction using the cube-literal matrix when a single MV variable may be present. This is best explained through the following example. Consider the following set of expressions and the corresponding cube-literal matrix.

$$\begin{aligned} f_6 &= X^{\{1\}} y_1 y_2 y_3 + X^{\{2\}} y_1 y_2 y_4 + y_5 \\ f_7 &= X^{\{3\}} y_1 y_2 y_5 \end{aligned}$$

	y_1	y_2	y_3	y_4	y_5
$X^{\{1\}} y_1 y_2 y_3$	1	1	1	0	0
$X^{\{2\}} y_1 y_2 y_4$	1	1	0	1	0
y_5	0	0	0	0	1
$X^{\{3\}} y_1 y_2 y_5$	1	1	0	0	1

Note that in this case the cube-literal matrix does not have any entries corresponding to the MV variable. A rectangle is defined as in the binary case. However, it is interpreted differently. The columns of the rectangle specify the binary variable component of the common cube. The MV literal in the common cube is the union of the MV literals of each of the cubes corresponding to the rows in the rectangle. Thus, for the rectangle, $\{R(1,2,4), C(1,2)\}$, the common cube is $y_1 y_2 X^{\{1,2,3\}}$. Extracting this results in the following decomposition.

$$\begin{aligned} f_6 &= y_8 X^{\{1\}} y_3 + y_8 X^{\{2\}} y_4 + y_5 \\ f_7 &= y_8 X^{\{3\}} y_5 \\ f_8 &= X^{\{1,2,3\}} y_1 y_2 \end{aligned}$$

Let \mathcal{R} be a rectangle in the cube-literal matrix. Let $\alpha_i X^{S_i}$ be the label of row r_i and β_j be the label of column c_j in \mathcal{R} . α_i has only literals of binary-valued variables. Thus row r_i represents cube $\psi_i = \alpha_i X^{S_i} [\prod_j \beta_j]$. ψ_i can be rewritten as $\alpha_i X^{S_i} ([\cup_i X^{S_i}] [\prod_j \beta_j])$. Since $([\cup_i X^{S_i}] [\prod_j \beta_j])$ is independent of i , this is a common cube to all ψ_i .

As in Section 5.4.2 the concept of incompletely specified literals is applicable here. If X takes values from $P = \{0, 1, 2, 3\}$ then f_8 can be equivalently expressed as:

$$f_8 = X^{\{1,2,3\}[0]} y_1 y_2$$

As with kernel intersections, we are interested in seeing if we can use this technique to extract all common cubes for all possible encodings. The following example illustrates the difficulty in doing this when the common cube in the encoded implementation consists of only the encoding variables. Consider the following expressions and encoding:

$$f_3 = X^{\{1\}} y_1$$

$$f_4 = X^{\{2\}} y_2$$

$$X^{\{1\}} : 11 \quad X^{\{2\}} : 01$$

Using the variables s_0 and s_1 for the encoding, we get the following encoded implementation.

$$\tilde{f}_3 = s_0 s_1 y_1$$

$$\tilde{f}_4 = \bar{s}_0 s_1 y_2$$

Here, s_1 is a common cube among the two expressions. However, this cannot be obtained by first detecting a common cube factor in the expressions with the MV variables and then selecting an encoding. This is because at least one common literal is needed in the cube-literal matrix for a rectangle to be extracted, and this literal must be of a non-encoding variable. There is no such common literal in this example.

However, as long as the common cube in the encoded implementation has at least one variable besides the encoding variables then the technique outlined in this section is sufficient to obtain it as shown by the following theorem.

Theorem 5.5.1 *Let \tilde{f} be a single cube common factor in $\tilde{\eta} = \mathcal{E}(\eta, E)$, not comprised entirely of the encoding variables. Then there exists a single cube common factor, f , in η such that $\tilde{f} = \mathcal{E}(f, E)$.*

Proof. Let $\tilde{f} = \beta s$ where β is the cube with the non-encoding variables and s is the cube with the encoding variables. Let \tilde{c}_i be a cube in $\tilde{\eta}$ for which \tilde{f} is a common cube. Then \tilde{c}_i can be written as $\tilde{f} \alpha_i s_i$ where α_i is the cube with the non-encoding variables and s_i is the cube with the encoding variables. Corresponding to this cube there is a cube $c_i = \mathcal{E}^{-1}(\tilde{c}_i, E)$

in η . $c_i = \alpha_i \beta \mathcal{E}^{-1}(ss_i, E)$. Let $\mathcal{E}^{-1}(ss_i)$ be the MV literal X^{S_i} . Therefore, $c_i = \alpha_i \beta X^{S_i}$. Thus using the cube-literal matrix we can extract the cube, $f = \beta [\cup_i X^{S_i}] = \beta X^{\cup_i S_i}$ that is common to each c_i . Now we need to show that $\mathcal{E}(f, E) = \tilde{f}$. Let $\mathcal{E}^{-1}(s, E) = X^S$. Note that f can be equivalently expressed as, $f = \beta X^{\cup_i S_i [S - \cup_i S_i]}$. Thus, f may be expanded to βX^S . Now, $\mathcal{E}(f, E) = \beta s = \tilde{f}$. ■

5.6 Circuit Simplification

The previous sections presented techniques for circuit decomposition in MV-networks. It is now shown how the other common multi-level optimization technique used with Boolean networks, *viz.* node simplification can be used with almost no modification in the context of the MV-network. Node simplification involves using a two-level logic minimizer for each node function. (Recall that each node function is stored in SOP form.) Since we know how to do two-level minimization with MV variables [64], nothing new needs to be developed here.

In [3] it was shown how implicit don't cares in the Boolean network are used in node simplification to capture logical relationships between different nodes in the network. These are now examined in the context of the MV-network. The satisfiability don't care set (SDC) for a Boolean network is defined as the set of signal values in the network that are inconsistent with the network. For node i in the network this is expressed as:

$$SDC(i) = f_i \bar{y}_i + y_i \bar{f}_i$$

This is the set of values for which the function computed by the node is not consistent with the output value of the node. The complete set of satisfiability don't cares for a network is the union of the contributions of all the nodes, i.e. $SDC = \bigcup_i SDC(i)$. Note that each node function in the MV-network is binary-valued, therefore its complement is defined. Thus, the equation of $SDC(i)$ is still valid in the MV-network, only the domain now is $P \times B^{\{m-1\}}$. As an example consider the function represented by the expression:

$$f_3 = X^{\{0\}} y_1 + X^{\{1,2\}} y_2$$

Here X takes values from $P = \{0, 1, 2\}$. \bar{f}_3 can be computed using DeMorgan's Laws.

$$\bar{f}_3 = (\overline{X^{\{0\}} + y_1}) (\overline{X^{\{1,2\}} + y_2}) = (\overline{X^{\{0\}}} + \bar{y}_1) (\overline{X^{\{1,2\}}} + \bar{y}_2) = \overline{X^{\{0\}}} \bar{y}_1 + \overline{X^{\{1,2\}}} \bar{y}_2$$

This is then used to compute $SDC(3)$.

$$SDC(3) = (X^{\{0\}} y_1 + X^{\{1,2\}} y_2) \bar{y}_3 + (X^{\{0\}} \bar{y}_1 + X^{\{1,2\}} \bar{y}_2) y_3$$

For a Boolean network, the observability don't care (ODC) at a node is the set of primary input values for which the output of this node is inconsequential (or cannot be observed) at the primary outputs. For node i in a Boolean network $ODC(i)$ is given by:

$$ODC(i) = \bigcap_j (F_{j y_i} \oplus F_{j \bar{y}_i})$$

Here F_j is the function at primary output j and $F_{j y_i}$ indicates the cofactor operation of this node function with literal y_i . The cofactor operation is defined as follows.

Definition 5.6.1 *The cofactor of a function f with respect to a literal l , denoted by f_l , is the function when l evaluates to 1.*

Again note that in the MV-network each node function is binary-valued, hence cofactor with respect to a signal and its complement are defined and therefore the equation for $ODC(i)$ is still valid. As an example consider the following expressions that describe an MV-network with a single output y_4 .

$$\begin{aligned} f_2 &= X^{\{0,1\}} y_1 \\ f_3 &= X^{\{2\}} y_1 \\ f_4 &= y_2 y_3 \end{aligned}$$

Now, $f_{4 y_2} = X^{\{2\}} y_1$ and $f_{4 \bar{y}_2} = 0$. Thus,

$$ODC(2) = X^{\{2\}} y_1 \oplus 0 = X^{\{0,1\}} + \bar{y}_1$$

From the above we see that node simplification with implicit don't cares is done in an MV-network in exactly the same way as in a Boolean network. The concept of permissible functions introduced in [60] for NOR gates is extended to Boolean networks in [68]. Here it is also shown how a two-level minimizer is employed to use them. Thus, by a similar argument to that used before, this is directly applicable in an MV-network.

5.7 Logic Verification

Combinational logic verification (also referred to as Boolean comparison) is an important part of any logic synthesis system. This is typically used to certify that the final circuit description is functionally equivalent to the initial specification. However, it may also be used during circuit optimization as in [8] to verify that the addition/removal of certain connections is valid.

Verifying the equivalence of two functions f_1 and f_2 is equivalent to checking that the function $f_1 \oplus f_2$ is always true. Thus, equivalence checking reduces to verifying that a function is a tautology. Almost all verification techniques use some form of Shannon's decomposition ([35]). This is a decomposition of a function, f , in terms of the functions f_x and $f_{\bar{x}}$, i.e. the functions in the two half spaces $x = 1$ and $x = 0$. It is given by:

$$f = xf_x + \bar{x}f_{\bar{x}}$$

The following classic result makes this decomposition useful in tautology checking.

Theorem 5.7.1 $f \equiv 1$ if and only if $f_x \equiv 1$ and $f_{\bar{x}} \equiv 1$.

Thus, checking for $f \equiv 1$ is replaced by the checks for $f_x \equiv 1$ and $f_{\bar{x}} \equiv 1$. Theorem 5.7.1 is used recursively on f_x and $f_{\bar{x}}$. Each level in the recursion reduces the number of variables in the function by 1 until finally no variables remain, i.e. the constant functions 0 and 1 remain. This decomposition is for binary-valued x . However, it is easily extended to multiple-valued X [64] as follows:

$$f = X^{\{0\}}f_{X^{\{0\}}} + \dots X^{\{n-1\}}f_{X^{\{n-1\}}}$$

Here $f_{X^{\{i\}}}$ is the value of f when X is equal to i . The following extension to Theorem 5.7.1 provides the recursive tautology check.

Theorem 5.7.2 $f \equiv 1$ if and only if, for all i , $f_{X^{\{i\}}} \equiv 1$.

This extension enables us to use all the Boolean network comparison techniques for MV-networks. Binary decision diagrams (BDD's) [18] have been used very successfully for logic verification [51]. These are a compact representation of a complete Shannon decomposition tree. A BDD is a canonical form of a binary-valued function of binary-valued variables. Hence verifying that two functions are identical reduces to verifying that their BDD's are

identical. In [74] the multiple-valued extensions to BDD's are described. These are naturally referred to as multiple-valued decision diagrams (MDD's). It is demonstrated that MDD's are canonical forms of multiple-valued functions of multiple-valued variables. As with BDD's the canonical form property of MDD's implies that two functions are equivalent if and only if their MDD's are identical. This property enables them to be directly used for verification in MV-networks.

Chapter 6

Practical Experiences

“It’s all about as curious as it can be,” said the Gryphon.

– Lewis Carroll, “Alice in Wonderland”

The optimization techniques for multi-level logic with multiple-valued inputs presented in Chapter 5 have been implemented as the program MIS-MV [42] which is an extension of MIS to handle multiple-valued inputs. This choice of name arises from historical reasons. The multiple-valued successor to the two-level logic minimizer ESPRESSO was named ESPRESSO-MV; the abbreviation MV standing for *multiple-valued*. It is only natural that the multiple-valued extension of MIS be named MIS-MV. MIS (Multi-level Interactive Synthesis), as the name suggests, is an interactive program with the various optimization techniques described in Section 5.1 being accessible through an interactive user interface. MIS-MV provides the same interface (with some additions) as MIS and defaults to it when all the inputs are binary-valued. The extensions include the capability of performing input encoding so that it can be directly used for the input encoding problem, which is the motivation for developing these ideas. This chapter examines the practical issues involved in the implementation, as well as results of using MIS-MV for input encoding on a set of examples.

6.1 Implementation Issues

This section describes the interesting implementation issues and problems that are specific to MIS-MV as well as the approaches used to tackle them.

6.1.1 Size Estimation in Algebraic Decomposition

In algebraic factorization and decomposition of Boolean networks the sequence of operations plays a critical part in the quality of the final results. The sequence referred to here is the sequence in which the algebraic divisors (common kernels and cubes) are extracted. The choice of a particular rectangle affects both the other available rectangles as well as the cost of the rectangles. The exact (or globally optimum) solution to algebraic decomposition is an open problem [63]. The solution accepted in the binary-valued case is a locally optimal one, i.e. a greedy choice is made at each decision step. The evaluation of a particular divisor is done by first estimating its size and then using this to determine the size reduction that would result if this divisor were selected. Determining the size of a divisor is not a problem in the binary-valued case since the size is directly measured in terms of the number of literals in the divisor. Consider the following example. The expression $y_1 y_2 y_3 y_4 + y_1 y_2 y_3 y_5$ is factored as $y_1 y_2 y_3 (y_4 + y_5)$ by extracting the common cube $y_1 y_2 y_3$. The size of the common cube is three literals and extracting this common cube made it possible to represent the same function in five literals instead of eight, a saving of three literals.

Unfortunately in the MV case there is no direct correspondence between the size of the MV-literal and its final encoded implementation. Consider the following example: $X^{\{0\}} y_1 y_2 + X^{\{1\}} y_1 y_3$ is factored as $X^{\{0,1\}} y_1 (X^{\{0\}[2,3]} y_2 + X^{\{1\}[2,3]} y_3)$; X takes values from $\{0, 1, 2, 3\}$. Since the size of the encoded implementation of the MV-literals is a function of the encoding selected, it is not possible to predict the value of this factor in terms of the size reduction obtained in the encoded implementation. In fact, depending on the encoding, the factor may even result in an increase in size. Consider the following encoding E_1 and the encoded implementation of the initial and factored expressions.

$$\begin{aligned} X^{\{0\}} &: 00 & X^{\{1\}} &: 10 \\ X^{\{2\}} &: 01 & X^{\{3\}} &: 11 \end{aligned}$$

Initial expression:

$$\bar{s}_0 \bar{s}_1 y_1 y_2 + s_0 \bar{s}_1 y_1 y_3$$

Factored expression:

$$\bar{s}_1 y_1 (\bar{s}_0 y_2 + s_0 y_3)$$

Thus, cube extraction results in a saving of two literals. However, consider the following

encoding E_2 .

$$X^{(0)} : 00 \quad X^{(1)} : 11$$

$$X^{(2)} : 01 \quad X^{(3)} : 10$$

Initial expression:

$$\bar{s}_0 \bar{s}_1 y_1 y_2 + s_0 s_1 y_1 y_3$$

Factored expression:

$$(\bar{s}_0 \bar{s}_1 + s_0 s_1) y_1 (\bar{s}_0 y_2 + s_0 y_3)$$

In this case, MV-cube extraction actually results in an increase in size! Thus, the potential reduction in size obtained using a factor must be estimated taking into account the effect of encoding. However, no encoding exists yet; the minimization is being done in order to select one. This is a cyclic dependency which must be broken.

The first attempt to model the effect of the encoding assumed that each MV-literal has an encoded implementation that is a single cube in the encoding space of minimum dimension, i.e. the encoding is minimum length¹. This turns out to be a poor estimate of the effect of the final encoding. The reason for this is that the final encoding is the same for all the MV-literals and in most cases the encoded implementation of each MV-literal will not be just a single cube. This is a limitation of this modelling.

This led to the next approach in which an actual encoding is done for the current MV-network. However, the MV-literals are not replaced by their encoded implementations; the codes are just used in the size estimation for factoring. This approach has the advantage that the estimated size is valid for at least one encoded implementation. It has the disadvantage that a relatively slow step of encoding must be done for the estimation. The other potential problem with this is that the final encoding may be very different from the one selected during estimations. However, based on experience with using this, it appears that there are no big changes between the estimates and the results obtained with the final encoding. This is the approach currently being used in MIS-MV. However, there is potential for improvement here.

6.1.2 Incompletely Specified Literals

In Chapter 5 the notion of *incompletely specified literals* was introduced. An incompletely specified literal captures the flexibility available to each literal in terms of how

¹The minimum number of bits needed to encode a p -valued variable is $\lceil \lg(p) \rceil$.

much it can be expanded. In MIS-MV incompletely specified literals are used in two different ways. These are now considered individually.

Node Simplification

As described in Chapter 5 each node function in the MV-network may be simplified using a two-level minimizer (with or without a don't care set). The result of using a two-level minimizer is a prime and irredundant expression ². Removing a redundant cube while making an expression irredundant is obviously desirable since there are fewer cubes in the resulting expression. However, as was observed in Chapter 5 expanding the MV-literal need not result in a size reduction in the encoded implementation. Let $X_j^{S_j}$ be a literal, $X_j^{S_j^+}$ be the fully expanded literal, (i.e. no other element can be added to S_j^+ while preserving logical functionality) and $X_j^{S_j^-}$ be the fully reduced³ literal, (i.e. no other element can be deleted from S_j^- while preserving logical functionality). The range of possibilities available to $X_j^{S_j}$ is captured by the incompletely specified literal $X_j^{S_j^-[S_j^+-S_j^-]}$. This is illustrated through an example. Consider the following prime and irredundant expression: $y_1 X^{\{0\}} + y_2 X^{\{1\}} + y_1 y_2 X^{\{0,1,2\}}$. The third cube may be reduced in X resulting in the equivalent expression: $y_1 X^{\{0\}} + y_2 X^{\{1\}} + y_1 y_2 X^{\{2\}}$. Thus this expression may be represented using incompletely specified literals as: $y_1 X^{\{0\}} + y_2 X^{\{1\}} + y_1 y_2 X^{\{2\}[0,1]}$.

It should be noted that the form of the final reduced expression depends on the order in which the cubes are reduced [9]. This in turn determines the incompletely specified literals. How a cube may be reduced depends on other cubes in the SOP expression.

Local Observability Don't Cares

Section 5.6 described how information about the circuit structure may be captured through implicit don't cares which may then be used in node simplification. While the SDC is relatively easy to compute, computing the ODC is computationally very intensive and not feasible for most real circuits. However, a very limited form of the observability don't care information can be easily captured by just looking at the immediate neighborhood of a particular node in the network. Consider the following example which consists of two node

²A two-level expression is irredundant if no single cube may be deleted from this expression while preserving functional equivalence. It is prime if no literal may be expanded while preserving functional equivalence. Expanding an MV literal $X_j^{S_j}$ involves replacing $X_j^{S_j}$ with $X_j^{S_j^+}$, where $S_j \subset S_j^+$.

³Reducing a literal $X_j^{S_j}$ involves replacing $X_j^{S_j}$ with $X_j^{S_j^-}$, where $S_j' \subset S_j$.

functions:

$$\begin{aligned} f_5 &= y_1 y_6 X^{\{0,1,2\}} + y_4 \\ f_6 &= y_2 X^{\{0,1,3,4\}} + y_3 X^{\{2,5\}} \end{aligned}$$

X takes values from $\{0, 1, \dots, 7\}$. Note that $X^{\{3,4,5,6,7\}}$ is an observability don't care for f_6 . This is so because in the only place that f_6 is used, it is ANDed with $X^{\{0,1,2\}}$ and thus for each product term of f_6 only the values of X that are contained in $\{0, 1, 2\}$ are of any interest; all others will be deleted by the AND. These observability don't cares can be directly utilized by capturing them in the form of incompletely specified literals in f_6 as follows:

$$\begin{aligned} f_5 &= y_1 y_6 X^{\{0,1,2\}} + y_4 \\ f_6 &= y_2 X^{\{0,1\}[3,4,5,6,7]} + y_3 X^{\{2\}[3,4,5,6,7]} \end{aligned}$$

In this example, f_6 is being used in only one cube, i.e. y_6 appears in only one cube. In general, a variable y , corresponding to node function f , will appear in several cubes. Let $l^i X^{S_i[D_i]} (\prod_k l_k^i)$ be the i^{th} such cube. l^i is a literal of y and l_k^i is a literal of some other binary-valued variable y_k . Let $A(f_k) = \cup_j (S_j \cup D_j)$, where j varies over all the cubes in f_k . $A(f_k)$ contains all the values of X that may appear in any cube of f_k . Similarly, let $A(\bar{f}_k)$ be the set that contains all values of X that appear in \bar{f}_k . For literal l_k , let $A(l_k) = A(f_k)$ if $l_k = y_k$ and $A(l_k) = A(\bar{f}_k)$ if $l_k = \bar{y}_k$. Let $O_i = (S_i \cup D_i) \cap (\cap_k A(l_k))$. For $X \notin O_i$, $X^{S_i[D_i]} (\prod_k l_k^i)$ evaluates to 0. Thus, the value of l^i is inconsequential for these values of X , i.e. X^{P-O_i} is a don't care for f at cube i . $X^{\cap_i (P-O_i)}$ is a don't care for f at each cube i . This may be directly used to modify the literals of X in f . Consider the following example in which $X \in \{0, 1, 2, 3, 4\}$.

$$\begin{aligned} f_1 &= X^{\{0,2\}[3]} \\ f_2 &= X^{\{0,1\}[2,3]} \\ f_3 &= y_1 y_2 X^{\{0,1\}[4]} \\ f_4 &= X^{\{2\}[0,3]} \\ f_5 &= y_1 \bar{y}_4 X^{\{4\}[0]} \end{aligned}$$

The local observability don't cares for f_1 need to be computed. At f_3 it is $X^{\{2,3,4\}}$. At f_5 it is $X^{\{1,2,3\}}$. Combining these gives $X^{\{2,3\}}$ as a don't care for f_1 . This enables f_1 to be re-expressed as $X^{\{0\}[2,3]}$.

Since only the fanout nodes and their immediate fanins are considered, computing this local observability don't care information is a fast operation. It should be pointed out that the don't cares exploited in determining the incompletely specified literals are *compatible don't cares* [68]. All these don't cares are valid simultaneously and may be used independently of each other.

6.1.3 Satisfiable Constraint Matrices

Theorem 5.4.2 specifies that a factorization suggested by a rectangle in the co-kernel cube matrix is valid if and only if its constraint matrix, M , is satisfiable. It is also demonstrated in Section 5.4.2 how a common factor may be extracted using a reduced constraint matrix, M_r , that is satisfiable. Thus, if M is not satisfiable there are two possibilities for deriving a satisfiable matrix from it.

1. Obtain a sub-matrix that is satisfiable. This is accomplished by deleting some rows and/or columns in M .
2. Derive a satisfiable reduced constraint matrix M_r from M . This is done by expanding some entries of M (deleting some values from the superscripts of the entries).

These are now considered individually. The following example is used to illustrate both possibilities. Consider the network with the following two functions:

$$\begin{aligned} f_6 &= y_1 y_2 X^{\{1,2\}} + y_1 y_3 X^{\{2\}} + y_1 y_4 X^{\{3\}} \\ f_7 &= y_5 y_2 X^{\{2,3\}} + y_5 y_3 X^{\{2\}} + y_5 y_4 X^{\{3,4\}} \end{aligned}$$

The following constraint matrix is obtained from the rectangular covering formulation. The details of the formulation have been omitted here for brevity.

$$M = \begin{bmatrix} X^{\{1,2\}} & X^{\{2\}} & X^{\{3\}} \\ X^{\{2,3\}} & X^{\{2\}} & X^{\{3,4\}} \end{bmatrix}$$

Note that M is not satisfiable since:

$$\begin{aligned} ([\cup_i M_{i1}] \cap [\cup_j M_{1j}]) &= X^{\{1,2,3\}} \cap X^{\{1,2,3\}} \\ &= X^{\{1,2,3\}} \\ &\neq M_{11} \end{aligned}$$

Satisfiable Sub-matrices

There are several possible sub-matrices of M that are satisfiable. Column 1 may be deleted resulting in M_1 shown below.

$$M_1 = \begin{bmatrix} X^{(2)} & X^{(3)} \\ X^{(2)} & X^{(3,4)} \end{bmatrix}$$

Alternatively, column 3 may be deleted resulting in M_2 shown below.

$$M_2 = \begin{bmatrix} X^{(1,2)} & X^{(2)} \\ X^{(2,3)} & X^{(2)} \end{bmatrix}$$

Both M_1 and M_2 are satisfiable.

Similarly, deletion of either row is sufficient to generate a satisfiable sub-matrix. However, a matrix with a single row is probably not very useful. The problem of choosing the optimal set of rows and columns that need to be deleted can be formulated as a minimum cost covering problem which is defined as follows.

Minimum Cost Covering Problem: Let A be an $m \times n$ binary matrix, i.e. each $A_{ij} \in \{0, 1\}$. Associated with each column j in A is a cost for that column denoted by c_j . Find a binary row vector \mathbf{x} such that $A \cdot \mathbf{x}^T \geq (1, 1, \dots, 1)^T$ and $\sum_{j=1}^n x_j \cdot c_j$ is minimum.

The constraint $A \cdot \mathbf{x}^T \geq (1, 1, \dots, 1)^T$ specifies that for each row i there is at least one column j such that $A_{ij} = 1$ and $x_j = 1$. Column j is said to cover row i . The minimum cost covering problem is to find a set of columns that cover all rows with least cost. The minimum covering problem can also be viewed as a minimum cost monotone ⁴ CNF (conjunctive normal form) satisfiability problem. x_j is a binary-valued variable indicating whether column j is included or not in the cover ($x_j = 1$ indicates that it is included). Each row of A represents a clause corresponding to the disjunction of variables for the non-zero elements in the row. Since each clause is to be satisfied (i.e. at least one x_j should be 1 for each row), the covering problem is equivalent to finding an assignment of the x_j 's that satisfies the conjunction of these clauses. The decision problem for minimum cost covering is NP-complete [31]; however efficient heuristics exist which result in reasonably fast solutions for most practical instances [63].

⁴ A Boolean expression is said to be monotone if each variable appears in either uncomplemented or complemented form but not both.

The formulation of optimal row/column deletion as a minimum cost covering problem is done as follows. There is a column in A for each row r_i and each column c_j of M . The columns of A will subsequently be referred to by the corresponding rows and columns in M , i.e. as r_i or c_j . Let M_{ij} be an entry of M where the satisfiability condition is violated. Let V_{ij} be the set of values that cause the violation, i.e. $V_{ij} = ([\cup_i M_{ij}] \cap [\cup_j M_{ij}]) - M_{ij}$. To get rid of this violation one of the following must be done:

1. Delete row i .
2. Delete column j .
3. Delete all rows k such that $(M_{kj} \cap V_{ij}) \neq \phi$. Let R be the set of these rows.
4. Delete all columns l such that $(M_{il} \cap V_{ij}) \neq \phi$. Let C be the set of these columns.

This condition needs to be captured in A . Let x_{r_i} be the binary variable indicating if r_i is deleted. Similarly, x_{c_j} specifies if c_j is deleted. Satisfying the above condition implies finding a satisfying assignment for:

$$x_{r_i} + x_{c_j} + \prod_{k \in R} x_{r_k} + \prod_{l \in C} x_{c_l}$$

Note that this is not a CNF expression. Since the column covering formulation has a direct correspondence with CNF expressions, this expression needs to be converted to a CNF. The equivalent CNF is:

$$\prod_{k \in R} \prod_{l \in C} (x_{r_i} + x_{c_j} + x_{r_k} + x_{c_l})$$

Each clause in this expression specifies a row in A with a 1 in columns r_i , c_j , r_k and c_l .

Let column j correspond to a row (column) in M . The cost, c_j , of column j of A should reflect the increase in size of the circuit if column j is included in the cover, i.e. the corresponding row (column) deleted from M . An approximation to this is the estimated size of the cubes in the original expressions corresponding to this row (column) in M . For the given example and column c_3 , the cost is the estimated size of the cubes $y_1 y_4 X^{\{3\}}$ and $y_5 y_4 X^{\{3,4\}}$. The size estimation is done as described in Section 6.1.1.

Reduced Constraint Matrices

As with satisfiable sub-matrices there are several possible ways to derive a reduced constraint matrix from a given constraint matrix. The value 3 may be deleted from M_{21}

giving M_3 shown below.

$$M_3 = \begin{bmatrix} X^{\{1,2\}} & X^{\{2\}} & X^{\{3\}} \\ X^{\{2\}} & X^{\{2\}} & X^{\{3,4\}} \end{bmatrix}$$

Alternatively the value 3 may be deleted from M_{13} giving M_4 shown below.

$$M_4 = \begin{bmatrix} X^{\{1,2\}} & X^{\{2\}} & X^{\{\}} \\ X^{\{2,3\}} & X^{\{2\}} & X^{\{3,4\}} \end{bmatrix}$$

Both these matrices are satisfiable. The problem of finding the optimal set of values to be deleted (under a cost function described later in this section) can again be formulated as a minimum cost covering problem. In this case each column of A is associated with a 3-tuple (k, l, D) , where the set of values D need to be deleted from M_{kl} . As before, consider a violation of the satisfiability condition at M_{ij} and let V_{ij} be the set of values whose deletion will remove the conflict, i.e. $V_{ij} = ([\cup_i M_{ij}] \cap [\cup_j M_{ij}]) - M_{ij}$. The values in V_{ij} may be deleted from either row i or column j . Thus, one of the following needs to be done:

1. For each k such that $M_{kj} \cap V_{ij} \neq \phi$, let $D_{kj} = M_{kj} \cap V_{ij}$. Delete D_{kj} from all such M_{kj} 's i.e., select all columns (k, j, D_{kj}) in the column cover. Let C_1 be the set of these columns.
2. For each l such that $M_{il} \cap V_{ij} \neq \phi$, let $D_{il} = M_{il} \cap V_{ij}$. Delete D_{il} from all such M_{il} 's, i.e., select all columns (i, l, D_{il}) in the column cover. Let C_2 be the set of these columns.

Let x_c be the binary variable specifying if the column c is included in the cover or not. The condition that needs to be satisfied is:

$$\prod_{c_{1i} \in C_1} x_{c_{1i}} + \prod_{c_{2j} \in C_2} x_{c_{2j}}$$

This is equivalent to the CNF expression:

$$\prod_{c_{1i} \in C_1} \prod_{c_{2j} \in C_2} (x_{c_{1i}} + x_{c_{2j}})$$

This is captured by adding a row in A with ones in columns c_{1i} and c_{2j} .

The cost of a column should reflect the size of the extra logic needed to separately implement that part of the cube that corresponds to the deleted entries. For the example given and column $(1, 3, X^{\{3\}})$, it is the estimated size of the cube $y_1 y_4 X^{\{3\}}$. As before the size estimation is done as described in Section 6.1.1.

6.1.4 The Encoding Problem

Section 5.4.2 formally introduced encodings and encoded implementations. This section states the encoding problem that needs to be solved and the approaches used in MIS-MV to tackle this. Given an MV-network, an encoding needs to be selected such that the size of the encoded network is minimized. The size is measured in terms of the factored form literals of the encoded implementations of each node function. This is a difficult task and there seems to be no easy solution to it. An approximate solution to this can be found by minimizing the number of literals in SOP form in the encoded implementation of each literal of X that appears in any node function. This is a relatively easier task to handle. This problem is referred to as the *minimum literal encoding problem* and is stated precisely below.

Problem P1: Minimum Literal Encoding Problem : Let S be a set of incompletely specified literals of X . Find an encoding E , such that

$$\sum_{X^{S_i[D_i]} \in S} \#lits(\mathcal{E}(X^{S_i[D_i]}, E))$$

is minimized. $\#lits(\mathcal{E}(X^{S_i[D_i]}, E))$ is the number of literals in $\mathcal{E}(X^{S_i[D_i]}, E)$.

Input encoding for two-level implementations is currently handled by solving the following problem which is referred to as the *minimum cube encoding problem* [59, 79].

Problem P2: Minimum Cube Encoding Problem : Let S be a set of literals of X . Find an encoding E , such that

$$\sum_{X^{S_i} \in S} \#cubes(\mathcal{E}(X^{S_i}, E))$$

is minimized. $\#cubes(\mathcal{E}(X^{S_i}, E))$ is the number of cubes in $\mathcal{E}(X^{S_i}, E)$.

P1 differs from P2 in two ways. First the cost function being minimized is different. The second and more critical difference is that P2 does not consider incompletely specified literals. These differences make it difficult for existing techniques for solving P2 to be used directly or to be easily modified for P1. As a result an approach had to be developed to tackle P1.

Since this was not the main focus of this research, a solution was sought that could be implemented quickly. Algorithm `sa_encoding` described in Figure 6.1 is a very simplistic

algorithm based on the concept of simulated annealing [41], which has been successfully used to tackle several combinatorial optimization problems.

In the inner loop of the annealing, a new encoding is selected by doing a pairwise swap of two codes in the encoding. If this results in a smaller size, then the new encoding is accepted. If not, it is accepted with probability $e^{-\frac{\delta}{T}}$. δ is the increase in size after the swap. The function `toss_a_coin` outputs a 1 with probability p and a 0 with probability $1-p$. `size(S,E)` is $\sum_{X^{S_i|D_i} \in S} \#lits(\mathcal{E}(X^{S_i|D_i}, E))$, and is computed as follows. The sum of product expression $\mathcal{E}(X^{S_i}, E)$ is minimized with a two-level minimizer with $\mathcal{E}(X^{D_i}, E)$ as the don't cares. The cost function for this minimization is the number of literals in the SOP form ⁵. Since this step is to be performed in the inner loop of the algorithm only a single *expand* step is used instead of a complete two-level minimization.

The temperatures are selected by a piece-wise linear cooling schedule. Two different rates of cooling are used: an initial rapid cooling and a final slow cooling. There is equilibrium at a particular temperature if three consecutive passes do not result in any change in size. As will be seen in Section 6.2, the quality of results obtained from algorithm `sa_encoding` is reasonably good. However, it is potentially very slow.

The definition of the minimum literal encoding problem has prompted other researchers to look at solutions for it. Very recently a solution has been proposed [66], that is based on the notion of *dichotomies* [78, 80]

6.2 Experimental Results

Two sets of experiments were conducted using MIS-MV for input encoding. These were directed towards answering the following two questions.

1. How well does MIS-MV compare with other existing input encoding programs such as NOVA [79], MUSTANG [25], JEDI [45] and MUSE [34]?
2. What is the relative importance of the different multi-level optimization techniques?

The example descriptions used for these experiments are the 1989 International Workshop on Logic Synthesis (IWLS) benchmarks of FSM descriptions [47]. The state

⁵The primary cost function generally used in two-level minimization is the number of cubes in the SOP form. However, when a two-level minimizer is used in a multi-level network for node simplification, the cost function used is the number of literals in the SOP form.

```
/* inputs: S
   outputs: E (the encoding)
*/

sa_encoding(S)
{
    E = random_encoding(S);
    foreach T{ /* T is the temperature */
        repeat{
            new_E = pairwise_swap(E);
             $\delta$  = size(S, new_E) - size(S, E);
            if( $\delta < 0$ )
                p = 1;
            else
                p =  $e^{-\delta/T}$ ;
            accept = toss_a_coin(p);
            if(accept){
                E = new_E;
            }
        }until equilibrium;
    }
    return E;
}
```

Figure 6.1: Algorithm sa_encoding

variable that is an input to this description is multiple-valued. The state variable that is an output of this description is kept *one hot*; i.e. there is a separate signal for each value. This is to enable the experiments to focus on input encoding and exclude output encoding effects [56]. The experiments were conducted as follows:

- The minimum-length encoding was always used.
- A single simplified Boolean script ⁶ was used both for multi-valued and binary-valued optimization. The script has two main parts, node simplification followed by algebraic restructuring. Encoding may be done at any point in the script.
- The script was run twice in all cases.
- For MIS-MV:
 1. ESPRESSO was run on the unencoded machine.
 2. All or part of (depending on the stage where the encoding is done) the first script was run on the MV-network.
 3. The inputs were encoded, using the simulated annealing algorithm.
 4. Any remaining part of the first script and the complete second script was run on the Boolean network.
- For NOVA, MUSTANG, JEDI and MUSE:
 1. These are general encoding programs tackling input, output and input-output encoding. They were run in *input oriented* mode with the appropriate command line options. (“-e ih” for NOVA, “-p -c” for MUSTANG, “-e i” for JEDI and “-e p” for MUSE.)
 2. ESPRESSO was run on the unencoded machine.
 3. The symbolic input was encoded.
 4. ESPRESSO was run again, using the unused codes as don’t cares.
 5. The script was executed twice to be compatible with the two scripts used for MIS-MV.

⁶This is the standard Boolean script distributed with MIS release 2.1.

In order to examine the relative contributions of the different multi-level optimization techniques, different runs of MIS-MV were considered with the encoding done at different steps.

1. At the beginning. These results reflect the quality of just the encoding program since at this point it has exactly the same two-level information as the other programs.
2. After *simplify*. These results reflect the advantage gained by including node simplification with the SDC.
3. After algebraic optimization. These results reflect the additional advantage gained by considering the algebraic restructuring techniques of MIS-MV.

Table 6.1 contains the results, expressed as the number of factored form literals in the encoded implementation. The following observations are made from these results.

1. MIS-MV performs very well in comparison with the other programs. Its result is consistently either the best or close to (within 5%) the best among all programs. In addition there are some cases where it does significantly better than the other programs (e.g. *keyb*, *tbk*).
2. The column labelled *beginning* reflects the performance of just the encoding algorithm. These results indicate that it does a reasonably good job of encoding in comparison with the other programs.
3. For a large number of cases the node simplification step in MIS-MV seems to contribute the most.

However there are two cases (*keyb* and *tbk*) for which the algebraic optimizations are the main contributors to the reduction in area. These are also the cases where MIS-MV clearly out-performs the other programs. From this it seems that on the average MIS-MV will do about as good or slightly better than other programs; when it can extract useful algebraic factors in the MV-network it will clearly out-perform other programs that do not exploit this information.

4. There are some examples (e.g. *ex2*) where encoding after algebraic optimization results in larger circuits in comparison with encoding at an earlier stage. This directly points to inaccuracies in size estimation during algebraic decomposition.

example	NOVA	MUSTANG	JEDI	MUSE	best MIS-MV	beginning	simplify	algebraic opt.
bbara	106	96	96	99	84	84	84	85
bbsse	151	148	125	126	130	130	132	131
bbtas	32	37	34	36	31	35	31	31
beecount	70	65	57	60	56	62	56	58
cse	214	208	189	192	191	191	199	195
dk14	98	108	97	102	79	97	79	81
dk15	65	65	65	65	65	65	68	69
dk16	351	314	254	244	225	225	247	261
dk17	58	69	63	58	58	58	62	63
dk27	38	34	30	29	27	27	27	27
dk512	93	78	73	73	68	70	68	69
donfile	186	195	132	131	123	127	123	123
ex1	246	252	256	239	232	240	232	236
ex2	167	197	179	169	143	143	144	154
ex3	98	98	87	96	82	82	86	82
ex4	84	73	71	72	72	90	74	72
ex5	83	80	79	79	67	67	69	69
ex6	98	90	91	92	84	85	85	84
ex7	94	100	93	84	78	89	79	78
keyb	195	203	186	180	146	186	172	146
kirkman	168	181	175	195	160	169	166	160
lion	16	14	16	16	16	16	16	16
lion9	43	61	55	55	38	40	38	38
mark1	98	99	94	92	90	90	94	92
mc	32	30	32	30	30	35	30	30
modulo12	71	77	58	72	71	71	71	71
opus	82	77	83	70	70	87	70	74
planet	551	538	454	511	466	512	466	473
s1	345	377	347	291	249	335	253	249
s1a	253	264	262	195	214	217	214	225
s8	48	47	50	52	48	52	48	48
sand	542	519	552	498	509	523	509	528
shiftreg	35	34	24	25	24	24	24	24
sse	151	148	125	126	130	130	132	131
styr	501	460	413	418	438	442	438	465
tav	27	27	27	27	27	27	27	27
tbk	567	603	463	570	393	426	456	393
train11	92	88	65	79	59	60	59	59
train4	14	18	14	14	14	14	15	15
total	6163	6172	5566	5562	5087	5423	5243	5232

Table 6.1: Input Encoding Comparison

6.2.1 An Example MIS-MV Run

In this section an example circuit is considered at various stages of optimization using MIS-MV. The circuit selected for this purpose is *keyb* since it helps illustrate several interesting aspects of the optimization in a single example. In the rest of this section a description of the circuit as output by MIS-MV is given along various points in the optimization script and interesting aspects highlighted. This description is in terms of the expressions describing the node equations.

Initial Circuit

Figure 6.2 shows the description of the initial circuit. The circuit has seven binary-valued inputs named $v0, v1, \dots, v6$ and a single MV variable $v7$ which has nineteen possible values numbered $0, 1, \dots, 18$. The complement of a binary-valued variable, v , is written as v' . A literal of $v7$ is written as a set of values. For example, $v7^{0,1}$ is written as $\{v7.0, v7.1\}$. An incompletely specified literal such as $v7^{0,1}[2,3]$ is written as $\{v7.0, v7.1[v7.2, v7.3]\}$. There are twenty one binary-valued outputs named $v8.0, v8.1, \dots, v8.20$. The node functions and output variables are referred to by the same name. If the name is on the left hand side of an equation, it represents the function; if it is on the right hand side of the equation, it represents the output variable of that function.

After Node Simplification

Figure 6.3 has the circuit description after node simplification. Note that $v8.0, v8.4, v8.6, v8.19$ have been re-expressed in terms of the other node functions. This is a result of using the SDC's from the rest of the circuit in the simplification. These functions also demonstrate the use of incompletely specified literals after node simplification.

After Algebraic Decomposition

Figure 6.4 contains the description of the circuit after algebraic decomposition using cube and kernel extraction. $[23]$ is a common cube that has been extracted and used in seven places. $[21], [25]$ and $[26]$ are common kernels. Also, local observability don't care information is used to derive the incompletely specified literal in $\{v8.10\}$. All values that do not appear in $[23]$ are used as don't cares for this literal.

$$\begin{aligned}
\{v8.0\} &= v0 \ v3 \ {v7.1\Box} + v0 \ v2 \ {v7.1\Box} + v0 \ v1 \ {v7.1\Box} + v0 \ v4 \ {v7.1\Box} \\
&+ v0 \ v5 \ {v7.1\Box} + v0 \ v6 \ {v7.1\Box} + v1 \ v4 \ {v7.1,v7.4\Box} + v1 \ v5 \ {v7.1, \\
&v7.4\Box} + v1 \ v2 \ {v7.1,v7.4\Box} + v1 \ v6 \ {v7.1,v7.4\Box} + v1 \ v3 \ {v7.1, \\
&v7.4\Box} + v2 \ v4 \ {v7.1,v7.4,v7.7\Box} + v3 \ v5 \ {v7.0, \ v7.1, \ v7.4,v7.7\Box} + \\
&v3 \ v6 \ {v7.0,v7.1,v7.4,v7.7\Box} + v2' \ v3 \ {v7.2, \ v7.3, \ v7.5,v7.6,v7.8, \\
&v7.9\Box} + v0 \ {v7.2,v7.3\Box} + v2 \ v3' \ {v7.2, \ v7.3,v7.5, \ v7.6,v7.8,v7.9\Box} \\
&+ v4 \ {v7.2,v7.5,v7.8,v7.11\Box} + v2 \ v3 \ {v7.1,v7.2, \ v7.4,v7.5,v7.7,v7.8\Box} \\
&+ v5 \ {v7.2,v7.5,v7.8,v7.11,v7.12, \ v7.14\Box} + v6 \ {v7.2,v7.5,v7.8,v7.11, \\
&v7.12,v7.14,v7.16\Box} + v1 \ {v7.2,v7.3,v7.5,v7.6\Box} + v2 \ v5 \ {v7.1,v7.3, \\
&v7.4,v7.6,v7.7,v7.9\Box} + v3 \ v4 \ {v7.1,v7.3,v7.4,v7.6, \ v7.7,v7.9\Box} + v2 \\
&v6 \ {v7.1,v7.3,v7.4, \ v7.6,v7.7,v7.9\Box} + v4 \ v6 \ {v7.0,v7.1,v7.3,v7.4, \ v7.6,v7.7,v7.9, \\
&v7.10\Box} + \{v7.17\Box\} + \{v7.18\Box\} + v5 \ v6 \ {v7.0,v7.1, \ v7.3,v7.4,v7.6, \\
&v7.7, \ v7.9,v7.10,v7.13\Box} \\
\{v8.1\} &= v3' \ v4' \ v5' \ v6' \ {v7.0\Box} \\
\{v8.2\} &= v3' \ v4' \ v5' \ v6' \ {v7.0\Box} + v3' \ v4' \ v5 \ v6' \ {v7.0\Box} + v3' \ v4 \ v5' \ v6' \\
&\{v7.0\Box\} \\
\{v8.3\} &= v3 \ v5' \ v6' \ {v7.0\Box} \\
\{v8.4\} &= v0' \ v1' \ v2' \ v3' \ v4' \ v5' \ v6' \ {v7.1\Box} \\
\{v8.5\} &= v0' \ v1' \ v2 \ v3 \ v4' \ v5' \ v6' \ {v7.3\Box} + v0' \ v1' \ v2' \ v3' \ v4' \ v5' \ v6' \\
&\{v7.2\Box\} + v0' \ v1' \ v2' \ v3 \ v4' \ v5' \ v6' \ {v7.1\Box} + v0' \ v1' \ v2 \ v3' \ v4' \ v5' \\
&v6' \ {v7.1\Box} + v0' \ v1' \ v2' \ v3' \ v4' \ v5' \ v6 \ {v7.1,v7.3\Box} + v0' \ v1 \ v2' \ v3' \\
&v4' \ v5' \ v6' \ {v7.1\Box} + v0' \ v1' \ v2' \ v3' \ v4' \ v5 \ v6' \ {v7.1,v7.3\Box} + v0' \ v1' \\
&v2' \ v3' \ v4 \ v5' \ v6' \ {v7.1,v7.3\Box} + v0 \ v1' \ v2' \ v3' \ v4' \ v5' \ v6' \ {v7.1\Box} \\
\{v8.6\} &= v0' \ v1' \ v2' \ v3' \ v4' \ v5' \ v6' \ {v7.3\Box} \\
\{v8.7\} &= v1' \ v2' \ v3' \ v4' \ v5' \ v6' \ {v7.4\Box} \\
\{v8.8\} &= v1' \ v2 \ v3 \ v4' \ v5' \ v6' \ {v7.6\Box} + v1' \ v2' \ v3 \ v4' \ v5' \ v6' \ {v7.4\Box} \\
&+ v1' \ v2' \ v3' \ v4' \ v5' \ v6' \ {v7.5\Box} + v1' \ v2 \ v3' \ v4' \ v5' \ v6' \ {v7.4\Box} + \\
&v1' \ v2' \ v3' \ v4' \ v5' \ v6 \ {v7.4,v7.6\Box} + v1' \ v2' \ v3' \ v4' \ v5 \ v6' \ {v7.4, \\
&v7.6\Box} + v1 \ v2' \ v3' \ v4' \ v5' \ v6' \ {v7.4\Box} + v1' \ v2' \ v3' \ v4 \ v5' \ v6' \\
&\{v7.4,v7.6\Box\} \\
\{v8.9\} &= v1' \ v2' \ v3' \ v4' \ v5' \ v6' \ {v7.6\Box} \\
\{v8.10\} &= v2' \ v3' \ v4' \ v5' \ v6' \ {v7.7\Box} \\
\{v8.11\} &= v2 \ v3 \ v4' \ v5' \ v6' \ {v7.9\Box} + v2' \ v3 \ v4' \ v5' \ v6' \ {v7.7\Box} + v2 \\
&v3' \ v4' \ v5' \ v6' \ {v7.7\Box} + v2' \ v3' \ v4' \ v5' \ v6 \ {v7.7,v7.9\Box} + v2' \ v3' \\
&v4' \ v5 \ v6' \ {v7.7,v7.9\Box} + v2' \ v3' \ v4' \ v5' \ v6' \ {v7.8\Box} + v2' \ v3' \ v4 \\
&v5' \ v6' \ {v7.7,v7.9\Box} \\
\{v8.12\} &= v2' \ v3' \ v4' \ v5' \ v6' \ {v7.9\Box} \\
\{v8.13\} &= v4' \ v5' \ v6' \ {v7.10\Box} \\
\{v8.14\} &= v4' \ v5' \ v6 \ {v7.10\Box} + v4' \ v5 \ v6' \ {v7.10\Box} + v4 \ v5' \ v6' \ {v7.10, \\
&v7.12\Box} + v4' \ v5' \ v6' \ {v7.11,v7.12\Box} \\
\{v8.15\} &= v5' \ v6' \ {v7.13\Box} \\
\{v8.16\} &= v5' \ v6 \ {v7.13\Box} + v5 \ v6' \ {v7.13\Box} + v5' \ v6' \ {v7.14\Box} \\
\{v8.17\} &= v6' \ {v7.15\Box} \\
\{v8.18\} &= v6 \ {v7.15\Box} + v6' \ {v7.16\Box} \\
\{v8.19\} &= v0' \ v1' \ v2' \ v3' \ v4' \ v5' \ v6' \ {v7.3\Box} + v0' \ v1' \ v2' \ v3' \ v4' \ v5' \\
&v6' \ {v7.1\Box} + v1' \ v2' \ v3' \ v4' \ v5' \ v6' \ {v7.6\Box} + v1' \ v2' \ v3' \ v4' \ v5' \\
&v6' \ {v7.4\Box} + v2' \ v3' \ v4' \ v5' \ v6' \ {v7.7\Box} + v3' \ v4' \ v5' \ v6 \ {v7.0\Box} \\
&+ v3' \ v4' \ v5 \ v6' \ {v7.0\Box} + v3' \ v4 \ v5' \ v6' \ {v7.0\Box} + v3' \ v4' \ v5' \ v6' \\
&\{v7.0\Box\} + v3 \ v5' \ v6' \ {v7.0\Box} + v4' \ v5' \ v6' \ {v7.10\Box} + v5' \ v6' \ {v7.13\Box} \\
\{v8.20\} &= \{v7.18\Box\}
\end{aligned}$$

Figure 6.2: *keyb*: Initial Circuit


```

{v8.0} = {v8.5}' {v8.8}' {v8.11}' {v8.12}' {v8.14}' {v8.16}' {v8.18}'
        {v8.19}' {v7.0,v7.1,v7.2,v7.3,v7.4,v7.5,v7.6,v7.7,v7.8,v7.9,v7.10,
        v7.11,v7.12,v7.13,v7.14,v7.16,v7.17,v7.18□}
{v8.1} = v3' v4' v5' v6' {v7.0□}
{v8.2} = v3' v4' v5' v6' {v7.0□} + v3' v4' v5' v6' {v7.0□} + v3' v4' v5' v6'
        {v7.0□}
{v8.3} = v3 v5' v6' {v7.0□}
{v8.4} = {v8.19} {v7.1[v7.2,v7.5,v7.8,v7.9,v7.11,v7.12,v7.14,v7.15, v7.16,
        v7.17,v7.18]}
{v8.5} = v0' v1' v2' v3' v4' v5' v6' {v7.1,v7.3□} + v0' v1' v2' v3' v4' v5
        v6' {v7.1,v7.3□} + v0' v1' v2' v3' v4' v5' v6' {v7.1,v7.3□} + v0' v1'
        v2' v3' v4' v5' v6' {v7.3□} + v0' v1' v2' v3' v4' v5' v6' {v7.1□} + v0'
        v1' v2' v3' v4' v5' v6' {v7.1□} + v0' v1' v2' v3' v4' v5' v6' {v7.1□} +
        v0' v1' v2' v3' v4' v5' v6' {v7.1□} + v0' v1' v2' v3' v4' v5' v6' {v7.2□}
{v8.6} = {v8.19} {v7.3[v7.2,v7.5,v7.8,v7.9,v7.11,v7.12,v7.14,v7.15,
        v7.16,v7.17,v7.18]}
{v8.7} = v1' v2' v3' v4' v5' v6' {v7.4□}
{v8.8} = v1' v2' v3' v4' v5' v6' {v7.4,v7.6□} + v1' v2' v3' v4' v5' v6'
        {v7.4,v7.6□} + v1' v2' v3' v4' v5' v6' {v7.4,v7.6□} + v1' v2' v3' v4' v5'
        v6' {v7.6□} + v1' v2' v3' v4' v5' v6' {v7.4□} + v1' v2' v3' v4' v5' v6'
        {v7.4□} + v1' v2' v3' v4' v5' v6' {v7.4□} + v1' v2' v3' v4' v5' v6'
        {v7.5□}
{v8.9} = v1' v2' v3' v4' v5' v6' {v7.6□}
{v8.10} = v2' v3' v4' v5' v6' {v7.7□}
{v8.11} = v2' v3' v4' v5' v6' {v7.7,v7.9□} + v2' v3' v4' v5' v6' {v7.7,
        v7.9□} + v2' v3' v4' v5' v6' {v7.7,v7.9□} + v2' v3' v4' v5' v6' {v7.9□}
        + v2' v3' v4' v5' v6' {v7.7□} + v2' v3' v4' v5' v6' {v7.7□} + v2' v3'
        v4' v5' v6' {v7.8□}
{v8.12} = v2' v3' v4' v5' v6' {v7.9□}
{v8.13} = v4' v5' v6' {v7.10□}
{v8.14} = v4' v5' v6' {v7.10□} + v4' v5' v6' {v7.10□} + v4' v5' v6' {v7.10,
        v7.12□} + v4' v5' v6' {v7.11,v7.12□}
{v8.15} = v5' v6' {v7.13□}
{v8.16} = v5' v6' {v7.13□} + v5' v6' {v7.13□} + v5' v6' {v7.14□}
{v8.17} = v6' {v7.15□}
{v8.18} = v6' {v7.15□} + v6' {v7.16□}
{v8.19} = {v8.2} + v5' v6' {v8.14}' {v7.0,v7.10,v7.13[v7.12]} + v1' v2' v3'
        v4' v5' v6' {v8.14}' {v7.4,v7.6[v7.0,v7.7,v7.10,v7.11,v7.12, v7.13]} +
        v0' v1' v2' v3' v4' v5' v6' {v8.14}' {v7.1,v7.3[v7.0, v7.4,v7.6,v7.7,
        v7.10,v7.11,v7.12,v7.13]} + v3' v4' v5' v6' {v8.11}' {v8.14}' {v7.7
        [v7.0,v7.10,v7.11,v7.12,v7.13]}
{v8.20} = {v7.18□}

```

Figure 6.3: *keyb*: Circuit after Node Simplification

```

{v8.0} = {v8.5}' {v8.8}' {v8.11}' {v8.12}' {v8.14}' {v8.16}' {v8.18}' {v8.19}',
        {v7.0,v7.1,v7.2,v7.3,v7.4,v7.5,v7.6,v7.7,v7.8,v7.9,v7.10,v7.11,v7.12,
        v7.13, v7.14,v7.16,v7.17,v7.18□}
{v8.1} = v3' [30] {v7.0□}
{v8.2} = v3' [25] {v7.0[v7.2,v7.5,v7.8,v7.11,v7.13,v7.14,v7.15,v7.16, v7.17,
        v7.18]}
{v8.3} = v3 [29] {v7.0□}
{v8.4} = {v8.19} {v7.1[v7.2,v7.5,v7.8,v7.9,v7.11,v7.12,v7.14,v7.15,v7.16,
        v7.17,v7.18]}
{v8.5} = v0 v1' [23] {v7.1[v7.0,v7.10,v7.11,v7.12,v7.13,v7.14,v7.15, v7.16,
        v7.17,v7.18]} + v0' [26] {v7.1,v7.2,v7.3[v7.0,v7.7,v7.8,v7.9, v7.10,v7.11,
        v7.12,v7.13,v7.14,v7.15,v7.16,v7.17,v7.18]}
{v8.6} = {v8.19} {v7.3[v7.2,v7.5,v7.8,v7.9,v7.11,v7.12,v7.14,v7.15, v7.16,
        v7.17,v7.18]}
{v8.7} = v1' [23] {v7.4[v7.0,v7.10,v7.11,v7.12,v7.13,v7.14,v7.15,v7.16,v7.17,
        v7.18]}
{v8.8} = [26] {v7.4,v7.5,v7.6[v7.0,v7.7,v7.8,v7.9,v7.10,v7.11,v7.12,v7.13,
        v7.14,v7.15,v7.16,v7.17,v7.18]}
{v8.9} = v1' [23] {v7.6[v7.0,v7.10,v7.11,v7.12,v7.13,v7.14,v7.15,v7.16,v7.17,
        v7.18]}
{v8.10} = [23] {v7.7[v7.0,v7.10,v7.11,v7.12,v7.13,v7.14,v7.15,v7.16,v7.17,
        v7.18]}
{v8.11} = [21] {v7.7,v7.8,v7.9[v7.0,v7.10,v7.11,v7.12,v7.13,v7.14,v7.15,
        v7.16,v7.17,v7.18]}
{v8.12} = [23] {v7.9[v7.0,v7.10,v7.11,v7.12,v7.13,v7.14,v7.15,v7.16,v7.17,
        v7.18]}
{v8.13} = [30] {v7.10□}
{v8.14} = [25] {v7.10,v7.12[v7.2,v7.5,v7.8,v7.11,v7.13,v7.14,v7.15, v7.16,
        v7.17,v7.18]} + [30] {v7.11,v7.12□}
{v8.15} = [29] {v7.13□}
{v8.16} = [22] {v7.13□} + [29] {v7.14□}
{v8.17} = v6' {v7.15□}
{v8.18} = v6 {v7.15□} + v6' {v7.16□}
{v8.19} = {v8.2} + {v8.14}' [28] [29] + v1' {v8.14}' [23] [27]
{v8.20} = {v7.18□}
[21] = [23] {v7.2,v7.5,v7.8[v7.0,v7.10,v7.11,v7.12,v7.13,v7.14,v7.15,v7.16,
        v7.17,v7.18]} + v2' v3' [25] {v7.1,v7.3,v7.4,v7.6,v7.7, v7.9[v7.2,v7.5,
        v7.8,v7.11,v7.13,v7.14,v7.15,v7.16,v7.17,v7.18]} + [24] [30]
[22] = v5' v6 + v5 v6'
[23] = v2' v3' [30] {v7.1,v7.2,v7.3,v7.4,v7.5,v7.6,v7.7,v7.8,v7.9□}
[24] = v2 v3 {v7.3,v7.6,v7.9} + v2' v3 {v7.1,v7.4,v7.7} + v2 v3' {v7.1,v7.4,
        v7.7}
[25] = v4' [22] {v7.0,v7.1,v7.3,v7.4,v7.6,v7.7,v7.9,v7.10□} + v4 [29] {v7.0,
        v7.1,v7.3,v7.4,v7.6,v7.7,v7.9,v7.10,v7.12□}
[26] = v1' [21] {v7.1,v7.2,v7.3,v7.4,v7.5,v7.6□} + v1 [23] {v7.1,v7.4□}
[27] = {v7.4,v7.6□} + v0' {v7.1,v7.3□}
[28] = {v7.0,v7.10,v7.13} + v3' v4' {v8.11}' {v7.7}
[29] = v5' v6'
[30] = v4' [29]

```

Figure 6.4: *keyb*: Circuit after Algebraic Decomposition

Encoded Implementation

Figure 6.5 shows the final encoded implementation with the code selection being done after algebraic decomposition. A five bit code is used for the nineteen values. The code bits are $[c0], [c1], \dots, [c4]$. The encoding does a very good job and exploits the don't cares in the incompletely specified literals. A good instance of this is in [21] where: $\mathcal{E}(v7^{\{1,3,4,6,7,9\}}[2,5,8,11,13,14,15,16,17,18]) = [c1]$.

```

{v8.0} = {v8.5}', {v8.8}', {v8.11}', {v8.12}', {v8.14}', {v8.16}', {v8.18}',
        {v8.19}', [c1] + {v8.5}', {v8.8}', {v8.11}', {v8.12}', {v8.14}', {v8.16}',
        {v8.18}', {v8.19}', [c2] + {v8.5}', {v8.8}', {v8.11}', {v8.12}', {v8.14}',
        {v8.16}', {v8.18}', {v8.19}', [c3] + {v8.5}', {v8.8}', {v8.11}', {v8.12}',
        {v8.14}', {v8.16}', {v8.18}', {v8.19}', [c4]
{v8.1} = v3' [30] [c0]', [c3]', [c4]
{v8.2} = v3' [25] [c3]', [c4]
{v8.3} = v3 [29] [c0]', [c3]', [c4]
{v8.4} = {v8.19} [c2]', [c3] [c4]',
{v8.5} = v0' [26] [c4]', + v0 v1' [23] [c0]', [c2]', [c3] [c4]',
{v8.6} = {v8.19} [c2] [c3] [c4]',
{v8.7} = v1' [23] [c0]', [c2]', [c4]
{v8.8} = [26] [c4]
{v8.9} = v1' [23] [c2] [c3] [c4]
{v8.10} = [23] [c2]', [c3]',
{v8.11} = [21] [c3]',
{v8.12} = [23] [c2] [c3]', [c4]',
{v8.13} = [30] [c1]', [c2] [c3] [c4]
{v8.14} = [25] [c1]', [c3] + [30] [c1]', [c2]', [c3] [c4]
{v8.15} = [29] [c1]', [c2] [c3]', [c4]',
{v8.16} = [29] [c1]', [c2]', [c3] [c4]', + [22] [c1]', [c2] [c3]', [c4]',
{v8.17} = v6' [c1]', [c2]', [c3]', [c4]',
{v8.18} = v6' [c2]', [c3]', [c4] + v6 [c1]', [c2]', [c3]', [c4]',
{v8.19} = {v8.2} + {v8.14}', [29] [c0]', [c1]', [c2] + v1' {v8.14}', [23] [c0]',
        [c4] + v0' v1' {v8.14}', [23] [c0]', [c3] + {v8.14}', [29] [c1]', [c2] [c3]',
        [c4]' + v3' v4' {v8.11}', {v8.14}', [29] [c1] [c2]', [c3]',
{v8.20} = [c0] [c1]', [c2] [c4]
[21] = [23] [c0] + v2' v3' [25] [c1] + v2 v3 [30] [c0]', [c1] [c2] + v2' v3
        [30] [c0]', [c1] [c2]', + v2 v3' [30] [c0]', [c1] [c2]',
[22] = v5' v6 + v5 v6'
[23] = v2' v3' [30] [c1]
[25] = v4 [29] [c0]' + v4' [22] [c0]', [c1] + v4' [22] [c0]', [c2]
[26] = v1' [21] [c1] [c3] + v1 [23] [c0]', [c2]', [c3]
[29] = v5' v6'
[30] = v4' [29]

```

Figure 6.5: *keyb*: Circuit after Encoding

It is a common experience that the more one knows about a subject, the more one is aware of what one does not know. This is particularly true in the case of practical experiences, where the complexity of the situation often leads to a deeper understanding of the limitations of one's knowledge.

One of the most important aspects of practical experience is the ability to recognize and learn from mistakes. It is through the process of trial and error that one gains the most valuable insights into the nature of a problem and the best way to solve it.

Another key element of practical experience is the development of a strong sense of responsibility. When one is involved in a practical task, one must take ownership of the outcome and be prepared to accept the consequences of one's actions.

Finally, practical experience teaches one the importance of communication. In many cases, the most effective way to solve a problem is through collaboration and the sharing of ideas and information with others.

Overall, practical experience is a valuable and essential part of learning. It provides a unique opportunity to apply theoretical knowledge to real-world situations and to develop the skills and attitudes necessary for success in any field.

It is through practical experience that one truly understands the value of knowledge and the importance of continuous learning. As one gains more experience, one becomes more confident and more capable of tackling the challenges of the world.

Chapter 7

Conclusions

*Thus grew the tale of Wonderland:
Thus slowly, one by one,
Its quaint events were hammered out –
and now the tale is done*

– Lewis Carroll, “Alice in Wonderland”

Recent research in logic synthesis has made a significant impact on the way digital circuits are designed today. However, current logic synthesis techniques focus only on combinational parts of the logic description. The research reported in this thesis attempts to overcome this limitation. In particular, extensions of known combinational logic optimization techniques are sought that are applicable in sequential logic synthesis. The motivation behind this is to maximize the leverage that can be obtained from the large body of research in combinational logic synthesis.

The first contribution of this work has been a clean formulation of the applicability of combinational logic optimization techniques across latch boundaries. A precise description of circuits for which latches may temporarily be removed is given. This enables any combinational logic optimization technique to be used on these circuits. The latch migration is defined in terms of the well-developed concept of retiming. This enables the interactions between retiming and combinational optimization to be easily examined. This has some interesting results in performance optimization.

In terms of practical experiences, the results in area optimization have not been encouraging. No additional advantage seems to be obtained by considering logical relationships across latch boundaries. However, this may be only a property of the circuits

examined or an artifact of the inability of current combinational optimization programs to get out of a local minimum. Additional work needs to be done in two areas. First, it is of interest to characterize circuits for which the additional information provided by retiming and resynthesis can be exploited. Second, stronger combinational logic optimization techniques are needed than those available currently. In performance optimization, these ideas have already shown applicability for pipelined circuits. Their extensions to arbitrary (non-pipelined) sequential circuits are currently being examined [72].

One inherent limitation of retiming and resynthesis is the inability to take into account the final positions of latches during combinational resynthesis. This is important in area optimization since any area reduction obtained by combinational optimization may be offset by an increase in the number of latches during subsequent retiming. This is less of a factor in performance optimization since the area increase does not affect the primary objective, *viz.*, improving the performance of the circuit. It must be pointed out that the techniques presented in [58], which have some commonality with retiming and resynthesis, do not have this limitation. They explicitly consider the cost of latches during logic optimization. However, in order to do this each combinational logic optimization technique needs to be redefined. As a result, existing programs cannot be used.

The second contribution of this thesis is the development of techniques for multi-level multiple-valued optimization and their application to the input encoding problem for multi-level logic. In addition to being a result in its own right it has application in sequential logic synthesis where the input encoding problem may be used to approximate the state assignment problem. The most significant developments here are the "algebraic" factorization techniques, the notion of incompletely specified literals, and the formulation of the minimum literal encoding problem. While initial results are promising, there are two areas in which work needs to be done to increase practical applicability. The first is a more accurate estimation of the size of algebraic factors. In the absence of this the sophisticated algebraic optimization techniques are of little advantage since they are not used correctly. The second is development of efficient heuristics to solve the minimum literal encoding problem. The current approach of using simulated annealing is too slow to be an acceptable solution. There is some work currently being done in this area [66]. Finally, in order to completely solve the state assignment problem in sequential logic synthesis, the output encoding problem for multi-level logic needs to be solved. This seems to be a very difficult problem and there is no obvious direction to pursue.

Appendix A

Discrete Mappings and Boolean Functions

“Please come back, and finish your story!” Alice called after it. And the others all joined in chorus “Yes, please do!”.

– Lewis Carroll, “Alice in Wonderland”

A.1 Introduction

Most functions needed to specify the behavior of digital logic systems are binary-valued functions of binary-valued variables ($\{0, 1\}^n \mapsto \{0, 1\}$). These are also referred to as *switching functions* [17]. The fact that all switching functions are also Boolean functions [17] enables all properties of Boolean functions to be directly applied to switching functions; these properties need not be proved separately. However not all functions that arise in the context of circuit specification and design are switching functions. In the most general form these functions are multiple-valued functions of multiple-valued variables. These functions are referred to as *discrete mappings*. However, these functions are not Boolean functions and hence properties of Boolean functions do not directly apply to them. This has two consequences. First, it requires that all properties of Boolean functions be re-examined to determine which of these hold for discrete mappings. Second, it results in an asymmetry between discrete mappings and switching functions which is theoretically inelegant. This chapter shows that corresponding to a discrete mapping there is a Boolean function which can be used to evaluate the discrete mapping in a direct way. In addition, a compact and

natural way of representing the Boolean formula corresponding to this function is presented. Conventional methods of representing discrete mappings as factored form representations are examined and their limitations pointed out. Finally it is shown that corresponding to a set of expressions representing a discrete mapping there exists an equivalent Boolean formula. This enables properties of Boolean formulas to be applied to these expressions. This is what researchers have been doing intuitively in the past; this chapter provides the justification for it.

A.2 Boolean Functions: A Review

This section provides a brief review of the background material needed in the rest of this chapter. This material has been taken from the text [17].

A.2.1 Boolean Algebras

Consider a quintuple:

$$(B, +, \cdot, 0, 1)$$

in which B is a set, called the carrier, $+$ and \cdot are binary operations on B , and 0 and 1 are distinct members of B . The algebraic system so defined is a *Boolean algebra* provided the following postulates are satisfied:

1. *Commutative Laws.* For all $a, b \in B$:

$$a + b = b + a$$

$$a \cdot b = b \cdot a$$

2. *Distributive Laws.* For all $a, b, c \in B$:

$$a + (b \cdot c) = (a + b) \cdot (a + c)$$

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

3. *Identities.* For all $a \in B$:

$$0 + a = a$$

$$1 \cdot a = a$$

4. *Complements.* For any $a \in B$, there is a unique element $a' \in B$ such that:

$$a + a' = 1$$

$$a \cdot a' = 0$$

The following properties are true for all $a, b, c \in B$. These are useful in manipulating Boolean expressions.

1. *Associativity.*

$$a + (b + c) = (a + b) + c$$

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

2. *Idempotence.*

$$a + a = a$$

$$a \cdot a = a$$

3.

$$a + 1 = 1$$

$$a \cdot 0 = 0$$

4. *Absorption.*

$$a + (a \cdot b) = a$$

$$a \cdot (a + b) = a$$

5. *Involution.*

$$(a')' = a$$

6. *De Morgan's Laws.*

$$(a + b)' = a' \cdot b'$$

$$(a \cdot b)' = a' + b'$$

7.

$$a + a' \cdot b = a + b$$

$$a \cdot (a' + b) = a \cdot b$$

8. *Consensus.*

$$a \cdot b + a' \cdot c + b \cdot c = a \cdot b + a' \cdot c$$

$$(a + b) \cdot (a' + c) \cdot (b + c) = (a + b) \cdot (a' + c)$$

A.2.2 Boolean Formulas

Given a Boolean algebra B , the set of *Boolean formulas* on the n symbols x_1, x_2, \dots, x_n is defined by the following rules:

1. The elements of B are Boolean formulas.
2. The symbols x_1, x_2, \dots, x_n are Boolean formulas.
3. If g and h are Boolean formulas, then so are:

$$(a) (g) + (h)$$

$$(b) (g) \cdot (h)$$

$$(c) (g)'$$

4. A string is a Boolean formula if and only if its being so follows from finitely many applications of the rules above.

A.2.3 Boolean Functions

An n -variable function $f : B^n \mapsto B$ is called a Boolean function if and only if it can be expressed as an n -variable Boolean formula.

A.3 Discrete Mappings

Let $f : P_0 \times P_1 \times \dots \times P_{n-1} \mapsto P_n$ be a discrete mapping with $P_j = \{0, 1, \dots, p_{j-1}\}$. Let $P = \{P_0 \times P_1 \times \dots \times P_n\}$. f is not a Boolean function since it does not meet the condition that $f : B^n \mapsto B$ for some Boolean algebra B . Corresponding to f there is the relation

$R \subseteq P$ defined in the natural way as the set of points in P consistent with f . Let $B = 2^P$, the power set of P , i.e. the set of all subsets of P . B is a Boolean algebra described by $(2^P, \cup, \cap, \phi, P)$. Let $\xi : B \mapsto B$ be defined as:

$$\xi(x) = R \cap x \quad x \in B \quad (\text{A.1})$$

$R \cap x$ is a Boolean formula and hence ξ is a Boolean function. Equation A.1 is the *minterm canonical form* for this function.

Let $m \in \{P_0 \times P_1 \times \dots \times P_{n-1}\}$ and $\psi(m) = \{m\} \times P_n$. $\psi(m)$ is the set of $n+1$ -tuples corresponding to the n -tuple m that have all p_n possible values in the last field. ξ corresponds to f in the sense that given any m , $f(m)$ may be computed by ξ as follows. $\xi(\psi(m))$ is a singleton set containing the tuple in R with the first n fields the same as that of m . Field $n+1$ in this tuple is $f(m)$.

Example A.3.1 *The switching function corresponding to an AND gate is used to illustrate the above. Here $f : \{0, 1\}^2 \mapsto \{0, 1\}$. Consider $m = (0, 1)$.*

$$\begin{aligned} R &= \{(0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 1)\} \\ \psi(m) &= \{(0, 1)\} \times \{0, 1\} \\ &= \{(0, 1, 0), (0, 1, 1)\} \\ \xi(\psi(m)) &= \{(0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 1)\} \cap \{(0, 1, 0), (0, 1, 1)\} \\ &= \{(0, 1, 0)\} \end{aligned}$$

$f(m)$ is the last field of the $n+1$ -tuple $(0, 1, 0)$, i.e. $f(m) = 0$.

Example A.3.2 *Each person in a certain western town is to be classified as being one of {good, bad, ugly} (abbreviated as {g, b, u}). This classification is to be done based on the person's occupation which is one of {priest, teacher, outlaw} (abbreviated as {p, t, o}) and their nature which is one of {honest, selfish, cruel} (abbreviated as {h, s, c}). To be good you have to be a priest or be honest and not an outlaw. Cruel outlaws are ugly. Everyone else is just bad.*

The classification function is a discrete mapping $f : \{p, t, o\} \times \{h, s, c\} \mapsto \{g, b, u\}$. Consider $m = (t, c)$.

$$R = \{(p, h, g), (p, s, g), (p, c, g), (t, h, g), (t, s, b), (t, c, b), (o, h, b), (o, s, b), (o, c, u)\}$$

$$\begin{aligned}
\psi(m) &= \{(t, c)\} \times \{g, b, u\} \\
\xi(\psi(m)) &= R \cap \psi(m) \\
&= \{(t, c, b)\}
\end{aligned}$$

$f(m)$ is the last field of the $n + 1$ -tuple (t, c, b) , i.e. $f(m) = b$.

A.4 Compact Representations of Boolean Functions

Typically functions need to be represented in some compact manner; it is not possible to describe them in terms of all the points in the domain and the corresponding range. Compact representations cluster points in the domain and describe f in terms of these clusters. Generally, these clusters are fewer than the number of points in the domain.

As in Section A.3 let $B = 2^P$ and $m \in P$. Let $m[j]$ be the value of field j in m . One natural way to cluster points in P is to group all points with the same value of $m[j]$ (for some given j) together and refer to them collectively. Let $\chi_j^{S_j} = P_0 \times \dots \times P_{j-1} \times S_j \times P_{j+1} \times \dots \times P_n$. Thus, $\chi_j^{S_j}$ has all points for which $m[j] \in S_j$. For Example A.3.2 $\chi_0^{\{p\}}$ is the set of all points for which $m[0] = p$. Note that $\chi_j^{S_j} \in B$ and $\overline{\chi_j^{S_j}} = \chi_j^{P_j - S_j}$.

Theorem A.4.1 *Let $\chi = \{\chi_j^{S_j} | j \in \{0, 1, \dots, n\}, S_j \subseteq P_j\}$. Let $b \in B$. b can be expressed in terms of a Boolean expression restricted to elements of χ .*

Proof. The statement needs to be proven only for the atoms of B , (the singleton sets) since any other element of B can be obtained by a union of the atoms. Let $\{a\}$ be an atom and $a[i]$ be field i of a . $a = \cap_i (\chi_i^{\{a[i]\}})$. ■

An immediate corollary of this result is that R can be expressed as a Boolean expression restricted to elements of χ . Thus the Boolean formula in Equation A.1 can be re-written by expressing R as a Boolean expression restricted to the elements of χ . In practice Theorem A.4.1 is not used to re-write R , but rather R is derived directly from some description of the function.

Consider f in Example A.3.2. R is derived directly from the conditions specified as follows. The set of points in P that represent priests or honest people who are not outlaws is naturally expressed as:

$$\chi_0^{\{p\}} \cup \left(\chi_1^{\{h\}} \cap \overline{\chi_0^{\{o\}}} \right)$$

This can be simplified to:

$$\chi_0^{\{p\}} \cup (\chi_1^{\{h\}} \cap \chi_0^{\{t\}})$$

Similarly the set of points that represent cruel outlaws is: $\chi_0^{\{o\}} \cap \chi_1^{\{c\}}$. The rest of the people are obviously expressed as:

$$\overline{\chi_0^{\{p\}} \cup (\chi_1^{\{h\}} \cap \chi_0^{\{t\}}) \cup (\chi_0^{\{o\}} \cap \chi_1^{\{c\}})}$$

This can be simplified to:

$$(\chi_0^{\{t\}} \cap \chi_1^{\{s,c\}}) \cup (\chi_0^{\{o\}} \cap \chi_1^{\{h,s\}})$$

Thus, R can be expressed as:

$$(\chi_2^{\{g\}} \cap (\chi_0^{\{p\}} \cup (\chi_1^{\{h\}} \cap \chi_0^{\{t\}}))) \cup (\chi_2^{\{b\}} \cap ((\chi_0^{\{t\}} \cap \chi_1^{\{s,c\}}) \cup (\chi_0^{\{o\}} \cap \chi_1^{\{h,s\}}))) \cup (\chi_2^{\{u\}} \cap (\chi_0^{\{o\}} \cap \chi_1^{\{c\}}))$$

A.5 Expressions Representing Discrete Mappings

Another way to cluster the points in the domain is to partition the domain based on the value of the function. Let $\Pi = \{\pi_0, \pi_1, \dots, \pi_{p_n-1}\}$ be a partition of $P_0 \times P_1 \times \dots \times P_{n-1}$ such that: $m \in \pi_i \Leftrightarrow f(m) = i$. For the switching function f in Example A.3.1, $\pi_0 = \{(0,0), (0,1), (1,0)\}$ and $\pi_1 = \{(1,1)\}$.

Each π_i may be described by its characteristic function \tilde{f}_i defined as follows.

$$\tilde{f}_i : P_0 \times P_1 \times \dots \times P_{n-1} \mapsto \{0, 1\} \quad i \in P_n$$

$$\tilde{f}_i(m) = \begin{cases} 1 & \text{if } m \in \pi_i \\ 0 & \text{otherwise} \end{cases}$$

\tilde{f}_i tests for membership in π_i ; it evaluates to 1 for exactly the points in π_i . The following representation has been commonly used to describe the \tilde{f}_i in the literature. Let $S_j \subseteq P_j$ and X_j be a p_j -valued variable. $X_j^{S_j}$ is termed a literal of X_j and is defined as:

$$X_j^{S_j}(m) = \begin{cases} 1 & \text{if } m[j] \in S_j \\ 0 & \text{otherwise} \end{cases}$$

$X_{j_1}^{S_{j_1}} \cdot X_{j_2}^{S_{j_2}}$ is defined as the logical AND of $X_{j_1}^{S_{j_1}}$ and $X_{j_2}^{S_{j_2}}$. Similarly, $X_{j_1}^{S_{j_1}} + X_{j_2}^{S_{j_2}}$ is defined as the logical OR of $X_{j_1}^{S_{j_1}}$ and $X_{j_2}^{S_{j_2}}$. The complement of a literal $X_j^{S_j}$ is denoted as $\overline{X_j^{S_j}}$ and defined as $\overline{X_j^{S_j}} = X_j^{P_j - S_j}$. A factored form with these literals is defined recursively by the following rules:

1. A literal is a factored form.
2. The complement of a literal is a factored form.
3. The $+$ of two factored forms is a factored form.
4. The \cdot of two factored forms is a factored form.

A factored form, \mathcal{F}_i may be used to represent the \tilde{f}_i .

Example A.5.1 *An example of a factored form is:*

$$(X_1^{\{0,1\}} X_2^{\{2\}} + X_2^{\{1\}}) \cdot X_3^{\{0\}}$$

For f in Example A.3.1 the following are the factored form representations of the \tilde{f}_i :

$$\begin{aligned}\tilde{f}_0 &= X_0^{\{0\}} + X_1^{\{0\}} \\ \tilde{f}_1 &= X_0^{\{1\}} \cdot X_1^{\{1\}}\end{aligned}$$

For f in Example A.3.2 the following are the factored form representations for \tilde{f}_g and \tilde{f}_u :

$$\begin{aligned}\tilde{f}_g &= X_0^{\{p\}} \cup \left(X_1^{\{h\}} \cap \overline{X_0^{\{o\}}} \right) \\ \tilde{f}_u &= X_0^{\{o\}} \cap X_1^{\{c\}}\end{aligned}$$

However, there seems to be no direct way to obtain \tilde{f}_b since these expressions are not Boolean and De Morgan's Laws cannot be directly applied in this case. If they do hold then it must be proven separately for these expressions. This is a limitation of this representation.

In Section A.3 it was shown how the Boolean function ξ may be obtained given the relation R for the discrete mapping. The rest of this section examines how the ξ may be obtained if the discrete mapping is specified as factored form expressions representing the characteristic functions of the partitions. This is accomplished by first deriving a Boolean formula for each factored form.

Let \mathcal{B} be a Boolean formula generator; i.e. given a factored form \mathcal{F} , $\mathcal{B}(\mathcal{F})$ is a Boolean formula obtained from \mathcal{F} .

\mathcal{B} is defined recursively as follows:

1. $\mathcal{B}(X_j^{S_j}) = x_j^{S_j} \cap x$

$$2. \mathcal{B}(X_{j_1}^{S_{j_1}} + X_{j_2}^{S_{j_2}}) = \mathcal{B}(X_{j_1}^{S_{j_1}}) \cup \mathcal{B}(X_{j_2}^{S_{j_2}})$$

$$3. \mathcal{B}(X_{j_1}^{S_{j_1}} \cdot X_{j_2}^{S_{j_2}}) = \mathcal{B}(X_{j_1}^{S_{j_1}}) \cap \mathcal{B}(X_{j_2}^{S_{j_2}})$$

Note that $\mathcal{B}(\overline{X_j^{S_j}}) = \overline{\chi_j^{S_j}} \cap x$. This follows from:

$$\begin{aligned} \overline{X_j^{S_j}} &= X_j^{P_j - S_j} \\ \mathcal{B}(\overline{X_j^{S_j}}) &= \mathcal{B}(X_j^{P_j - S_j}) \\ &= (P_0 \times P_1 \times \dots \times P_{j-1} \times (P_j - S_j) \times P_{j+1} \times \dots \times P_{n-1} \times P_n) \cap x \\ &= (P - (P_0 \times P_1 \times \dots \times P_{j-1} \times S_j \times P_{j+1} \times \dots \times P_{n-1} \times P_n)) \cap x \\ &= \overline{\chi_j^{S_j}} \cap x \end{aligned}$$

In the following discussion when $\mathcal{B}(\mathcal{F})$ and \mathcal{F} are followed by arguments they are being used in place of the functions they represent. The following result shows how $\mathcal{B}(\mathcal{F})$ can be used to evaluate \mathcal{F} .

Theorem A.5.1

$$\mathcal{F}(m) = 0 \Rightarrow (\mathcal{B}(\mathcal{F})(\psi(m)) = \phi)$$

$$\mathcal{F}(m) = 1 \Rightarrow (\mathcal{B}(\mathcal{F})(\psi(m)) = \psi(m))$$

Proof. The proof is by induction on the number of literals in \mathcal{F} .

Induction Hypothesis: The theorem statement is true for all \mathcal{F} with $< k$ literals.

Induction Basis: Consider \mathcal{F} with only one literal, i.e. $\mathcal{F} = X_j^{S_j}$.

1. $\mathcal{F}(m) = 0$. Then $X_j^{S_j}(m) = 0$, which implies $\chi_j^{S_j} \cap \psi(m) = \phi$ since field j of no tuple in $\chi_j^{S_j}$ is the same as field j of any tuple in $\psi(m)$.
2. $\mathcal{F}(m) = 1$. Then $X_j^{S_j}(m) = 1$, which implies $\chi_j^{S_j} \cap \psi(m) = \psi(m)$ since $\chi_j^{S_j}$ contains all tuples with the same field j as that of m .

Induction Step: Consider \mathcal{F} with k literals. There are two cases.

1. $\mathcal{F} = \mathcal{F}_1 + \mathcal{F}_2$

\mathcal{F}_1 and \mathcal{F}_2 have $< k$ literals.

(a) $\mathcal{F}(m) = 0$. Then $\mathcal{F}_1(m) = 0$ and $\mathcal{F}_2(m) = 0$. By the induction hypothesis,
 $\mathcal{B}(\mathcal{F}_1)(\psi(m)) \cup \mathcal{B}(\mathcal{F}_2)(\psi(m)) = \phi$.

(b) $\mathcal{F}(m) = 1$. Then $\mathcal{F}_1(m) = 1$ or $\mathcal{F}_2(m) = 1$. By the induction hypothesis,
 $\mathcal{B}(\mathcal{F}_1)(\psi(m)) \cup \mathcal{B}(\mathcal{F}_2)(\psi(m)) = \psi(m)$.

2. $\mathcal{F} = \mathcal{F}_1 \cdot \mathcal{F}_2$

\mathcal{F}_1 and \mathcal{F}_2 have $< k$ literals.

(a) $\mathcal{F}(m) = 1$. Then $\mathcal{F}_1(m) = 1$ and $\mathcal{F}_2(m) = 1$. By the induction hypothesis,
 $\mathcal{B}(\mathcal{F}_1)(\psi(m)) \cap \mathcal{B}(\mathcal{F}_2)(\psi(m)) = \psi(m)$.

(b) $\mathcal{F}(m) = 0$. Then $\mathcal{F}_1(m) = 0$ or $\mathcal{F}_2(m) = 0$. By the induction hypothesis,
 $\mathcal{B}(\mathcal{F}_1)(\psi(m)) \cap \mathcal{B}(\mathcal{F}_2)(\psi(m)) = \phi$.

■

This result establishes the correspondence between $\mathcal{B}(\mathcal{F})$ and \mathcal{F} . For the factored form in Example A.5.1:

$$\begin{aligned} \mathcal{B}((X_1^{\{0,1\}} X_2^{\{2\}} + X_2^{\{1\}}) \cdot X_3^{\{0\}}) &= (\chi_1^{\{0,1\}} \cap x \cap \chi_2^{\{2\}} \cap x \cup \chi_2^{\{1\}} \cap x) \cap \chi_3^{\{0\}} \cap x \\ &= (\chi_1^{\{0,1\}} \cap \chi_2^{\{2\}} \cup \chi_2^{\{1\}}) \cap \chi_3^{\{0\}} \cap x \end{aligned}$$

Note the similarity between the two expressions. There is an obvious one-to-one mapping from the factored form to the Boolean formula. If the Boolean formula is expressed in terms of the χ 's then the reverse mapping is also one-to-one. This is significant since it shows that all properties for Boolean formulas (same as those for Boolean expressions) hold for factored form expressions and they need not be proven separately. Thus complementation using De Morgan's Laws also holds for factored form expressions.

In the above discussion only a single factored form was considered. Recollect that f is described in terms of a factored form \mathcal{F}_i for each \tilde{f}_i . The following theorem shows how the Boolean formulas for these factored forms are combined to give $\xi(x)$ defined in Section A.3.

Theorem A.5.2

$$\xi(x) = \cup_i \left[(\mathcal{B}(\mathcal{F}_i)) \cap \chi_n^{\{i\}} \right] \quad i \in P_n$$

Proof. Let $f(m) = j$. By Theorem A.5.1:

$$\mathcal{B}(\mathcal{F}_i)(\psi(m)) = \begin{cases} \phi & i \neq j \\ \psi(m) & i = j \end{cases}$$

Thus, $\xi(\psi(m)) = \psi(m) \cap \chi_n^{\{j\}}$. This is a singleton set in which the $n + 1$ -tuple has the first n fields the same as m and the last field is j . This is consistent with the definition given in Equation A.1. ■

For the switching function in Example A.3.1:

$$\begin{aligned} \chi_0^{\{0\}} &= \{(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1)\} \\ \chi_1^{\{0\}} &= \{(0, 0, 0), (0, 0, 1), (1, 0, 0), (1, 0, 1)\} \\ \chi_0^{\{1\}} &= \{(1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)\} \\ \chi_1^{\{1\}} &= \{(0, 1, 0), (0, 1, 1), (1, 1, 0), (1, 1, 1)\} \\ \chi_2^{\{0\}} &= \{(0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 0)\} \\ \chi_2^{\{1\}} &= \{(0, 0, 1), (0, 1, 1), (1, 0, 1), (1, 1, 1)\} \\ \xi(x) &= \left((\chi_0^{\{0\}} \cup \chi_1^{\{0\}}) \cap x \cap \chi_2^{\{0\}} \right) \cup \left((\chi_0^{\{1\}} \cap \chi_1^{\{1\}}) \cap x \cap \chi_2^{\{1\}} \right) \\ &= \{(0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 1)\} \cap x \end{aligned}$$

This, as expected, is the same as that derived in Example A.3.1.

A.6 Conclusions

In Section A.3, it was shown how a Boolean function may be obtained given a discrete mapping. Next, in Section A.4, it was described how the Boolean formula corresponding to this Boolean function may be expressed in compact form. This Boolean formula may be manipulated using any property of Boolean expressions. This has direct application in the simplification of Boolean expressions, both in sum-of-product form as well as in factored form. In Section A.5, conventional methods for representing discrete mappings were presented. Since these do not involve Boolean expressions or Boolean functions, none of their properties can be directly used, but rather must be proved separately. Subsequently, it was demonstrated how these may be converted to the representation presented in Section A.4 in order to exploit the properties of Boolean expressions.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let \mathcal{X} and \mathcal{Y} be sets.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function. Let \mathcal{X} and \mathcal{Y} be sets.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function.

Bibliography

- [1] D. B. Armstrong. A programmed algorithm for assigning internal codes to sequential machines. *IRE Transactions on Electron Computers*, EC-11:466–472, August 1962.
- [2] P. Ashar, S. Devadas, and A. R. Newton. Irredundant interacting sequential machines via optimal logic synthesis. *IEEE Transactions on Computer-Aided Design*, March 1991. To appear.
- [3] K. Bartlett, R. K. Brayton, G. D. Hachtel, R. Jacoby, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic minimization using implicit don't cares. *IEEE Transactions on Computer-Aided Design*, CAD-7(6):723–740, June 1988.
- [4] K. A. Bartlett, G. Boriello, and S. Raju. Timing optimization of multi-phase sequential logic. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 356–366, January 1990. To appear, *IEEE Transactions on Computer-Aided Design*, January 1991.
- [5] T. Ben-Tzur. Personal Communication, 1989.
- [6] C. L. Berman, J. L. Carter, and K. F. Day. The fanout problem: From theory to practice. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the 1989 Decennial Caltech Conference*, pages 69–99. MIT Press, March 1989.
- [7] C. L. Berman and L. Trevillyan. A global approach to circuit size reduction. In J. Allen and F. T. Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*, pages 203–214. MIT Press, March 1988.

- [8] D. Bostick, G. D. Hachtel, R. Jacoby, M. R. Lightner, P. Moceyunas, C. R. Morrison, and D. Ravenscroft. The Boulder Optimal Logic Design system. In *Proceedings of the International Conference on Computer-Aided Design*, pages 62–65, November 1987.
- [9] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.
- [10] R. K. Brayton and C. McMullen. The decomposition and factorization of Boolean expressions. In *Proceedings of the International Symposium on Circuits and Systems*, pages 49–54, May 1982.
- [11] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic optimization and the rectangular covering problem. In *Proceedings of the International Conference on Computer-Aided Design*, pages 66–69, November 1987.
- [12] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.
- [13] R. K. Brayton and F. Somenzi. Boolean relations and the incomplete specification of Boolean networks. In *Proceedings of the International Conference on Very Large Scale Integration*, pages 231–240, August 1989.
- [14] M. A. Breuer. Generation of optimal code for expressions via factorization. *Communications of the ACM*, 12(6):333–340, June 1969.
- [15] M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems*. Computer Science Press, Woodland Hills, CA, 1976.
- [16] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles for sequential benchmark circuits. In *Proceedings of the International Symposium on Circuits and Systems*, May 1989.
- [17] F. M. Brown. *Boolean Reasoning*. Kluwer Academic Publishers, 1990.
- [18] R. E. Bryant. Graph based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):667–691, August 1986.

- [19] K-T. Cheng and V.D. Agrawal. Design of sequential machines for efficient test generation. In *Proceedings of the International Conference on Computer-Aided Design*, pages 358–361, November 1989.
- [20] M. Dagenais, V. Agarwal, and N. Rumin. McBoole: A new procedure for exact logic minimization. *IEEE Transactions on Computers*, C-33:229–238, January 1986.
- [21] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L. Trevillyan. LSS: A system for production logic synthesis. *IBM Journal of Research and Development*, 28(5):326–328, September 1984.
- [22] S. Devadas. Approaches to multi-level sequential logic synthesis. In *Proceedings of the Design Automation Conference*, pages 270–276, June 1989.
- [23] S. Devadas and K. Keutzer. Boolean minimization and algebraic factorization procedures for fully testable sequential machines. In *Proceedings of the International Conference on Computer-Aided Design*, pages 208–211, November 1989.
- [24] S. Devadas and K. Keutzer. Necessary and sufficient conditions for robust delay fault testability. In W. J. Dally, editor, *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, pages 221–238. MIT Press, April 1990.
- [25] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. MUSTANG: State assignment of finite state machines targeting multi-level logic implementations. *IEEE Transactions on Computer-Aided Design*, CAD-7(12):1290–1300, December 1988.
- [26] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Optimal logic synthesis and testability: Two faces of the same coin. In *Proceedings of the International Testing Conference*, pages 3–13, September 1988.
- [27] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. A synthesis and optimization procedure for fully and easily testable sequential machines. *IEEE Transactions on Computer-Aided Design*, CAD-8(10):1100–1107, October 1989.
- [28] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Irredundant sequential machines via optimal logic synthesis. *IEEE Transactions on Computer-Aided Design*, CAD-9(1):8–18, January 1990.

- [29] S. Devadas and A. R. Newton. Decomposition and factorization of sequential finite state machines. *IEEE Transactions on Computer-Aided Design*, CAD-8(11):1206–1217, November 1989.
- [30] S. Devadas and A. R. Newton. Exact algorithms for output encoding, state assignment and four-level Boolean minimization. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 387–396, January 1990. To appear, *IEEE Transactions on Computer-Aided Design*, January 1991.
- [31] M. R. Garey and D. S. Johnson. *Computers and Intractability*, page 222. W. H. Freeman and Company, 1979.
- [32] D. Gregory, K. Bartlett, A. DeGeus, and G. Hachtel. SOCRATES: A system for automatically synthesizing and optimizing combinational logic. In *Proceedings of the Design Automation Conference*, pages 79–85, June 1986.
- [33] R. Gupta, R. Gupta, and M. A. Breuer. BALLAST: A methodology for partial scan design. In *Proceedings of the International Symposium on Fault Tolerant Computing*, pages 118–125, June 1989.
- [34] G. Hachtel, X. Du, and P. Moceyunas. Algorithms for state assignment based on multi-level representation. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 367–376, January 1990. To appear, *IEEE Transactions on Computer-Aided Design*, January 1991.
- [35] G. D. Hachtel and R. M. Jacoby. Verification algorithms for VLSI synthesis. *IEEE Transactions on Computer-Aided Design*, CAD-7(5):616–640, May 1988.
- [36] G. D. Hachtel, R. M. Jacoby, K. Keutzer, and C. R. Morrison. On the properties of algebraic transformations and the multifault testability of multilevel logic. In *Proceedings of the International Conference on Computer-Aided Design*, pages 422–425, November 1989.
- [37] J. Hartmanis. On the state assignment problem for sequential machines I. *IRE Transactions on Electronic Computers*, pages 157–165, June 1961.
- [38] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, Englewood Cliffs, N. J., 1966.

- [39] R. M. Karp. Some techniques for the state assignment of synchronous sequential machines. *IEEE Transactions on Electron Computers*, EC-13:507–518, October 1964.
- [40] K. Keutzer, S. Malik, and A. Saldanha. Is redundancy necessary to reduce delay? In *Proceedings of the Design Automation Conference*, pages 228–234, June 1990. Accepted for publication, *IEEE Transactions on Computer Aided Design*.
- [41] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.
- [42] L. Lavagno, S. Malik, R. K. Brayton, and A. Sangiovanni-Vincentelli. MIS-MV: Optimization of multi-level logic with multiple-valued inputs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 560–563, November 1990.
- [43] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing synchronous circuitry by retiming. In R. E. Bryant, editor, *Advanced Research in VLSI: Proceedings of the Third Caltech Conference*, pages 23–36. Computer Science Press, 1983.
- [44] C. E. Leiserson and J. B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, Spring 1983.
- [45] B. Lin and A. R. Newton. Synthesis of multiple level logic from symbolic high-level description languages. In *Proceedings of the International Conference on Very Large Scale Integration*, pages 187–196, August 1989.
- [46] B. Lin and A.R. Newton. Restructuring state machines and state assignment: Relationship to minimizing logic across latch boundaries. In *International Workshop on Logic Synthesis*, May 1989.
- [47] R. Lisanke. Logic synthesis and optimization benchmarks: User's guide, 1989. Address: MCNC - P.O. Box 12889 - Research Triangle Park - NC 27709.
- [48] S. Malik, R. K. Brayton, and A. Sangiovanni-Vincentelli. Encoding symbolic inputs for multi-level logic implementation. In *Proceedings of the International Conference on Very Large Scale Integration*, pages 221–230, 1989.
- [49] S. Malik, E. Sentovich, R. K. Brayton, and A. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimization of sequential networks with combinational techniques.

- In *Proceedings of the Hawaii International Conference on System Sciences*, pages 397–406, January 1990. To appear, *IEEE Transactions on Computer-Aided Design*, January 1991.
- [50] S. Malik, K. J. Singh, R. K. Brayton, and A. Sangiovanni-Vincentelli. Performance optimization of pipelined circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 410–413, November 1990.
- [51] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proceedings of the International Conference on Computer-Aided Design*, pages 6–9, November 1988.
- [52] E. McCluskey. Minimization of Boolean functions. *Bell System Technical Journal*, 35:1417–1444, April 1956.
- [53] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.
- [54] C. Mead and L. Conway. *Introduction to VLSI Systems*, chapter 2, pages 45–46. Addison Wesley, 1980.
- [55] C. Mead and L. Conway. *Introduction to VLSI Systems*, chapter 3, pages 85–86. Addison Wesley, 1980.
- [56] G. De Micheli. Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros. *IEEE Transactions on Computer-Aided Design*, CAD-5(4):597–616, October 1986.
- [57] G. De Micheli. Performance-oriented synthesis of large-scale domino CMOS circuits. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):751–765, 1987.
- [58] G. De Micheli. Logic transformations for synchronous logic synthesis. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 407–416, January 1990. To appear, *IEEE Transactions on Computer-Aided Design*, January 1991.
- [59] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design*, CAD-4(3):269–285, July 1985.

- [60] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The transduction method - Design of logic networks based on permissible functions. *IEEE Transactions on Computers*, C-38(10):1404-1424, October 1989.
- [61] W. Quine. The problem of simplifying truth functions. *American Math. Monthly*, 59:521-531, 1952.
- [62] J. Roth and R. Karp. Minimization over Boolean graphs. *IBM Journal of Research and Development*, 6(2):227-238, April 1962.
- [63] R. Rudell. *Logic synthesis for VLSI Design*. PhD thesis, University of California, Berkeley, April 1989.
- [64] R. L. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):727-750, September 1987.
- [65] K. A. Sakallah, T. N. Mudge, and O. A. Olukotun. checkTc and minTc: Timing verification and optimal clocking of synchronous digital circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 552-555, November 1990.
- [66] A. Saldanha, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. A framework for satisfying input and output encoding constraints. Submitted to, Design Automation Conference, 1991.
- [67] A. Saldanha, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Multi-level logic simplification using don't cares and filters. In *Proceedings of the Design Automation Conference*, pages 272-282, June 1989.
- [68] H. Savoj and R. K. Brayton. The use of observability and external don't cares for the simplification of multi-level networks. In *Proceedings of the Design Automation Conference*, pages 297-301, June 1990.
- [69] R. B. Segal. BDSYN: Logic description translator; BDSIM: Switch level simulator. Master's Thesis M87/33, Electronics Research Lab., University of California, Berkeley. May 1987.
- [70] E. Sentovich. SIS: An interactive system for the synthesis of sequential logic circuits. Unpublished Manuscript.

- [71] N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli. Retiming of circuits with single-phase transparent latches. In preparation.
- [72] K. J. Singh. *Performance Optimization of Digital Circuits*. PhD thesis, University of California, Berkeley. In Preparation.
- [73] K. J. Singh, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *Proceedings of the International Conference on Computer-Aided Design*, pages 282–285, November 1988.
- [74] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *Proceedings of the International Conference on Computer-Aided Design*, pages 92–95, November 1990.
- [75] A. Stoelzle, S. Narayanaswamy, K. Kornegay, H. Murveit, and J. Rabaey. A VLSI wordprocessing subsystem for a real time large vocabulary continuous speech recognition system. In *Proceedings of the Custom Integrated Circuits Conference*, 1989.
- [76] H. Touati and R. K. Brayton. Computing initial states of retimed circuits. Unpublished Manuscript.
- [77] H. Touati, C. Moon, R. K. Brayton, and A. Wang. Performance-oriented technology mapping. In W. J. Dally, editor, *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, pages 79–97. MIT Press, April 1990.
- [78] J. Tracey. Internal state assignment for asynchronous sequential machines. *IRE Transactions on Electronic Computers*, pages 551–560, August 1966.
- [79] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State assignment of finite state machines for optimal two-level logic implementations. In *Proceedings of the Design Automation Conference*, pages 327–332, June 1989.
- [80] S. Yang and M. Ciesielski. On the relationship between input encoding and logic minimization. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 377–386. January 1990. To appear. *IEEE Transactions on Computer-Aided Design*, January 1991.