Copyright © 1989, 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.

ALGORITHMS FOR MULTI-LEVEL LOGIC OPTIMIZATION

by

Albert Ren Rui Wang

Memorandum No. UCB/ERL M89/50

28 April 1989

COVER PAGE

ALGORITHMS FOR MULTI-LEVEL LOGIC OPTIMIZATION

by

Albert Ren Rui Wang

Memorandum No. UCB/ERL M89/50

28 April 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Algorithms for Multi-level Logic Optimization

Albert Ren Rui Wang

Ph.D. Department of Electrical Engineering and Computer Science

Abstract

Accompanied with the recent advancement in integrated circuit technology is the need for an automatic multi-level logic optimization tool. Such a tool must be able to explore the entire design space, understanding different design styles and making appropriate tradeoffs. This thesis presents algorithms in four areas of multi-level logic optimization: factoring logic functions, simplification of logic functions, global phase assignment, and technology-independent timing optimization. All problems are provided with abstractions for better understanding and ease of analysis. The algorithms are used in various phases of the optimization process and are components in the multi-level logic optimization system MIS.

1. Factored forms have been shown to be a useful abstraction of logic functions in the multi-level logic design style. Algorithms will be given for deriving the factored forms from the sum-of-products forms of logic functions. In addition, a system is proposed in which logic functions can be manipulated directly in an efficient way, thus providing a stable internal representation for a multi-level logic optimization system. Certain properties of optimal factored forms will also be investigated which could lead to more powerful factoring algorithms.

2. A combinational circuit can be abstracted as a Boolean network containing a set of logic functions. A procedure will be given to simplify a function using a don't-care set derived from its environment. Large effort has gone into making the procedure more efficient, which involves filtering out the useless or "almost" useless part of the don't-care set and finding a better representation for the don't-care set.

3. Phase assignment is the generalization of the inverter minimization problem, an NP-hard problem. An integer programming formulation is used to derive an exact algorithm to solve the problem optimally for circuits with some special properties. A set of heuristic algorithms will be given to solve the problem with varying quality-versus-performance tradeoffs.

4. Timing optimization has always been an essential part of any multi-level logic optimization system. This thesis concentrates on the technology independent aspect of timing optimization, i.e., global circuit re-structuring. An algorithm will be given which incrementally improves the global structure of the circuit. Emphasis is placed on one step of the algorithm, timing-driven decomposition. A technique for speeding up the optimization process, incremental delay trace, will be presented.

Most of the algorithms presented in this thesis have been implemented and tested, and installed as part of MIS. Experimental results will be given to illustrate the efficiency of the algorithms.

Prof. Alberto Sangiovanni-Vincentelli
Thesis Committee Chairman

Acknowledgments

I am indebted to my research advisor Alberto Sangiovanni-Vincentelli for his unfailing support and constant encouragement, guidance, and interest to all that I did during the course of this research. I am grateful to Alberto for taking his precious time reading the early draft of this thesis and providing invaluable suggestions and comments.

I am truly fortunate to have been associated with Professor Robert Brayton who has been a constant source of inspiration. I am indebted to Bob for many stimulating discussions, numerous profitable ideas, and valuable suggestions and constructive comments on this thesis.

I would like to thank Professor Ralph McKenzie for graciously consenting to serve on my thesis committee and take his time reading this thesis and providing useful comments.

I would like to express my appreciation to Richard Rudell for many stimulating discussions and ideas, for his support and collaboration to many of the experiments, and for his friendship.

Thanks are due to all members of the CAD group for providing the enjoyable working environment. In particular, I would like to thank Srinivas Devadas, Hi-Keung Tony Ma, Kanwar Jit Singh, Alex Saldanha, Sharad Malik, Patrick McGeer, Ellen Santovich, and Abdul Malik for their interests in this research and for many helpful discussions. I would like to thank Wendell Baker and Rick Spickelmier for providing constant support and help in writing and formatting this thesis.

Special thanks are due to my colleagues and friends in industry. In particular, I would like to thank Gary Gannot, Ewald Detjens, Kurt Keuzter, Mauro Pipponzi, Ajit Prabhu for many helpful discussions and ideas, and for providing me with examples from real designs.

Supports from the Defense Advanced Research Projects Agency under contract N00039-87-C-0182, the National Science Foundation under subcontract ECS-8430435, AT&T, Intel Corporation, and SGS Thomson are all grateful acknowledged.

I dedicate this thesis, with love, to my Parents whose unending love and support of all that I do is greatly appreciated and never forgotten.

I am grateful to my Uncle Charles Wang for providing support and help for my education in many occasions and making the completion of this thesis possible.

Finally, I would like to thank my Wife Pearl Wang for her everlasting love, en-

courragement, understanding and patient throughout my academic career.

Contents

Acknowledgements i

Table of Contents iii

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Factoring Logic Functions 4
- 1.2 Boolean Simplification 5
- 1.3 Phase Assignment 6
- 1.4 Timing Optimization 7

2 Basic Definitions 9

3 Factoring Logic Functions 21
- 3.1 Introduction 21
- 3.2 Basic Definition 23
- 3.3 Manipulating Factored Forms 26
  - 3.3.1 List of operations 27
  - 3.3.2 Notation 28
  - 3.3.3 Basic operations 29
  - 3.3.4 Generating kernels 35
  - 3.3.5 Higher level operations 42
- 3.4 Heuristic Factoring Algorithms 46
  - 3.4.1 Generic Factoring Algorithm 48
  - 3.4.2 Quick Factoring 52
  - 3.4.3 Good Factoring 53
  - 3.4.4 Boolean Factoring 54
  - 3.4.5 Complement Factoring 55
  - 3.4.6 Dual Factoring 56
  - 3.4.7 Factoring Incompletely Specified Functions 56
  - 3.4.8 Summary of Factoring Algorithms 57

3.5 Properties of Optimum Factored Form 58
3.5.1 Optimum Factoring Theorems 58
3.5.2 Essential Variables 60
3.5.3 Redundant Variables 65
3.6 Experiments and Results 67
3.7 Future work 70

4 Boolean Simplification **73**
4.1 Introduction 73
4.2 Basic Definitions 74
4.3 Simplification procedure 77
4.3.1 An Example 77
4.3.2 Boolean Substitution 78
4.3.3 Simultaneous Boolean Substitution 80
4.3.4 Output don't-care conditions 82
4.3.5 Simplification 83
4.4 Filtering Don't-care Sets 84
4.4.1 An exact filter 85
4.4.2 Topological interpretation of the exact filter 87
4.4.3 Approximate filters 92
4.4.4 Putting it together 94
4.5 Don't-care Set Representation 94
4.5.1 Reduction using new variables 95
4.5.2 Reduction by complementation 96
4.5.3 Requirement for a new two-level minimizer 97
4.6 Literal Weight 98
4.7 Experiments and Results 99
4.8 Future Work 106

5 Phase Assignment **107**
5.1 Introduction 107
5.2 Basic Definitions 109
5.3 Complexity and Formulation 111
5.3.1 Complexity of Phase Assignment 112
5.3.2 A Mathematical Programming Formulation 116
5.3.3 Exact Solution on Trees 118
5.4 Heuristic Algorithms 122
5.4.1 Computing Inverter Saving 122
5.4.2 Incremental Update of Inverter Saving 124
5.4.3 Quick Phase 126
5.4.4 Good Phase 127
5.4.5 Look-ahead 128
5.5 Generalized Phase Assignment 129
5.5.1 NN-class and Related Definitions 130
5.5.2 Choosing A Move 132

5.5.3 Generalization of Heuristic Algorithms 136
5.6 Experiments and Results 137
5.7 Future Work 142

6 Timing Optimization **145**
6.1 Introduction 145
6.2 Basic Definition 147
6.3 Technology Independent Timing Optimization 148
6.4 Re-structuring Algorithm 149
6.4.1 Identifying critical nodes 151
6.4.2 Computing weights of ϵ-critical nodes 154
6.4.3 Finding minimum weighted cut-sets 156
6.4.4 Re-synthesis 159
6.4.5 Re-structuring algorithm 159
6.5 Timing-Driven Decomposition into Trees 159
6.5.1 Theorems About Optimum Decomposition 160
6.5.2 Timing Decomposition Algorithms 166
6.6 Incremental Delay Trace 168
6.6.1 Graph Theoretical Results 169
6.6.2 Application in Incremental Delay Trace 176
6.7 Future Work 180

7 Summary and Conclusion **181**
7.1 Factoring Logic Functions 181
7.2 Simplifying Logic Functions 182
7.3 Phase Assignment 183
7.4 Technology-Independent Timing Optimization 184

Bibliography **185**

List of Figures

3.1 A logic function (a) and its implementation (b) 22
3.2 Factoring tree of $((a' + b)cd + e)(a + b') + e'$ 24
3.3 Factoring tree of $(ab + a'c + b'c')(d + e)$ 29
3.4 primitive tree manipulations 30
3.5 $AND(f, g)$ 30
3.6 Certain structure of an incompletely specified function 65
3.7 Special structure of the cube matrix of f 66

4.1 Special structure of an incompletely specified function $(F, D + E)$ 86
4.2 A special circuit structure 88
4.3 Several examples of non-minimal $RI(f)$ 91
4.4 Structure of DI-sufficient $RI(f)$ 92
4.5 Structure that disjoint support filter looks for 93
4.6 A case of non-equal literals of f 98

5.1 Equivalent implementations of $a(\overline{b} + c)$ 108
5.2 Reducing inverters by complementing certain functions 108
5.3 Replacing f by its dual d_f 111
5.4 Steps in constructing a DAG from a graph 114
5.5 Partition of $FANOUTS(f)$ into $NFO(f)$ and$PFO(f)$ 117
5.6 Associate constraint variables with nodes and inverters 118
5.7 Greedy algorithm falls into local minimum (d) 119
5.8 Finding $OPA_N(\eta_f)$ and $OPA - P(\eta_f)$ 121
5.9 Computing inverter savings of flipping f 123
5.10 Nodes affected when f is flipped 125
5.11 look-a-head gives better result on this circuit 129
5.12 Three cells in a standard cell library 130
5.13 (b) can be derived from (a) by negating some inputs and output 131
5.14 Replacing (a) by (b) to save two inverters in (c) 133
5.15 Hierarchy of permutable pins and pin groups 134
5.16 Deficiency of $QUICK_PHASE$ and $QUICK_PHASE'$ 140
5.17 Partial hill-climbing of $GOOD_PHASE$ 140
5.18 Quality of *Random_Greedy* vs. *Good_Phase* 141

5.19 CPU time of *Random_Greedy* vs. *Good_Phase* 142
5.20 Quality of *Random_Greedy* vs. *Good_Phase* 143

6.1 Equivalent circuits with different timing property 146
6.2 Reducing delay by collapsing and re-decomposition 150
6.3 Node y is easier to be speed up than x is 154
6.4 Area increase during re-synthesis 155
6.5 To re-synthesis at node x, nodes u, v, and w have to be duplicated. 156
6.6 case 1: p and q are independent 162
6.7 case 2: p depends on q 163
6.8 A structure always exists in D_f 164
6.9 Structure of the tree decomposition of sum of orthogonal cubes 165
6.10 When A_x changes from 1 to 2, only A_y is effected 168
6.11 Inserting a node between a pair of connecting nodes 171
6.12 Effect of $DELTA$ on the number of nodes updated 173
6.13 Two ways of updating an ordering function 176
6.14 If x and y are scheduled before updating the arrival time, z is evaluated only once. Otherwise, z might be evaluated twice. 178

List of Tables

2.1 Kernels and co-kernels of $(a+b+c)(d+e)f+g$ 14

3.1 Summary of factoring algorithms on 153 functions 68
3.2 Comparison of various factoring algorithms 69
3.3 Comparison of factoring algorithms on contrived examples 70
3.4 Comparison of factoring algorithms on contrived examples 70

4.1 Sizes of the entire fanin don't-cares and their alternative representation . . 101
4.2 Sizes of one level fanout don't-cares and their complements 102
4.3 Effect of exact and heuristic filters on the sizes of fanin don't-care set . . . 103
4.4 Effect of exact and heuristic filters on fanout don't-cares 104
4.5 Simplification results using various don't-care sets 105

5.1 Comparison of $QUICK_PHASE$ and $GOOD_PHASE$ 139

Chapter 1

Introduction

Research done over the past thirty years has led to efficient methods for implementing combinational logic in optimal two-level form using programmable logic arrays (PLA's). It is commonly accepted that most of the problems in the area of PLA synthesis, including logic minimization and verification, layout generation, folding, testing, with possible exception of state encoding, are now solved to the extent that efficient programs exist and produce nearly optimal results. More importantly, this advance has been accompanied and supported by a rich collection of theory with effects far beyond its original scope.

However, due to the rapid advancement in integrated circuit (IC) technology, digital systems are becoming increasingly complex. Many logic functions are inappropriate for PLA implementation because of more stringent silicon area and/or speed constraints. For example, there exist functions whose minimum two-level representation has $2^n - 1$ product terms, where n is the number of primary inputs. In addition, even if a two-level representation contains a reasonable number of terms, there are many cases in which a multi-level representation can be implemented in less area and generally as a much faster circuit. Furthermore, new developments in VLSI design style such as gate-arrays, standard-cells, and programmable logic devices, have made multi-level logic an attractive and economical implementation style for its short design circle, reasonable silicon efficiency, and its ability to adapt to new technologies. For these reasons, the multi-level logic implementation style has been widely used in designing digital systems.

As complexity of digital systems increases, so does the complexity of the combinational logic components inside the systems. It is no longer feasible or economical for a designer to optimize the logic manually. Automatical tools for optimizing multi-level

logic have become essential to produce high-quality designs. Another need for automatic multi-level logic optimization tools comes from recent development of tools for architectural design of digital systems. These "high-level" tools synthesize a system from a functional level description and produce logic which, even though functionally correct, may contain many redundances, relying on logic optimization tools to derive efficient implementations. Such tools must be efficient enough to produce results in a reasonable amount of time and yet sophisticated enough to explore the entire design space, make appropriate tradeoffs, and understand different electrical design styles (e.g., domino logic, static CMOS) and layout design styles (e.g., Weinberger arrays, gate matrix, standard cells, and gate arrays).

Optimal multi-level logic synthesis is a known difficult problem which has been studied since the 1950's. However, much work still remains to be done in order to achieve the same level of advancement as for two-level logic synthesis. In recent years, an increasing level of research has been apparent in multi-level logic optimization. The first of the modern developments is the Logic Synthesis System (LSS) [20] at IBM, which has as target technology a variety of gate arrays and standard cells. The Yorktown Silicon Compiler, which automatically synthesizes and lays out CMOS dynamic logic, is based on multi-level logic and has domino CMOS logic as its primary target technology. The SOCRATES system [26] is a multi-level logic synthesis system which uses gate arrays and standard cells, and is one of the earliest to emphasize optimized timing performance. More recent work in multi-level logic optimization includes MIS [14] developed at University of California, Berkeley, and BOLD [7] developed at University of Colorado, Boulder. Both MIS and BOLD are aimed at optimization techniques which are independent of particular technologies, and were developed with the goal of bringing the multi-level logic optimization to the level of science obtained for two-level logic optimization. In particular, MIS was developed as an interactive system with totally open architecture. The objective is to provide not only an efficient optimization tool but, more importantly, an environment in which new algorithms can be implemented quickly and experiments can be performed easily.

A commonly accepted optimization criterion for multi-level logic synthesis is to minimize the area occupied by the logic equations (which is measured as a function of the number of gates, transistors, and nets in the final set of equations) while simultaneously satisfying the timing constraints derived from the performance requirements of the systems. Considerations such as testability should also be included; however, in most current systems, testability is only considered indirectly as a side-effect of a less redundant implementation.

For multi-level logic optimization, two basic methodologies have evolved: 1) *global* optimization, where the logic functions are manipulated into an optimal multi-level form with little consideration of the form of the original description (e.g., the Yorktown Silicon Compiler [11], part of Angel [29], SOCRATES [26], and FDS [23]); 2) *peephole* optimization, where local transformations are applied to a user-specified (or globally-optimized) logic function (e.g., a part of Angel, LSS [20], MAMBO [27], and SOCRATES). Global optimization is generally accomplished in algorithmic fashion, where functions are *eliminated* and *created* according to certain algorithms. The algorithmic-based techniques are generally used to derive a "good" global structure of the multi-level logic and therefore are mostly technology independent. In contrast, peephole optimization is usually based on a finite set of rules, where the rule set and the order in which to apply the rules are dependent upon the particular technology used in the final implementation.

Global structuring algorithms have been proposed in the past (e.g., [2], [35]), but these techniques have required an exhaustive search which is prohibitively expensive for the complexity of the designs of interest today. The problem of global structuring can be viewed as a problem of optimal common divisor extraction, i.e., expressing each output of the logic as a single Boolean function of the inputs (*collapsing*), identifying a set of common divisors among two or more functions, creating intermediate functions for the divisors, and re-expressing each output in terms of the inputs and the intermediate functions such that the total cost of the logic is minimized. The first difficulty comes from the fact that single-equation representation of some functions is too large to be practical. So, the algorithms must be able to cope with an existing multi-level logic directly rather than relying on being able to collapse the logic network first. The second difficulty is due to the fact that the set of possible divisors for a typical function is too large to enumerate and identifying an optimal sub-set of them is even harder. In practice, an iterative improvement scheme is used to reduce the size of the logic incrementally. The method iterates over two major steps: selectively eliminating some intermediate functions while keeping the size of the logic under control; extract common divisors of existing functions. This approach was inspired by the algorithms for two-level minimization, first experimented in YLE [10], and later enhanced in MIS.

The success of PLA minimization is largely due to its abstraction as a two-level logic minimization in which problems can be well-defined and made independent of the implementation technology. One of the most attractive feature of multi-level logic is its

ability to adapt to different technologies and implementation styles. But, it is this feature that makes multi-level logic optimization less well-defined. Recent research emphasis has been placed on abstracting the multi-level logic implementation problem and providing a simple mathematical model, the Boolean algebra, in which theorems can be proved and algorithms can be designed. Two general concepts are used to aid the abstraction of various implementation problems:

1. the concept of logic function in a factored form representing logic gates. For example, NAND gate is represented by the function $O = (ab)'$ and $AOI223$ gate is represented by the function $O = (ab + cd + efg)'$. Using this representation, logic gates can be combined, separated, eliminated, or created by directly manipulating the logic functions using rules of Boolean algebra.

2. the concept of Boolean network representing a combinational logic block, which is simply a set of inter-dependent set of logic functions.

All problems solved in this thesis came from real design requirements and are presented, using this abstraction, as well-defined combinatorial optimization problems.

This thesis addresses four individual problems in multi-level logic optimization: factoring logic functions, simplifying logic functions, minimizing the total number of inverters, and improving circuit performance.

1.1 Factoring Logic Functions

There are many ways of representing logic functions; among them are the binary decision diagram [19], the truth table, the disjunctive form (SOP), the spectral form [39], and the factored form. A factored form is a parenthesized logic expression (e.g., a+b(c+d)). There are many attractive properties of factored forms when compared with other representations. Perhaps the most important feature of factored forms is their association with final implementations of the functions they represent. In most design styles (for example, complex-gate CMOS design) the implementation of a function corresponds directly to its factored form. For example, the factored form $a + b(c + d)$ specifies not only the number of transistor pairs in the gate realizing the function, but also the way in which the transistors are connected. Hence, factored forms are used to derive the physical implementation of

logic gates in the static-CMOS complex gate design style, and are useful in predicting area and delay of a circuit in a technology independent fashion.

Another feature of factored forms is the compactness of the representation. Factored forms are more compact than disjunctive forms. For example, the factored form $(a+b)(c+d)$ when expressed as a sum-of-products form would be $ac+bc+ad+bd$. Furthermore, many commonly used logic operations can be easily performed on factored forms. By duality, every factored form of a function gives a factored form of its complement, which is a property not found in the sum-of-product form. These properties make factored forms useful as an internal representation of logic functions in a logic synthesis and optimization system.

Chapter 3 of this thesis is intended to achieve three goals. The first is to define systemically a set of logic operations on factored forms. Efficient algorithms will be given and their complexities will be analyzed. The results can be used to implement a new logic synthesis system in which the factored forms are used as the internal representation of logic functions. The second goal is to develop and analyze algorithms for deriving good factored forms from sum-of-product forms. A spectrum of algorithms will be given in order to achieve desired performance versus quality tradeoffs. Last but not least, several properties of optimum factored forms are studied, from which more powerful algorithms may be derived and lower bounds on the size of factored forms can be computed to verify the quality of factoring algorithms.

1.2 Boolean Simplification

A Boolean network is simply a collection of interconnected logic functions. Each one can be represented as a sum-of-products expression or a factored expression. The sum-of-products form of each function in a Boolean network can be minimized, i.e. replaced by an equivalent, but simpler, sum-of-products form using two-level logic minimization algorithms. Two-level minimization can be made more powerful in this context by providing the minimizer with various don't-care sets derived from the immediate environment of a function. Just how much of the don't-care sets to derive depends on how thorough and how fast this minimization process is expected to be.

The ultimate goal of simplifying a function in a Boolean network is to replace it with another equivalent, but minimal, function which has the least literals in its factored

form. At the moment, there is no algorithm for minimizing a function with this objective. Instead, two-level minimization algorithms are used to minimize the number of literals in the sum-of-products form of a function as an approximation to the number of literals in the factored form.

The don't-care set of a function in multi-level logic can and do become very large in the sum-of-products representation. Furthermore, the dimension of the Boolean space in which a function and its don't-care set are defined can also become very large. Together, they limit seriously the effectiveness of our simplification procedure. Chapter 4 looks further into ways of reducing the size of a don't-care set by finding and removing the useless or "almost" useless cubes from the don't-care set. Doing so not only reduces the size of a don't-care set (in the sense of Boolean space as well as the representation), but also restricts the dimension of the Boolean space. Experiments performed in MIS have shown that this approach improves significantly the performance of the simplification process at the potential expense of a small loss in quality of results.

1.3 Phase Assignment

The principle of duality is an important property of Boolean functions. It simply states that every truth statement involving operator + and * and constant 0 and 1 remains true when both the operators and the constants are switched. The duality principle makes it possible to implement a Boolean function in either un-complemented or complemented form provided that necessary inverters are supplied at the inputs as well as the output of the function. In a given multi-level combinational logic network, each function can be implemented in its present or complemented form with appropriate inverters. However, the costs of implementing a function or its complement may differ. For example in static CMOS technology, NAND gates are generally preferred over NOR gates due to performance considerations. More importantly, the choice of implementing a function in its present or complemented form affects the number of inverters needed at the inputs and output of the function, which may in turn affect how other functions are to be implemented. The phase-assignment problem is to determine for each function in a circuit whether to complement it such that the total cost of the circuit is minimized.

To solve the phase assignment problem, the first question to be answered is whether an algorithm exists for solving the problem optimally in a reasonable amount of time. It

will be shown in Chapter 5 that the problem is NP-complete. Only when the network is in a very specific form can the phase assignment problem be solved optimally in polynomial time. A dynamic programming algorithm will be designed to minimize the total number of inverters in a network whose topology is a tree.

To offer a practical solution to the phase assignment problem, several heuristic algorithms have been developed, tested on a large set of examples, and experimentally shown to be very efficient and effective.

When used in a cell-based design style, e.g., standard cell or gate array technology, the phase-assignment problem can be extended to allow more general modifications to functions than just complementations. New modifications will be formally defined which extends the heuristic algorithms to solve this *generalized phase assignment.*

1.4 Timing Optimization

Being able to meet performance requirements is essential in synthesizing digital logic circuits. As the circuit complexity increases, many of the manual methods for performance improvement have become impractical. Automatic performance optimization of digital circuits has played and will be playing a more and more important role in any synthesis system. Such performance optimization system must be able to work with different levels of circuit hierarchy and at various steps of the design process (e.g. re-timing, reducing delay in combinational logic, delay-driven layout, etc.). Chapter 6 deals exclusively with performance optimization of combinational logic circuits (timing optimization). The results can be used as a component in a performance driven synthesis system.

Timing optimization of combinational circuits can be viewed as a three-phase process. In the first phase, circuits are globally restructured to have better "timing properties". There, the quality of the circuits is judged not by the detailed timing figures, but rather, by their structure. An example of global restructuring is the conversion from a carry-ripple adder to a carry-look-ahead adder. This phase is characterized by its independence of target technology. The objective here is to look for global structural changes of circuits to achieve delay reductions that can not be obtained by lower level techniques such as transistor sizing or buffering.

The second phase of timing optimization is performed during the physical design process. Here, the target technology is known and more accurate timing information are

available. Optimization involves transistor sizing, buffering, delay-driven placement, etc. This phase is characterized by its dependence on a particular target technology and on the existence of fast and relatively accurate timing simulators.

The third and last phase of timing optimization is performed when actual designs are available. There, much more accurate timing analyzers are used to fine-tune the circuit parameters. This phase serves both the optimization and verification purpose.

There have been several previous attempts to solve this problem. SOCRATES [26] uses a rule based approach and tries to achieve global restructuring through a sequence of local transformations. Even though the system is very flexible in adapting to various cell libraries and target technology, it is heavily dependent on the rule set and the order in which the rules are applied. More recently, an algorithmic-based restructuring technique was tested in the Yorktown Silicon Compiler [10]. Even though the work lacks detailed algorithms and theoretical analysis, it had several interesting ideas from which some ideas presented in this chapter were originated.

Chapter 6 is dedicated to the first phase of timing optimization, technology-independent timing optimization. Detailed description of algorithms will be given. Emphasis will be placed on an important step in the optimization process, timing-driven decomposition. Exact conditions will be given under which optimal solutions can be obtained. Last, a technique for improving the optimization time, *incremental delay trace*, will be presented.

Chapter 2

Basic Definitions

Boolean Algebra

Boolean algebra is an algebraic structure defined on a set of elements B with two binary operators $+$ and $\star$ (called *disjunction* and *conjunction*) provided the following (Huntington) postulates are satisfied:

1. B is closed under the operator $+$ and $\star$, i.e.

$$x, y \in B \Rightarrow x + y \in B \textit{ and } x \star y \in B.$$

2. There exists an identity element with respect to $+$ designated by 0 and an identity element with respect to $\star$ designated by 1, i.e.

$$\exists 0 \in B(\forall x \in B(x + 0 = x))$$
$$\exists 1 \in B(\forall x \in B(x \star 1 = x))$$

3. Both $+$ and $\star$ are commutative, i.e. for all $x, y \in B$ the following equalities hold:

$$x + y = y + x$$
$$x \star y = y \star x$$

4. $+$ is distributive over $\star$ and $\star$ is distributive over $+$, i.e. for all $x, y, z \in B$ the following equalities hold:

$$x \star (y + z) = (x \star y) + (x \star z)$$
$$x + (y \star z) = (x + y) \star (x + z)$$

5. For every element $x \in B$, there exists an element $x' \in B$ (called the *complement* of x) such that

$$\begin{aligned} x + x' &= 1 \\ x \star x' &= 0 \end{aligned}$$

6. B has at least two elements.

A *two-valued Boolean algebra* is a Boolean algebra in which the set B has exactly two elements, $B = \{0, 1\}$. In this thesis, Boolean algebra refers exclusively to the two-valued Boolean algebra.

Duality and De Morgan's Law

An important property of Boolean algebra is the *duality principle*. It states that for every truth (false) statement $S(+, \star, 0, 1)$ the corresponding statement in which the operators and identity elements are interchanged, i.e. $S(\star, +, 1, 0)$, is also true (false). The duality principle is often used as the following specific forms of De Morgan's Law:

$$(x + y)' = x'y'$$
$$(xy)' = x' + y'$$

Single-output Boolean Function

A *binary variable* can take the value of 0 or 1, representing one dimension of the Boolean space $B = \{0, 1\}$. A *single-output Boolean function* F maps an n-dimensional Boolean space to a one dimensional Boolean space, i.e.

$$F : \{0, 1\}^n \rightarrow \{0, 1\}.$$

A single-output Boolean function is also called *logic function* or simply *function* when there is no confusion. There are two special functions denoted by 0 and 1. Function 0 maps $\{0, 1\}^n$ to the constant 0 and function 1 maps $\{0, 1\}^n$ to the constant 1.

Sum-Of-Products Form

A *literal* is a variable or its complement (e.g., x or x').

A *cube* is a set C of literals such that $x \in C$ implies $x' \notin C$ (e.g., $\{x, y, z'\}$ is a cube, and $\{x, x'\}$ is not a cube). A cube represents the conjunction of its literals. The

trivial cubes, written as 0 and 1, represent the Boolean functions 0 and 1 respectively. By definition, function 0 is represented by the empty set ϕ and 1 is represented by a set containing only the empty set, $\{\phi\}$. Since a cube is a set, cube-containment is naturally defined by the set-containment.

A sum-of-products (SOP) form (also called *expression*) is a set of cubes. For example, $\{\{x\},\{y,z'\}\}$ is an expression consisting of two cubes $\{x\}$ and $\{y,z'\}$. An expression represents the disjunction of its cubes.

Conventionally, a cube is written as a list of its literals omitting the operator $\star$ (e.g., cube $\{x,y',z\}$ is written as $xy'z$ rather than $x\star y'\star z$). An expression is written as the sum of its cubes with implicit parenthesis around each cube, e.g., expression $\{\{x,z'\},\{y',z\}\}$ is written as $xz'+y'z$ instead of $(x\star z')+(y'\star z)$.

An expression is *algebraic* if no cube in the expression contains another. For example, expression $a+bc$ is algebraic, and expression $a+ab$ is non-algebraic because $\{a\}\subseteq\{a,b\}$.

The *support* of an expression f is a set of variables, $sup(f)$, f depends on, i.e.

$$sup(f)=\{x|\exists C\in f \textit{ such that } x\in C \textit{ or } x'\in C\}.$$

For example, $sup(xy+x'z')=\{x,y,z\}$.

Two expressions f and g are said to be *orthogonal*, $f\perp g$, if they have *disjoint support*, i.e., $sup(f)\cap sup(g)=\phi$.

Factored Form

An alternative representation (to sum-of-products form) of a logic function is the *factored form*. It is the generalization of sum-of-products form allowing nested parenthesis. For example, the expression

$$ace+ade+bce+bde+e'$$

can be written in factored form as

$$e(a+b)(c+d)+e'.$$

In other words, a factored form is a sum of products of sums of products,..., of arbitrary depth. Since the formal definition of factored forms is used exclusively in Chapter 3, the definition will be given in the chapter along with some of properties.

Division

Since Boolean algebra does not have additive or multiplicative inverses, there can be no division operation. However, one can define operations which, when given functions f and p, find functions q and r such that $f = pq + r$. Every such operation is like the division operation and is therefore called *division* of f by p generating *quotient* q and *remainder* r. It is clear that such a division operation is not unique. Even for a given division operation, the resulting q and r may be dependent upon the particular representation of f and p. Next, two classes of divisions are introduced and examples will be given for each one.

Algebraic and Boolean Division

The *product* of two cubes c and d is a cube defined by (recall that a cube is a set of literals)

$$cd = \begin{cases} 0 & if\ \exists x\ (x \in c \cup d\ and\ x' \in c \cup d) \\ c \cup d & otherwise \end{cases}$$

The *product* of two expressions f and g is a set defined by

$$fg = \{cd | c \in f\ and\ d \in g\ and\ cd \neq 0\}.$$

Notice that $cd = 0$ if and only if $c \cup d$ contains both a literal and its complement.

fg is an *algebraic product* if f and g are orthogonal (have disjoint support); otherwise fg is a *Boolean product.* For example, $(a+b)(c+d) = ac + ad + bc + bd$ is an algebraic product and both $(a+b)(a+c) = a + ab + ac + bc$ and $(a+b)(b'+c) = ab' + ac + bc$ are Boolean products.

An operation OP is called *division* if, given two functions f and p, it generates q and r ($< q, r >= OP(f,p)$) such that $f = pq + r$. If pq is an algebraic product, OP is called an *algebraic division*; otherwise pq is a Boolean product and OP is therefore called a *Boolean division.* An algebraic division is usually denoted by $ALGE_DIV$ and a Boolean division is usually denoted by $BOOL_DIV$.

Weak and Strong Division

Weak division and *strong division* are specific examples of algebraic division and Boolean division.

Given two algebraic expressions f and p, a division is called *weak division* if 1) it generates q and r such that pq is an algebraic product, 2) r has as few cubes as possible, and 3) $pq + r$ and f are the same expression (having the same set of cubes). Given the expressions f and p, it can be shown that q and r generated by weak division are unique. *WEAK_DIV* denotes the operation of weak division. Often, f/p is used to denote the quotient of weak dividing f by p. [17] gives an $O(n \log n)$ algorithm for weak division. A linear algorithm for weak division exists given that expression f and p are represented in a special way [38].

Unlike weak division, strong division finds the quotient q and the remainder r by the following procedure:

STRONG_DIV(f,p)

1. Let x be a literal representing expression p.
2. $D = x'p + xp'$.
3. $g = MINIMIZE(f, D, x)$
4. $q = \{c - \{x\} | c \in g \text{ and } x \in c\}$.
5. $r = \{d | d \in g \text{ and } x \notin d\}$.
6. return (q, r).

In *STRONG_DIV*, x is a special variable representing expression p. D is the exclusive-OR of x and p. For example, if $p = a + b$, then $D = x'a + x'b + xa'b'$. Since x is always equal to p, function D is always false. So, D can be used as the don't-care set to reduce the number of cubes in f, which is done by $MINIMIZE$. Notice that the literal x' may appear in the minimized f. The objective of *STRONG_DIV* is to find a quotient and a remainder which are as simple as possible. If x' is allowed, then the size of the reaminder cannot be controlled. To prevent this, one has to modify the $MINIMIZE$ to avoid using the litral x'. Finally, all the cubes g containing literal x form the quotient and the remaining cubes form the remainder. [14] provides comprehensive discussions of the strong division. Usually, $f \div p$ is used to denote the quotient of strong divide f by p. Strong division is a Boolean division because the quotient q and the divisor p are not guaranteed to be disjoint.

Note also that unlike weak division, q and r are not unique and in general depends on the initial representation of f and p.

Kernel

The notion of a kernel of an algebraic expression was introduced in [17] to provide means for finding subexpressions common to two or more expressions. All operations used to find kernels are algebraic (i.e., algebraic product, algebraic division, etc.), but the word *algebraic* is omitted for brevity. In particular, algebraic division is done by $WEAK_DIV$.

An expression is *cube-free* if no cube divides the expression evenly (e.g., $ab + c$ is cube-free; $ab + ac$ and abc are not cube-free). Notice that a cube-free expression must have more than one cube.

The primary divisors of an expression f are the set of expressions

$$[D(f) = \{f/c | c \text{ is a cube } \}.$$

The kernels of an expression f are the set of expressions

$$K(f) = \{g | g \in D(f) \text{ and } g \text{ is cubefree } \}.$$

In other words, the kernels of an expression f are the cube-free primary divisors of f.

A cube c used to obtain the kernel $k = f/c$ is called a *co-kernel* of k, and $C(f)$ is used to denote the set of co-kernels of f. For example, the kernels and their corresponding co-kernels of the function

$$\begin{aligned} x &= adf + aef + bdf + bef + cdf + cef + g \\ &= (a + b + c)(d + e)f + g \end{aligned}$$

are listed in Table 2.1. Notice that a kernel may have more than one co-kernel and a

kernel	co-kernel
$a + b + c$	$df,\ ef$
$d + e$	$af,\ bf,\ cf$
$(a + b + c)(d + e)f + g$	1

Table 2.1: Kernels and co-kernels of $(a + b + c)(d + e)f + g$

co-kernel can be the trivial cube 1 if the original expression was cube-free.

For certain operations described in the following chapters, it is nearly as effective and frequently more efficient to compute a certain subset of $K(f)$ rather than the full set. This leads to the following recursive definition for the *level* of a kernel. Let

$$K^n(f) = \begin{cases} \{k \in K(f) | K(k) = \{k\}\} & n = 0 \\ \{k \in K(f) | \forall k_1 \in K(k)\ k_1 \neq k \textit{ such that } k_1 \in K^{n-1}(f)\} & n > 0 \end{cases}$$

If $k \in K^0(f)$, then k is a level-0 kernel of f. If $k \in K^n(f)$ and $k \notin K^{n-1}(f)$, then k is a level-n kernel of f. According to the definition, a kernel is said to be of level-0 if it has no kernels except itself. Similarly, a kernel is of level n if it has at least one level n-1 kernel but no kernels (except itself) of level n or greater. This gives us a natural partition of the kernels:

$$K^0(f) \subset K^1(f) \subset K^2(f) \subset \cdots \subset K^n(f) \subset K(f).$$

Notice that the set of level-i kernel is given by $K^{i+1}(f) - K^i(f)$.

Multiple-output Boolean Function

A *multiple-output Boolean function F* maps an n-dimensional Boolean space to an m-dimensional Boolean space, i.e.,

$$F : \{0,1\}^n \rightarrow \{0,1\}^m.$$

A multiple-output Boolean function F can also be defined (equivalently) as a set of single-output Boolean functions, i.e.,

$$F = \{F_i | F_i : \{0,1\}^n \rightarrow \{0,1\}\}.$$

Multi-level Logic

In many applications, it is infeasible to describe each single-output function of a multiple-output function as a single expression or a single factored form. Often, a set of intermediate functions are introduced, each one depends on the original inputs and possibly other intermediate functions. Then, each single-output function can be expressed as a function of original inputs as well as the intermediate functions. For example, the multiple-output function

$$\begin{aligned} F_1 &= ((a+b)c+d)e+f \\ F_2 &= ((a+b)c+d)g+h \end{aligned}$$

can be expressed as the following set of functions involving intermediate variables x and y:

$$\begin{aligned} F_1 &= ye + f \\ F_2 &= yg + h \\ y &= xc + d \\ x &= a + b. \end{aligned}$$

Multi-level logic refers to any multiple-output Boolean function represented by a set of interconnected functions. Therefore, multi-level logic is a particular representation of multiple-output functions. If F is a multiple-output Boolean function, $\eta(F)$ is often used to denote a multi-level logic representation of F. All the original inputs of F are called *primary inputs* of $\eta(F)$, denoted by the set $PI(\eta(F))$. All the outputs of F are called *primary outputs* of $\eta(F)$, are denoted by the set $PO(\eta(F))$. All other variables in $\eta(F)$ are called *intermediate variables.* The set of functions of $\eta(F)$ are denoted by $FUNCTION(\eta(F))$.

Boolean Network

A multi-level logic function can be graphically represented as a directed acyclic graph (DAG) where each node i is associated with 1) a variable y_i and 2) a representation f_i of a logic function (sum-of-products form and/or factored form).

In the graph, an arc connects node i to node j if $y_i \in sup(f_j)$. The primary inputs and primary outputs correspond to special nodes in the graph, the *source* nodes and *sink* nodes of the DAG. There is no function associated with the source or sink nodes.

A node j is a *fan-in* of a node i if function f_i depends on variable y_j explicitly, i.e., there is an arc from j to i. The set of all fan-in's of a node i is denoted by

$$FI(i) = \begin{cases} \phi & i \text{ is a primary input} \\ \{j | j \in sup(i)\} & \text{otherwise} \end{cases}$$

The set of transitive fan-in's of a node i is denoted by $TFI(i)$ and is defined recursively as:

$$TFI(i) = \begin{cases} \phi & i \text{ is a primary input} \\ FI(i) \cup \bigcup_{j \in FI(i)} TFI(j) & \text{otherwise} \end{cases}$$

Likewise, a node j is a *fan-out* of node i if function f_j depends explicitly on variable y_i, i.e., there is an arc from i to j. The set of all fan-out's of a node i is denoted by

$$FO(i) = \begin{cases} \phi & i \text{ is a primary output} \\ \{j | i \in sup(j)\} & \text{otherwise} \end{cases}$$

The set of transitive fan-out's of a node i is denoted by $TFO(i)$ and is recursively defined as:

$$TFO(i) = \begin{cases} \phi & i \text{ is a primary output} \\ FO(i) \cup \bigcup_{j \in FO(i)} TFO(j) & \text{otherwise} \end{cases}$$

The set of all nodes in a Boolean network η is denoted by $NODE(\eta)$. Since every variable in a Boolean network corresponds to a signal in the implementation, variables are sometimes called *signals* and the set of signals of Boolean network is denoted by $SIGNAL(\eta)$. There is a one-to-one correspondence between functions in $FUNCTION(\eta)$, nodes in $NODE(\eta)$, and signals in $SIGNAL(\eta)$. Often, they are used interchangeably.

Incompletely Specified Function

An n-dimensional Boolean function partitions the input space (the domain) into two sets, the ON-set (points which are mapped to 1) and OFF-set (points which are mapped to 0). A Boolean function is *incompletely specified* if it partitions the input space into three sets, the ON-set, OFF-set, and don't-care set (points which can be mapped to either 0 or 1). An incompletely specified function can be represented as a three-element ordered set (f, d, r) in which the first element f represents the ON-set, the second element d represents the don't-care set, and the third element r represents the OFF-set. Together, f, d, and r form a partition of the entire Boolean space, i.e., every point belongs to one and only one of f, d, and r.

Sometimes, an incompletely specified function is represented by two sets (f, d) in which the first element is the ON-set, the second element is the don't-care set, and the OFF-set contains the rest of points and can be obtained from f and d.

Sometimes, the ON-set f as represented by a designer may overlap with the don't-care set d. In this case the real ON-set is implicitly understood as $f - d$.

Extraction

The operation *extraction* on a Boolean network is the process of creating some intermediate functions and variables and re-expressing the original functions as functions

of the original as well as the intermediate variables. For example, the process of translating

$$\begin{aligned} F &= (a+b)cd+e \\ G &= (a+b)e' \\ H &= cde \end{aligned}$$

to

$$\begin{aligned} F &= XY+e \\ G &= Xe' \\ H &= Ye \\ X &= a+b \\ Y &= cd \end{aligned}$$

is extraction. The associated optimization problem is to find a set of intermediate functions such that the resulting network is minimum, measured by either area or delay.

Decomposition

Decomposition of a function is the process of re-expressing a single function as a collection of new functions. For example, the process of translating

$$F = abc + abd + a'c'd' + b'c'd'$$

to

$$\begin{aligned} F &= XY + X'Y' \\ X &= ab \\ Y &= c+d \end{aligned}$$

is decomposition. The associated optimization problem is to find a decomposition with minimum total area or delay.

Factoring

Factoring is the process of deriving a factored form from a sum-of-products form of a function. For example

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

can be factored to

$$F = (a+b)(c+d) + e.$$

The associated optimization problem is to find a factored form with the minimum number of literals.

Substitution

Substitution of a function G into F is to re-express F as function of its original inputs and G. For example, substituting

$$G = a + b$$

into

$$F = a + bc$$

produces

$$F = G(a + c).$$

Collapsing

Collapsing is the inverse operation of substitution. If G is a fanin of F, collapsing G into F re-expresses F without G (un-does the operation of substituting G into F). For example, if

$$\begin{aligned} F &= Ga + G'b \\ G &= c + d \end{aligned}$$

then, collapsing G into F results in

$$\begin{aligned} F &= ac + ad + bc'd' \\ G &= c + d \end{aligned}$$

Chapter 3

Factoring Logic Functions

3.1 Introduction

There are many ways of representing logic functions; among them are the binary decision diagram [19], the truth table, the disjunctive form (SOP), the spectral form [39], and, of course, the factored form. A factored form is a parenthesized algebraic expression. For example, each of the following is a factored form:

$$\begin{aligned}
&a,\\
&a',\\
&ab'c,\\
&ab + c'd,\\
&(a+b)(c+a'+de)+f,
\end{aligned}$$

where a, a', b', c, ... are called literals.

Factored forms have many attractive properties. In most design styles (for example, complex-gate CMOS design) the implementation of a function corresponds directly to its factored form, as shown in Figure 3.1. As illustrated in the example, the factored form specifies not only the number of transistor pairs in the gate, but also the way in which they are connected. Hence, factored forms are useful in estimating area and delay in a multi-level logic synthesis and optimization system.

Factored forms in general are more compact than disjunctive forms. For example, if the following factored form were to be expressed as a sum-of-product form, it would have

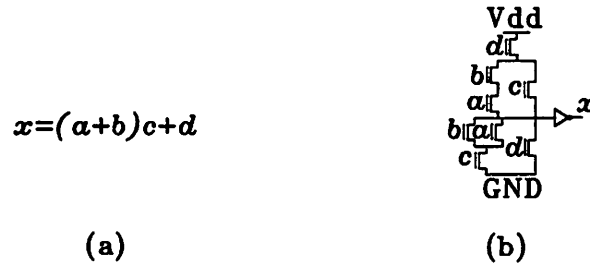


Figure 3.1: A logic function (a) and its implementation (b)

contained 16 product terms with 4 literals in each term.

$$(a+b)(c+d)(e+f)(g+h)$$

Furthermore, many commonly used logic operations can be easily performed on factored forms. By duality, factored forms also provide both functions and their complements, which is a property not found in sum-of-product forms. These properties make the factored forms useful for being used as an internal representation of logic functions in logic synthesis and optimization systems.

This chapter has three goals. The first is to define systemically a set of logic operations on factored forms and to derive efficient algorithms for performing those operations. The results can be used to implement a new logic synthesis system in which the factored forms are used as the internal representation of logic functions. The second goal is to develop and analyze algorithms for deriving good factored forms from sum-of-product forms. Last but not least, several properties of optimum factored forms are studied, from which more powerful algorithms may be derived and lower bounds on the size of factored forms can be computed to verify the quality of factoring algorithms.

Section 2 defines the factored forms and related terminology. It also provides notations for describing algorithms and procedures in the remaining sections. Section 3 defines certain logic operations to be performed directly on factored forms and derives, whenever possible, algorithms to implement the operations. Section 4 develops and analyzes a heuristic factoring algorithm. Several variations of the algorithm will also be presented and compared. Section 5 presents several properties of optimum factored forms of completely specified, as well as incompletely specified logic functions. These properties suggest certain algorithmic structures to factorize logic functions optimally. In several cases, these properties can be used to derive better lower bounds on the size of optimum factored forms.

Section 6 describes how the examples were obtained for testing the factoring algorithms and how the experiments are performed. Results are given to show the qualities of various algorithms. The tradeoffs between run time and quality of results among several algorithms are also presented.

3.2 Basic Definition

A factored form is a parenthesized algebraic expression. Any logic function can be represented by a factored form and any factored form is a representation of some logic function. To be precise, a factored form can be defined recursively by the following rules:

1. A *product* is either a single literal or a product of factored forms.
2. A *sum* is either a single literal or a sum of factored forms.
3. A factored form is either a product or a sum.

For example, each of the following is a factored form:

$$\begin{aligned}&x,\\&y',\\&abc',\\&a+b'c,\\&((a'+b)cd+e)(a+b')+e',\end{aligned}$$

where the first two are literals, the third is a product, the forth is a sum, and the last one is a sum of products of sums of .... By the definition, the following is not a factored form:

$$(a+b)'c$$

because it inverts $(a+b)$ internally which is not allowed by the definition. Like the disjunctive forms, factored forms are in general not unique, as illustrated by the following three equivalent factored forms:

$$\begin{aligned}&ab+c(a+b),\\&bc+a(b+c),\\&ac+b(a+c).\end{aligned}$$

If f is a logic function, then F^f is used to denote a factored form of f, and S^f is used to denote a disjunctive form of f. When there is no confusion, f is used to denote both a logic function and a factored form of the logic.

Factored forms can also be graphically represented as labeled trees, called factoring trees, in which each internal node including the root has a label of either "+" or "*", and each leaf has a label of either a variable or its complement. For example, Figure 3.2 is the factoring tree of $((a' + b)cd + e)(a + b') + e'$,

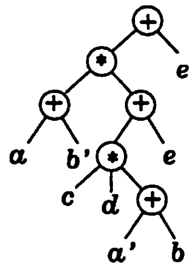


Figure 3.2: Factoring tree of $((a' + b)cd + e)(a + b') + e'$

Any sub-tree of a factoring tree is a *factor* of the factored form represented by the factoring tree. In other word, *factor* of a factored form is any sum term or product term in the factored form. For example, $a(b + c)$ and $de + f$ are factors of the factored form $a(b + c) + de + f$.

Two factored forms are said to be *equivalent* if the represent the same logic function, *f-equivalent* if their factoring trees, including the leaves, are isomorphic. For example, $a(b + c) + bc$ and $ab + c(a + b)$ are equivalent, and $(a + b)(c + d)e$ and $(c + d)e(a + b)$ are f-equivalent.

Some factored forms can be further factored. For example, $ab + ac$ can be further factored to $a(b + c)$, and $(a + b)c + (a + b)d$ can be further factored to $(a + b)(c + d)$. This leads to the following definition of *maximally* factored forms. A factored form is *maximally* factored if

1. For every sum of products, there are no two f-equivalent factors in the products.

2. For every product of sums, there are no two f-equivalent factors in the sums.

For example, the following factored forms are not maximally factored

$$ab + ac,$$
$$(a + b)(a + c),$$

because they can be further factored to

$$a(b + c),$$
$$a + bc.$$

Notice that in Boolean algebra, "+" is distributive over "*" and "*" is distributive over "+". The above factorings are obtained by direct application of the distributive law. The transformation of $ab+ac$ to $a(b+c)$ may seems to be easier than $(a+b)(c+d)$ to $a+bc$. This is because we often think a Boolean expression as a polynomial, in which multiplication is distributive over summation. But, in Boolean algebra, the OR, "+", is also distributive over the AND, "*". So, the second transformation should be just as simple.

The size of a factored form f is measured by the number of literals in the factored form and is denoted by $\rho(f)$. For example, $\rho((a+b)ca') = 4$ and $\rho((a+b+cd)(a'+b')) = 6$. A factored form is said to be *optimum* if no other equivalent factored form has less literals.

A factored form is *positive unate* in x if literal x appears in f and literal x' does not appear in f, *negative unate* in x if literal x' appears in f and literal x does not appear in f. f is *unate* in x if it is either positive unate or negative unate in x. f is *binate* in x if it is not unate in x. For example, $(a + b')c + a'$ is positive unate in c, negative unate in b and binate in a.

The cofactor of a factored form F with respect to a literal l (i.e. x or $\bar{x}$) is a factored form, denoted by F_l, obtained by replacing all occurrences of l with 1 and all occurrences of $\bar{l}$ with 0 and simplifying the factored form using the following identities of the Boolean algebra:

$$\begin{aligned} 1x &= x \\ 1 + x &= 1 \\ 0x &= 0 \\ 0 + x &= x \end{aligned}$$

Notice that after the constant propagation, part of the factored form may appear as $x + x$. In general, x is not a literal, but another factored form. In factor, the two x's may have different factored forms. Identifying these equivalent factored forms and obtaining the

simplification $x + x = x$ is the subject of later sections. Here, we are only interested in obtaining a factored form which represents the cofactor of a function with respect to a literal.

The cofactor of a factored form F with respect to a cube c is a factored form, denoted by F_c, obtained by successively cofactoring F with each literal in c.

EXAMPLE: Let $F = (x + \bar{y} + z)(\bar{x}u + \bar{z}\bar{y}(v + \bar{u}))$ and $c = v\bar{z}$. Then

$$F_{\bar{z}} = (x + \bar{y})(\bar{x}u + \bar{y}(v + \bar{u}))$$
$$F_c = (x + \bar{y})(\bar{x}u + \bar{y})$$

3.3 Manipulating Factored Forms

This section deals with direct manipulations of logic functions in their factored forms. Factored forms are more compact representations of logic functions than the traditionally used sum-of-products forms. For example, the following factored form

$$(a + b)(c + d(e + f(j + i + h + g)))$$

when represented as a sum-of-products form would be

$$ac + ade + adfg + adfh + adfi + adfj$$
$$+bc + bde + bdfg + bdfh + bdfi + bdfj.$$

In fact, every sum-of-products form can be viewed as a special factored form. So, factored form representations are intrinsically smaller than their corresponding sum-of-products forms, and in many cases much smaller as illustrated by the previous example. When measured in terms of number of inputs, there are functions whose size is exponential in sum-of-products forms, but polynomial in factored forms. The following Achilles' heel function is such an example.

$$\prod_{i=1}^{i=n/2} (x_{2i-1} + x_{2i}).$$

It is easy to see that there are n literals in the factored form and $2^{n/2}$ literals in the sum-of-products form.

In most design styles, e.g. complex-gate CMOS design, the implementation of a function corresponds directly to its factored form. Consequently, the number of literals in the factored form of a logic function is generally used as a technology independent estimate of the cost. Traditionally, sum-of-products forms are used as the internal representation of logic functions in most multi-level logic optimization systems because the algorithms and procedures for manipulating sum-of-products forms are readily available. But this approach has two undesirable consequences. First of all, the performance of such systems is unpredictable, for they may accidentally generate a function whose sum-of-products form is too large. Secondly, factoring algorithms have to be used constantly throughout the optimization process in order to provide continuously an estimate for the size of the Boolean network. The CPU resource spent on the factoring becomes significant when accumulated over time.

An obvious solution is to avoid sum-of-products forms by using factored forms as the internal representation of logic optimization systems. But this is not practical unless we know how to perform logic operations on the factored forms directly without converting them to sum-of-products forms. The goal of this section is to compile a set of logic operations commonly found in a logic optimization system and provide, whenever possible, algorithms for performing the operations.

3.3.1 List of operations

The following is the list of logic operations to be considered in this section. They are the basic operations frequently used in the multi-level logic optimization system MIS [14]. The list is by no means complete, but sufficient for implementing most optimization algorithms in MIS. Some of the more complicated operations are also included even though they can be expressed as compositions of simpler operations, because when operated on factored forms these operations are particularly simple to be implemented directly without using the basic operations.

- Basic logic operations.
  - Conjunction (logical AND).
  - Disjunction (logical OR).
  - Inversion.

  - Cofactoring.
  - Simulation.
- Operations used to generate kernels.
  - Dividing a factored form by a literal.
  - Dividing a factored form by a cube.
  - Finding the largest common cube of a factored form.
  - Generating kernels.
- Higher level operations.
  - Dividing a factored form by another factored form.
  - Collapsing two factored forms.
  - Decomposing a factored form.
  - Substituting a factored form into another factored form.
  - Tautology.
  - Simplifying factored forms.

3.3.2 Notation

A factored form can be represented as a labeled tree in which each node has a label of either "*", "+", or LEAF[1]. Each node also has a set of fanin nodes and one fanout node. A factored form can be specified by its factoring tree or simply the root node of the tree. Since every node in a factoring tree is a root of some sub-tree, it can also be viewed as a factored form.

In describing algorithms, variable $f, g, \ldots, x$ are used to denote factored forms, more precisely the root nodes of their corresponding factored forms. If f is a factored form (a node), $I(f)$ is the set of fanin nodes (factored forms) of f, $O(f)$ is the fanout node of f, $LEAVES(f)$ is the set of leaf nodes of f, and $NODES(f)$ is the set of all nodes of f. Notice that $O(f)$ is empty if f is a root. $OP(f) \in \{"*", "+", LEAF\}$ gives the label of f. The example in Figure 3.3 clarifies the above definitions. Node f is the root node, $g, h, x, y,$

[1] In actual implementation, a leaf node has an additional parameter indicating whether the leaf (literal) is complemented. Since this additional parameter is not need in describing algorithms, it is omitted.

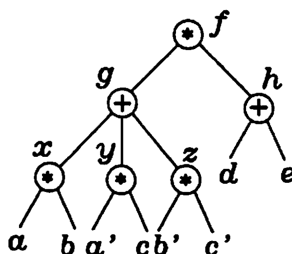


Figure 3.3: Factoring tree of $(ab + a'c + b'c')(d + e)$

and z are internal nodes, and $a, a', b, b', c, c', d, e$ are leaves. $I(g) = \{x, y, z\}$, $O(h) = f$, $OP(f) = " * "$, $OP(g) = " + "$, and $OP(a) = LEAF$.

Under these notations, two primitive subroutines are defined for manipulating factoring trees.

AddInput(f, x)
  $I(f) = I(f) \cup \{x\}$
  return f

DeleteInput(f, x)
  $I(f) = I(f) - \{x\}$
  return f

AddInput simply makes the node x an input of node f. If $f = ab$, then *AddInput*$(f, x) = abx$. If $f = g + h$, then *AddInput*$(f, x) = g + h + x$. *DeleteInput* simply removes node x from the fanins of f. It is the inverse operation of *AddInput*. Figure 3.4 shows graphically the affects of *AddInput* and *DeleteInput*.

3.3.3 Basic operations

Conjunction

Let f and g be two factoring trees. Let $AND(f, g)$ be a factored form of the logical conjunction of function f and g. Notice that $AND(f, g)$ is a logical operation expressed

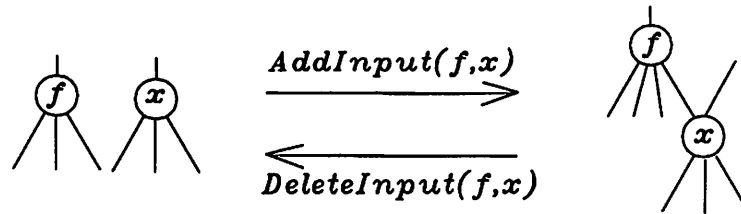


Figure 3.4: primitive tree manipulations

using factored forms. So, in addition to defining operation of $AND(f,g)$ on factored forms, we have to prove also that the resulting factored form is indeed a factored form of logic function fg.

$AND(f,g)$
  P = a new node.
  $OP(P) = "*"$
  $AddInput(P,f)$
  $AddInput(P,g)$
  return P.

Figure 3.5 illustrates the result graphically.

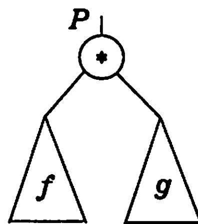


Figure 3.5: $AND(f,g)$

PROPOSITION 3.3.1 *The logic function represented by the factored form $AND(f,g)$ is the logical conjunction of logic functions represented by f and g.*

Proof. Any minterm which sets f and g both to 1 will set $AND(f,g)$ to 1. Otherwise the minterm will set $AND(f,g)$ to 0, because the root node of $AND(f,g)$ is labeled "*". ∎

PROPOSITION 3.3.2 *$AND(f, g)$ is an $O(1)$, constant, operation.*

Proof. The first two operations in the algorithm are trivially constant. *AddInput* involves inserting an element into a set which is, in general, a non-constant operation. But, in this application where we know that $g, f \notin I(P)$, the insertion is a constant operation. So, *AddInput* is a constant operation, and so is $AND(f, g)$. ■

Even though $AND(f, g)$ generates a correct factored form of fg, the resulting factored form can often be simplified and some of the simplifications are quite trivial. For example, if $f = abc$ and $g = abd$, $AND(f, g)$ returns a factoring tree $(abc)(abd)$ even though it can be easily simplified to $(abcd)$. However, the general problem of simplifying a factored form is quite difficult, and will be discussed at the end of this section. For now, we assume that there is a routine $SIMPLIFY(f)$ which can simplify the factored form f.

Disjunction

Logical disjunction can be handled in the exact same way as the conjunction, and all algorithms, propositions, and discussions of conjunction operation remain valid. So, we simply give the definition, the algorithm, and the corresponding propositions without further discussion.

Let f and g be two factoring trees. Let $OR(f, g)$ be a factored form of the logical disjunction of function f and g.

```
OR(f,g)
   P = a new node.
   OP(P) = "+"
   AddInput(P,f)
   AddInput(P,g)
   return P.
```

PROPOSITION 3.3.3 *The logic function represented by the factored form $OR(f, g)$ is the logical disjunction of logic functions represented by f and g.*

PROPOSITION 3.3.4 *$OR(f, g)$ is an $O(1)$, constant, operation.*

Inversion

Let f be a factored form. Let $INV(f)$ be a factored form of the logical inversion of function f.

```
INV(f)
   if OP(f) = LEAF {
      complement literal f
   if OP(f) = "*"
      OP(f) = " + "
   else /* OP(f) = "+" */
      OP(f) = " * "
   for each x ∈ I(f)
      INV(x)
   return f.
```

The algorithm is a direct application of DeMorgan's Law. It switches "*" and "+" and recursively inverts each fanins. At a leaf, it complements the literal, i.e. $x \to x'$ and $x' \to x$.

PROPOSITION 3.3.5 *The logic function represented by the factored form $INV(f)$ is the logical inversion of the logic function represented by f.*

Proof. This is DeMorgan's Law in its generalized form. ∎

PROPOSITION 3.3.6 *$INV(f)$ is an $O(\rho(f))$ operation, where $\rho(f)$ is the number of literals (leaf nodes) in f.*

Proof. The algorithm spends a constant amount of time on each node in the factoring tree f. Since there are $\rho(f)$ leaf nodes and each internal node has at least two fanins, the total number of nodes in the tree can be no more than $2\rho(f) - 1$. So, the algorithm INV is an $O(\rho(f))$ operation. ∎

Cofactoring

Recall the definition of cofactoring a factored form f with respect to a literal x. It replaces every occurrence of literal x with constant 1 and literal x' with constant 0, and propagates the constants using Boolean identities $1x = x, 0x = 0, 1 + x = 1$, and $0 + x = 0$. We first formalize the algorithm.

```
Cofactor(f, l_x)
   for each l ∈ LEAVES(f) s.t. l = x or l = x' {
      if ((l = x) and (l_x = x)  or  (l = x') and (l_x = x'))
         Propagate(FO(l), l, 1)
      else /* ((l = x) and (l_x = x')  or  (l = x') and (l_x = x)) */
         Propagate(FO(l), l, 0)
   }
   return f
```

```
Propagate(x, y, n)
   if OP(x) = " + "
      if n = 0
         DeleteInput(x, y)
      else /* n = 1 */
         Propagate(FO(x), x, 1)
   else /* OP(x) = " * " */
      if n = 0
         Propagate(FO(x), x, 0)
      else /* n = 1 */
         DeleteInput(x, y)
```

To show that procedure $Cofactor$ is well defined, we need to show that $Cofactor(f, x)$ is indeed an factored form of logic function f_x.

PROPOSITION 3.3.7 *$Cofactor(f, x)$ is a factored form of logic function f_x.*

Proof. To avoid confusion, let f be a logic function, F^f be its factored form, and f_x be the cofactoring of f with respect to x and F^f_x be $Cofactor(F^f, x)$. Let m be a minterm.

We say $m \in f$ if m is an implicant of f, $m \in F^f$ if F^f evaluates to 1 using input vector $V(m)$ in which a variable x is assigned 1 if literal x is in m, assigned 0 if literal x' is in m. We need to prove $F^f_x = F^{f_x}$ (i.e. F^f_x is a factored form of f_x).

$$\begin{aligned} m \in F^{f_x} &\iff m \in f_x \\ &\iff mx \in f \\ &\iff mx \in F^f \\ &\iff m \in F^f_x \end{aligned}$$

The last implication is true because F^f_x is just the partial result of evaluating F^f using $V(mx)$ where $V(m)$ has not yet been evaluated. Of course, when $V(m)$ does get evaluated, F^f_x has to be 1 because $mx \in F^f$, so $m \in F^f_x$. ∎

Like *AND* and *OR*, $Cofactor(f, x)$ is also quite efficient when the factored form f is minimal in some respect.

DEFINITION 3.3.8 *A factored form f is said to be alternating if $OP(x) \neq OP(y)$ $\forall x \in I(y)$, $\forall y \in NODES(f)$, i.e. for every node x in the factored tree of f, there is no fanin of x having the same label as x.*

PROPOSITION 3.3.9 *If the factored form of f is alternating, then $Cofactor(f, x)$ is an $O(\rho(f, x))$ operation, where $\rho(f, x)$ is the number times literals x and x' appear in f.*

Proof. First, its is easy to see that *Propagate*(x, y, n) is a constant time operation. There are two cases under which *Propagate* recursively calls itself. If $OP(x) =$ " * " and $n = 0$, then the constant to be propagated up is 0 and the label (operator) at the next level is "+". So, the propagation stops at the next level. If $OP(x) =$ " + " and $n = 1$, then the constant to be propagated up is 1 and the label (operator) at the next level is "*". So, the propagation again stops at the next level. Therefore, there can be no more than two levels of propagation for a given constant in routine *Propagate*, i.e. *Propagate* is an $O(1)$, constant, operation.

In *Cofactor*, with the help of certain data structure techniques, there is no need to loop over all leaves of f in order to find literals x or x'. In fact, it is easy to design a data structure in which this operation is $O(\rho(f, x))$. for instance, associate with each input variable x of f a linked list of pointers to the leaves of f which are either literal x or x'. Therefore, the whole algorithm $Cofactor(f, x)$ is an $O(\rho(f, x))$ operation. ∎

3.3.4 Generating kernels

Kernels [17] have been shown in several multi-level logic optimization systems [10] [14] to be good candidates for identifying common factors between a set of functions. This section presents methods for generating kernels from a given factored form directly and representing the kernels in their factored forms.

The following is a kerneling algorithm first proposed in [17] and has been shown in practice to be very efficient. It is outlined here for easy reference. The general strategy in the algorithm is to divide a function by a literal, obtain a kernel by making the quotient of the division cube-free, and proceed recursively to find lower level kernels. Full discussions of the algorithm can be found in [17].

$Kernel(f, k)$
    $f = cube_free(f)$
    $K = \{f\}$
    for $i = k$ to n {
        $c = largest_common_cube(f/l_i)$
        if $l_i \notin c$ *and* $\forall k \leq i$
            $K = K \cup Kernel(f/(l_i \cap c), i)$
    }
    return K

There are four basic operations used in the algorithm: dividing a function by a literal, dividing a function by a cube, finding the largest common cube of a function, making a function cube-free. It will become evident later on that dividing a function by a cube is equivalent to dividing by a literal, and making a function cube-free is equivalent to finding the largest common cube and dividing the function by that common cube. Therefore, only two operations are really needed and will be studied: dividing a function by a literal and finding the largest common cube of a function.

Kernels and operations used in generating kernels are all defined on algebraic sum-of-products expressions which are single-cube-containment minimum sum-of-products forms. When operating on factored forms, the notion of kernels is not yet well defined. Our approach is to define first the operations used to generate, from a factored form, a set of factored forms which will be shown to correspond to the set of kernels of an appropriately

derived algebraic expression.

Dividing a factored form by a literal

Let F, G, and H be factored forms. If F can be re-expressed as $xQ + R$ where Q and R are independent of literal x, then Q is called the quotient and R is called the remainder of dividing F by x. To avoid confusion, some times Q_x^F and R_x^F are used to denote the quotient and remainder of dividing F by x.

To compute the quotient and the remainder of a division, the following derivations are needed.

$$\begin{aligned}
F &= G + H \\
&= (xQ_x^G + R_x^G) + (xQ_x^H + R_x^H) \\
&= x(Q_x^G + Q_x^H) + (R_x^G + R_x^H) \\
&= xQ_x^F + R_x^F \\
F &= GH \\
&= (xQ_x^G + R_x^G)(xQ_x^H + R_x^H) \\
&= x(Q_x^G Q_x^H + Q_x^G R_x^H + R_x^G Q_x^H) + (R_x^G R_x^H) \\
&= xQ_x^F + R_x^F
\end{aligned}$$

So, we have the following formula for computing quotients and remainders.

$$Q_x^F = \begin{cases} Q_x^G + Q_x^H & \textit{if } \; F = G + H \\ Q_x^G Q_x^H + Q_x^G R_x^H + R_x^G Q_x^H & \textit{if } \; F = GH \end{cases}$$

$$R_x^F = \begin{cases} R_x^G + R_x^H & \textit{if } \; F = G + H \\ R_x^G R_x^H & \textit{if } \; F = GH \end{cases}$$

These equations suggest a recursive procedure for computing the quotient and remainder of dividing a factored form by a literal.

```
FACTOR_DIV(F,x)
  if OP(F) = LEAF {
    if F = x
      Q = 1
      R = 0
    else
      Q = 0
      R = F
  } else if OP(F) = " + " {
```

```
        Q = 0
        R = 0
        for each G ∈ I(F)
            (r,q) = FACTOR_DIV(G,x)
            Q = Q + q
            R = R + r
    } else /* OP(F) = "*" */ {
        Q = 0
        R = 1
        for each G ∈ I(F)
            (r,q) = FACTOR_DIV(G,x)
            Q = Q(q + r) + qR
            R = Rr
    }
    return (Q,R)
```

PROPOSITION 3.3.10 *$FACTOR_DIV(F,x)$ is an $O(\rho(F))$ operation.*

Proof. The algorithm works on each node twice, once on the way down and once on the way up. Since the total number of nodes in F is no more than $2\rho(F) - 1$, the algorithm is an $O(\rho(F))$ operation. ∎

Finding largest common cube

To simplify the algorithm description, the following definition of *cubes* is used. A *cube* is a set of literals. By definition, the empty set is the universal cube 1. For example, $abc = \{a,b,c\}$ and $abd' = \{a,b,d'\}$ are cubes. Cube AND in Boolean domain is set union in our notation. For example,

$$(abc)(abd') = (abcd') = \{a,b,c\} \cup \{a,b,d'\} = \{a,b,c,d'\}.$$

Furthermore, the common cube of two cubes is given by set intersection of the two cubes. For example,

$$common_cube(abc, abd') = \{a,b,c\} \cap \{a,b,d'\} = \{a,b\} = ab.$$

Let $CC(F)$ be the largest common cube of F. The above definition and examples make the following recurrence relation clear.

$$CC(F) = \begin{cases} CC(G) \cup CC(H) & if \;\; F = GH \\ CC(G) \cap CC(H) & if \;\; F = G + H \end{cases}$$

The following algorithm uses this recurrence relation to find the largest common cube of factored form F.

```
COMMON_CUBE(F)
   if OP(F) = LEAF
      CC = {F}
   else if OP(F) = " + "
      CC = {*} /* the set of all literals */
      for each G ∈ I(F)
         CC = CC ∩ COMMON_CUBE(G)
   else /* OP(F) = " * " */
      CC = φ
      for each G ∈ I(F)
         CC = CC ∪ COMMON_CUBE(G)
   return CC
```

Relationship with Kernels

Now, we have all the operations necessary to carry out the algorithm *Kernel* on factored forms. Let $FK(F)$ be the set of factored forms generated by the *Kernel* algorithm, using the new procedures $FACTOR_DIV$ and $COMMON_CUBE$ replacing f/l_i and *largest_common_cube*. We would like to establish some relationship between factors in $FK(F)$ and kernels in $K(E)$ where E is some algebraic expression derived from F. Our presentation begins with the following definitions.

DEFINITION 3.3.11 *Given a factored form f, let S^f be the sum-of-products form derived from f by multiplying f out literal by literal to a set of cubes and applying the following operations on the set of cubes:*

1. *Remove terms containing both a literal and its complement ($x'xc = 0$).*

2. *If a literal appears multiple times in a cube, remove all but one of the literals ($xxc = xc$).*

3. *Removing a cube if it is entirely contained in another cube ($abc + ab = ab$).*

S^f is called the derived sum-of-products of f, and the last operation is denoted by $SCC(S)$ which stands for single cube containment of a sum-of-products form.

The relationship between $FK(f)$ and $K(S^f)$ is given by the relationship between $S^{Q^f_x}$ and $Q^{S^f}_x$, and $CC(f)$ and $CC(S^f)$. Notice that $S^{Q^f_x}$ is the derived SOP of the quotient Q^f_x, $Q^{S^f}_x$ is the algebraic quotient of dividing the algebraic expression S^f by x, $CC(f)$ is the common cube computed from factored form f directly, and $CC(S^f)$ is the common cube derived from the algebraic expression S^f. If $S^{Q^f_x} = Q^{S^f}_x$ and $CC(f) = CC(S^f)$, then we have an ideal result $FK(f) = K(S^f)$.

LEMMA 3.3.12 *Let g and h be factored forms.*

$$S^{g+h} = SCC(S^g + S^h)$$
$$S^{gh} = SCC(S^g S^h)$$

Proof. Let g_1 and h_1 be the sum-of-products forms obtained by multiplying g and h out without the single-cube containment operation. We have,

$$SCC(S^g + S^h) = SCC(SCC(g_1) + SCC(h_1)) = SCC(g_1 + h_1) = S^{g+h}$$
$$SCC(S^g S^h) = SCC(SCC(g_1)SCC(h_1)) = SCC(g_1 h_1) = S^{gh}$$

■

LEMMA 3.3.13 *Let f be a sum-of-products expression, possibly non-algebraic. We have*

$$CC(f) = CC(SCC(f))$$

Proof. Let $c \in CC(f)$, $d, e \in f$ such that $d \subseteq e$. Since c is the largest common cube of f, d and e must be of forms pc and qc. So, c must be a common cube of $f - \{d\}$. If there is a literal l common to all the cubes of $f - \{d\}$, it must be in cube e and consequently must be in d since $d \subseteq e$. So, that literal must be in c because c is the largest common cube of f. So, c is also the largest common of $f - \{d\}$.

Let $c \in CC(SCC(f))$, $d, e \in f$ such that $d \subseteq e$. Then, e can be expressed as pc. Since $d \subseteq e$, d can be expressed as qc. So, c has to be a common cube of $SCC(f) \cup \{d\}$, and c has to be the largest because it is the largest common cube of $SCC(f)$, an expression with fewer cubes.

So we have, $CC(f) = CC(SCC(f))$. ■

Using Lemma 3.3.12 and Lemma 3.3.13, we can show that indeed $CC(f)$ is the same cube as $CC(S^f)$.

THEOREM 3.3.14 *Let f be a factored form, and S^f be the derived SOP of f. Then, $CC(f) = CC(S^f)$.*

Proof. By induction on the number of literals in f. It is trivially true if $\rho(f) = 1$.

If $f = gh$ then

$$\begin{aligned}
CC(f) &= CC(gh) \\
&= CC(g) \cup CC(h) \\
&= CC(S^g) \cup CC(S^h) && \textit{Induction} \\
&= CC(S^g S^h) \\
&= CC(SCC(S^g S^h)) && \textit{Lemma } 3.3.13 \\
&= CC(SCC(S^{gh})) && \textit{Lemma } 3.3.12 \\
&= CC(S^f)
\end{aligned}$$

If $f = g + h$ then

$$\begin{aligned}
CC(f) &= CC(g + h) \\
&= CC(g) \cap CC(h) \\
&= CC(S^g) \cap CC(S^h) && \textit{Induction} \\
&= CC(S^g + S^h) \\
&= CC(SCC(S^g + S^h) && \textit{Lemma } 3.3.13 \\
&= CC(SCC(S^{g+h}) && \textit{Lemma } 3.3.12 \\
&= CC(S^f)
\end{aligned}$$

■

DEFINITION 3.3.15 *Let S and T be two sum-of-products forms. S is dominated by T, denoted by $S \prec T$ if $\forall c \in S, c \in T$. S is simply a subset of cubes of T.*

For example, $ab + cd$ is dominated by $ab + cd + ade$ and $a + b$ is dominated by $a + b$.

LEMMA 3.3.16 *Let S be a sum-of-products expression. Then $Q_x^{SCC(S)} \prec SCC(Q_x^S)$*

Proof. First, we have

$$\begin{aligned} c \in Q_x^{SCC(S)} &\Rightarrow xc \in SCC(S) \\ &\Rightarrow xc \in S \\ &\Rightarrow c \in Q_x^S \\ &\Rightarrow c \in SCC(Q_x^S) \end{aligned}$$

The last implication is true because if $c \notin SCC(Q_x^S)$, xc would have been redundant which contradicts to $cx \in SCC(S)$. ■

For example, let $S = abcx + dx + ab$. Then, $Q_x^{SCC(S)} = d \prec SCC(Q_x^S) = abc + d$.

THEOREM 3.3.17 *Let f be a factored form and S^f be its derived SOP form. Then, $Q_x^{S^f} \prec S^{Q_x^f}$.*

Proof. By induction on the number of literals in f. When $\rho(f) = 1$, the relation is trivially true. Now, let $f = g \circ h$ where $\circ$ is either "*" or "+".

$$\begin{aligned} Q_x^{S^f} &= Q_x^{SCC(S^g \circ S^h)} & \\ &\prec SCC(Q_x^{(S^g \circ S^h)}) & \textit{Lemma } 3.3.16 \\ &= SCC(Q_x^{S^g} \circ Q_x^{S^h}) & \\ &\prec SCC(S^{(Q_x^g \circ Q_x^h)}) & \textit{Induction} \\ &= S^{(Q_x^g \circ Q_x^h)} & \\ &= S^{Q_x^f} & \end{aligned}$$

■

EXAMPLE:

$$O = (a + g)((b + c)(d + x) + e) + b(a + f)$$
$$S^{Q_x^O} = S^{(a+g)(b+c)} = ab + ac + bg + cg$$
$$Q_x^{S^O} = ac + bg + cg$$

Theorem 3.3.14 shows that the common cubes found on a factored form f by $COMMON_CUBE(f)$ and on the SOP form S^f are the same. Theorem 3.3.17 shows that the derived SOP of the quotient Q^f_x dominates $Q^{S^f}_x$, the quotient of dividing the derived SOP form of f by x. So, combining these two, we can make the following claim.

THEOREM 3.3.18 *Let f be a factored form. For every kernel $k \in K(S^f)$, there is a $g \in FK(f)$ such that $k \prec S^g$, i.e. k is dominated by S^g.*

It may appear on the surface that Theorem 3.3.18 decreases the usefulness of $FK(f)$. However, looking at it carefully yields the following result.

THEOREM 3.3.19 *Let $k \in K(S^f)$ and $g \in FK(f)$ where $k \prec S^g$. Let c be a co-kernel [14] of k and $d \in S^g$ such that $d \notin k$. Then, cube cd is redundant.*

Proof. Because $S^f = ck + r$ such that $k = Q^{S^f}_c$, and $d \notin k$, then cd is covered by $ck + r$. ■

Theorem 3.3.19 shows that a factor g in $FK(f)$ not only contains a kernel of $K(f)$, but also some redundant terms which can be used to find better common factors between functions. For example, let f_1, f_2, and f_3 be three logic functions in factored forms. Let $k_1 \in K(S^{f_1})$, $k_2 \in K(S^{f_2})$, $k_3 \in K(S^{f_3})$, $g_1 \in FK(f_1)$, $g_2 \in FK(f_2)$, and $g_3 \in FK(f_3)$. Suppose

$$\begin{aligned} k_1 &= c_2 + c_3, & g_1 &= c_1 + c_2 + c_3 \\ k_2 &= c_1 + c_3, & g_2 &= c_1 + c_2 + c_3 \\ k_3 &= c_1 + c_2, & g_3 &= c_1 + c_2 + c_3 \end{aligned}$$

where c_i is redundant in k_i for $i = 1, 2, 3$. It is easy to see that $c_1 + c_2 + c_3$ is a better common factor (Boolean) of three functions because of the introduction of the redundant cubes. This is not to say that $FK(f)$ finds all the redundant cubes. It simply shows that the redundant cubes introduced in $FK(f)$ may help improve the results.

In multi-level logic optimization, we often want to find intersections of kernels. Two problems remain open: how to find kernel intersections on the factored forms directly? and how to detect the intersections which consist of only redundant cubes?

3.3.5 Higher level operations

We listed a set of high level operations of factored forms which are useful in a multi-level logic synthesis and optimization system. Only a few will be discussed here to

show that they are all reduced to a hard problem of simplifying a factored form directly with a don't-care set also in a factored form.

Dividing a factored form by another

In Boolean algebra, there are many ways of defining division. 2 defined two divisions, $WEAK_DIV$ and $STRONG_DIV$, which operate on the sum-of-products form of logic functions. In this section, another division will be defined which operates on factored from of logic functions. First, let's look at some properties common to all divisions.

LEMMA 3.3.20 *Let* $f = q_x x + r_x = q_y y + r_y$, *then* $q_x \overline{q_y} x \subseteq r_y$ *and* $q_y \overline{q_x} y \subseteq r_x$.

Proof. First, $q_x \overline{q_y} x \subseteq q_x x \subseteq f$. But $q_x \overline{q_y} x \cap q_y y = \phi$. So, $q_x \overline{q_y} x \subseteq r_y$. Similarly, $q_y \overline{q_x} y \subseteq r_x$. ∎

THEOREM 3.3.21 *Let* f *and* g *be any logic functions, and let* $g = x + y$. *If* $f = q_x x + r_x$ *and* $f = q_y y + r_y$, *then* $f = q_x q_y g + r_x + r_y$.

Proof.

$$
\begin{aligned}
f &= q_x x + r_x \\
&= q_y y + r_y \\
&= q_x x + q_y y + r_x + r_y \\
&= q_x q_y x + q_y y + r_x + r_y & \quad q_x \overline{q_y} x \subseteq r_y \\
&= q_x q_y x + q_x q_y y + r_x + r_y & \quad \overline{q_x} q_y y \subseteq r_x \\
&= q_x q_y g + r_x + r_y
\end{aligned}
$$

∎

THEOREM 3.3.22 *Let* f *and* g *be any logic functions, and let* $g = x * y$. *If* $f = q_x x + r_x$ *and* $f = q_y y + r_y$, *then* $f = q_x q_y g + r_x + r_y$.

Proof.

$$
\begin{aligned}
f &= q_x x + r_x \\
&= q_y y + r_y \\
&= (q_x x + r_x)(q_y y + r_y) \\
&= q_x q_y xy + r_x q_y y + q_x x r_y + r_x r_y \\
&= q_x q_y xy + r_x + q_x x r_y + r_x r_y \\
&= q_x q_y xy + r_x + r_y + r_x r_y \\
&= q_x q_y xy + r_x + r_y \\
&= q_x q_y g + r_x + r_y
\end{aligned}
$$

■

These theorems have several interesting implications. First of all, it provides a definition of quotient and remainder of dividing f by g regardless of whether $g = xy$ or $g = x + y$. Furthermore, this definition is independent of particular representation of f and g. The theorems also suggest a recursive procedure for computing the division. The following algorithm carries out the division on factored forms of logic functions, but can be easily extended to operate on other representations.

```
FDIV(f,g)
   Q = 1
   R = 0
   for each x ∈ I(g) {
      (q,r) = FDIV(f,x)
      Q = AND(Q,q)
      R = OR(R,q)
   }
   return (Q,R)
```

Notice that AND and OR operate on the factored forms and return the results in factored forms. In addition, by definition of factored forms, the operator of g is either "+" or "*" which do not affect the resulting Q and R by previous theorems. However, in multi-level logic optimization, we are often interested in a quotient and a remainder which are simpler than the function itself and the procedure $FDIV$ does not guarantee that. So,

the following can be used to simplify the quotient and remainder:

$$Q = Simplify(Q, \overline{g} + R)$$
$$R = Simplify(R, gQ)$$

where the routine *Simplify* simplifies the first argument using the second argument as the don't-care set. When using factored forms, the simplification should be done on the factored forms directly.

Substitution

To substitute a factored form g into another factored form f, *Simplify* is again needed. In the following algorithm, a new variable x is introduced to represent function g in f after the substitution.

Substitution(f, g)
    Simplify$(f, OR(AND(x, INV(g)), AND(x', g)))$

Simplification - An open problem

Simplification can be done at various levels depending on the quality requirement and run time constraint. The most general form of simplification is to simplify a factored form with a don't-care set. However, simplification just on a factored form without the don't-care set is also useful in many situations. Currently, these two problems are still open.

Some simpler forms of simplification are possible. For example, constant propagation can be viewed as one form of simplification. Putting a factored form into alternating form is another form of simplification because it reduces the number of internal nodes in the factored form. Next, we propose a simplification algorithm which guarantees the result to be maximally factored. To do so, it needs a routine to check for $f-equivalence$ of two factored forms. By definition, two factored forms are f-equivalent if their factoring trees are isomorphic. Checking for tree isomorphism is a simple problem and is denoted by $Fequiv(f, g)$. The simplification routine is called $Fsim1(f)$. It assumes that f is in alternating form to begin with.

```
Fsim1(f)
   for each x ∈ I(f)
      x = Fsim1(x)
   while ∃x,y ∈ I(f) s.t. ∃u ∈ I(x), v ∈ I(y) and Fequiv(u,v) {
      g = new node
      OP(g) = OP(x)
      AddInput(g,u)
      AddInput(g,f)
      DeleteInput(x,u)
      DeleteInput(y,v)
      f = g
      Alternate(f)
   }
   return f
```

In the algorithm, *Alternate* puts a factored form into an alternating form. *Fsim*1 is a recursive algorithm. At each level of recursion, it first simplifies all the fanin nodes. Then, it performs the following operation assuming u and v are f-equivalent:

$$\begin{aligned} up + vq &\rightarrow u(p+q) \\ (u+p)(v+q) &\rightarrow u + pq \end{aligned}$$

3.4 Heuristic Factoring Algorithms

Factoring is the process of deriving a factored form of a given logic function represented in a sum-of-products form. The objective is to minimize the number of literals in the final factored form. Algorithms have been presented for solving exactly the problem of determining the optimum factored form [35]. However, in recent experiments using some modern extensions [50] for logic function manipulation, the complexity of these exact techniques still appear to be impractical for all but the smallest functions. The goal of this section is to develop fast heuristic factoring algorithms which rely on kernels [17] and find optimal factored forms.

As we have seen before, there are many equivalent factored forms of a given function. The difference in the number of literals of these equivalent factored forms obtained by various algorithms could be significant. For example, given the algebraic expression

$$abg + acg + adf + aef + afg + bd + ce + be + cd,$$

the following are three equivalent factored forms obtained by three different algorithms:

$$(b+c)(d+e)+((d+e+g)f+(b+c)g)a,$$
$$(b+c)(d+e+ag)+(d+e+g)af,$$
$$(af+b+c)(ag+d+e).$$

There are 12 literals in the first factored form and only 8 literals in the last one.

The previous example also shows two different kinds of factored forms. Taking the first factored form and multiplying it out literal by literal to get a sum of cubes without using Boolean identities $xx' = 0$ and $xx = x$, we can in fact recover the original sum-of-products form. But if we took the last factored form and multiplied it out in the same way, we would have got a different, non-algebraic, sum-of-products form because of the cube $afag$. This leads to the following definitions of algebraic and Boolean factored forms.

DEFINITION 3.4.1 *Let f be a factored form. f is said to be algebraic if the sum-of-products form obtained by multiplying f out directly (without using $xx' = 0$, $xx = x$, and single-cube containment) is algebraic. f is a Boolean factored form if it is not algebraic.*

For example, each of the following is an algebraic factored form:

$$a+bc,$$
$$(a+b)(c+d),$$
$$(b+c)(d+e+ag)+(d+e+g)af,$$

and each of the following is a Boolean factored form:

$$(a+b+c+d)(a'+b'+c'+d'),$$
$$(af+b+c)(ag+d+e),$$
$$(a+b)((c+d)(e+f)+g)+b(e+h).$$

None of the sum-of-products forms obtained by multiplying the Boolean factored forms out are algebraic. They either contain terms which are 0 ($xx'c = 0$), terms with redundant literals ($xxc = xc$), or redundant terms ($abc+ab = ab$). Some factoring algorithms guarantee the results to be algebraic and others do not. One way of classifying factoring algorithms is by the following definition:

DEFINITION 3.4.2 *A factoring algorithm is said to be algebraic if it guarantees the results to be algebraic starting from an algebraic sum-of-products expression. A factoring algorithm is Boolean if it is not algebraic.*

So far, the examples we have used are completely specified functions. In the process of multi-level logic optimization, we often encounter incompletely specified functions. Algorithms for factoring incompletely specified functions usually take very different approaches than those for factoring completely specified functions. The remaining of this section focuses mainly the algorithms for factoring completely specified functions.

3.4.1 Generic Factoring Algorithm

All heuristic factoring algorithms described in this section use the same top-down paradigm. Given a function F, routine $DIVISOR(F)$ is used to find a candidate divisor, D, which when substituted into F can simplify the expression. Then, the quotient (co-divisor) Q is found by dividing D into F using routine $DIVIDE(F, D)$. Now, the function can be represented as a partial factored form $F = (Q)(D) + (R)$ where R is the remainder. The algorithms then proceed to factor F, D, and R separately using the same method. This top-down recursive approach is described in the following procedure.

$FACTOR(F)$

1. if F has no factor, return
2. $D = DIVISOR(F)$
3. $(Q, R) = DIVIDE(F, D)$
4. return $FACTOR(Q)FACTOR(D) + FACTOR(R)$

Certain refinements of $FACTOR$ are needed to produce good results. We first use several examples to motivate the ideas behind the refinements. In each example, we list the original function F, the divisor D, the quotient Q, the partial factored form P, and the final factored form O given by $FACTOR$. It is important to point out that the discussions are restricted to algebraic operations only.

EXAMPLE:

$$F = abc + abd + ae + af + g$$
$$D = c + d$$
$$Q = ab$$
$$P = ab(c + d) + ae + af + g$$
$$O = ab(c + d) + a(e + f) + g$$

Obviously, O is not optimal because it is not maximally factored. It can be further factored to

$$a(b(c + d) + e + f) + g.$$

The problem occurs when the quotient Q is a single cube, and some of the literals of Q also appear in the remainder R. To solve the problem we first check the quotient Q. If Q is a single cube, we pick a literal in the cube which occurs in the greatest number of cubes of F. We then divide F by Q_1, the chosen literal, to obtain a new divisor D_1. Now, F has a new partial factored form $(Q_1)(D_1) + (R_1)$ and literal Q_1 does not appear in R_1 any more. Notice that the new divisor D_1 contains the original D as a divisor because Q_1 is a literal out of Q. When recursively factoring D_1, D will be discovered again.

LEMMA 3.4.3 *If Q is a single cube, the procedure outlined above guarantees that the partial factored form $Q_1D_1 + R_1$ is maximally factored.*

Proof. Let $D_1 = cD_2$ where c is a common cube of D_1 and D_2 is cube-free. Now, the partial factored form is

$$F = Q_1cD_2 + R_1.$$

It is easy to see that every literal in c must be in the original cube Q. Since Q_1 is the literal of Q occurring in the greatest number of cubes of the original sum-of-products expression, no literal in c can occur in R_1. In addition, D_1 was obtained by dividing F by literal Q_1, so Q_1 does not occur in R_1. Suppose D_2 is also a factor of R_1. Then the quotient of dividing F by D would have contained more than one cube since D_2 must contain D, which contradicts the fact that Q is a single cube. So, no factors of the product form Q_1cD2 can be a divisor of R_1. F is by definition maximally factored at this level. ∎

EXAMPLE:

$$F = ace + ade + bce + bde + cf + df$$
$$D = a + b$$
$$Q = ce + de$$
$$P = (ce + de)(a + b) + (c + d)f$$
$$O = e(c + d)(a + b) + (c + d)f$$

Again, the final factored form O is not maximally factored because $(c + d)$ is common to both products $e(c+d)(a+b)$ and $(c+d)f$. The final factored form should have been

$$(c + d)(e(a + b) + f).$$

The problem is that Q has a factor which is also a factor of R. The problem is solved by first making Q cube-free to get Q_1, then obtaining a new divisor D_1 by dividing F by Q_1. If D_1 is cube-free, we have obtained a partial factored form $F = (Q_1)(D_1) + R_1$, and can recursively factor Q_1, D_1, and R_1. If D_1 is not cube-free, let $D_1 = cD_2$. Then, the partial factored form becomes $F = c(Q_1)(D_2) + R_1$. Let $D_3 = Q_1D_2$, and we have a partial factoring $F = cD_3 + R_1$ which is the case illustrated by the previous example and can therefore be factored maximally. Therefore the problem is if c exists to take the most recurring literal in c and use that recursively factoring the quotient and remainder.

LEMMA 3.4.4 *If Q is not a single cube, the procedure outlined above maximally factorizes F at this level.*

Proof. Suppose D_1 is cube-free, then the partial factored form is $F = (Q_1)(D_1) + R_1$. Here, Q_1 cannot be a factor of R_1 because Q_1 is used to obtain D_1 by dividing into F. D_1 cannot be a factor of R_1 because D_1 contains a factor D which when dividing into F gives quotient Q_1. So, the partial factored form is maximal at this level. If D_1 is not cube-free, then we have a partial factored form $F = cD_3 + R_1$ which is factored by our previous procedure. Lemma 3.4.3 guarantees that the partial factored form obtained is maximal. ∎

Now, *FACTOR* can be improved by the procedures proposed. The new routine is called *GFACTOR* which stands for *Generic Factoring*. *GFACTOR* takes as an input a function and two more parameters which specify how to find the initial divisor D and how

to perform the division. As we'll see later on, by varying these two parameters, a spectrum of algorithms can be obtained with different run time versus quality tradeoffs.

```
GFACTOR(F,DIVISOR,DIVIDE)
  D = DIVISOR(F)
  if D = 0 then
    return F
  Q = DIVIDE(F,D)
  if |Q| = 1 {
    return LF(F,Q,DIVISOR,DIVIDE)
  } else {
    Q = make_cube_free(Q)
    (D,R) = DIVIDE(F,Q)
    if cube_free(D) {
      Q = GFACTOR(Q,DIVISOR,DIVIDE)
      D = GFACTOR(D,DIVISOR,DIVIDE)
      R = GFACTOR(R,DIVISOR,DIVIDE)
      return Q × D + R
    } else {
      C = common_cube(D)
      return LF(F,C,DIVISOR,DIVIDE)
    }
  }
```

```
LF(F,C,DIVISOR,DIVIDE)
  L = best_literal(F,C)
  (Q,R) = DIVIDE(F,L)
  C = common_cube(Q)
  Q = GFACTOR(Q,DIVISOR,DIVIDE)
  R = GFACTOR(R,DIVISOR,DIVIDE)
  return L × C × Q + R
```

The subroutine $LF(F,C)$ is a variation of *literal factoring* algorithm [9]: *best_literal* selects a literal in C which occurs in the most number of cubes of F. *common_cube*(Q) returns the largest common cube of Q. Instead of calling recursively LF on factors Q and R, we switch back to the generic factoring algorithm $GFACTOR$.

The following theorem shows that the results of $GFACTOR$ are always maximally factored.

THEOREM 3.4.5 *Algorithm $GFACTOR$ finds a maximally factored form.*

Proof. By induction on number of literals in the sum-of-products form of F. If F has only one literal, it is obviously maximal. Suppose F has n literals, each of the factors passed to the next recursive call of $GFACTOR$ has no more than $n-1$ literals, and, by the induction hypothesis $GFACTOR$ returns a maximally factored form. By Lemma 3.4.3 and Lemma 3.4.4, the factored form derived by $GFACTOR$ at this level is also maximal. So, the results of $GFACTOR$ are always maximally factored. ∎

Specific factoring algorithms are instances of $GFACTOR$ with specific choices of $DIVISOR$'s and $DIVIDE$ algorithms. Depending on a particular application of $GFACTOR$, appropriate $DIVISOR$ and $DIVIDE$ algorithms are chosen to obtain desired run time versus quality tradeoffs.

Notice that this result uses algebraic division for $DIVIDE$ everywhere. Whether the result extends to using strong division is still an open problem.

3.4.2 Quick Factoring

The basic operations of $GFACTOR$ are $DIVISOR$ and $DIVIDE$. Quick Factoring (QF) is a version of $GFACTOR$ in which $DIVISOR$ is replaced by $QUICK_DIVISOR$ which is the quickest way of finding a useful divisor and $DIVIDE$ is replaced by $ALGE_DIV$ which is the algebraic division [17] operation.

$QUICK_DIVISOR$ is a simple modification of the kerneling algorithm [17] that finds just one level-0 kernel.

```
QUICK_DIVISOR(F)
   If |F| ≤ 1
      return φ
   If every literal of F appears once
      return φ
   return ONE_0K(F)
```

$ONE_0K(F)$
  If every literal of F appears once
    return F
  l = Pick a literal appearing more than once
  $F = make_cube_free(F/l)$
  return $ONE_0K(F)$

If F has only one term or is itself a level-0 kernel, then $QUICK_DIVISOR$ returns ϕ. Otherwise, there is at least one multiple-cube divisor which is a level-0 kernel F and is not equal to F. This level-0 kernel is found by routine ONE_0K which arbitrarily picks a literal which appears more than once, divides F by the literal, and works recursively on the quotient. ONE_0K terminates on finding the first level-0 kernel which is when every literal appears only once.

With $QUICK_FACTOR$, quick factoring (QF) can now be defined as:

$QF(F)$
  return $GFACTOR(F, QUICK_DIVISOR, ALGE_DIV)$

3.4.3 Good Factoring

The experiments have shown that QF is very fast and in many cases finds good factored forms. But, because $QUICK_DIVISOR$ finds an arbitrary level-0 kernel, the results in some cases are not satisfactory. For example, in factoring the function

$$abg + acg + adf + aef + afg + bd + be + cd + ce,$$

$QUICK_DIVISOR$ may have chosen a level-0 kernel $(d+e+g)$ which leads to the following factored form

$$a(g(b+c) + f(d+e+g)) + (d+e)(b+c)$$

that has 12 literals. However, if we spend more time to examine all the kernels and choose $(af+b+c)$ as the divisor, we would obtain a better factored form

$$(af+b+c)(d+e) + ag(f+c+b)$$

with 11 literals.

Good factoring (GF) tries to obtain a better result by working harder to find a good divisor to start with. In particular, $GF(F)$ looks at all the kernels and picks one (k) which, when substituted into F, maximally reduces the total number of sum-of-products literals of F and k. This procedure is called $BEST_KERNEL$. Now, GF can be defined.

$GF(F)$
  return $GFACTOR(F, BEST_KERNEL, ALGE_DIV)$

3.4.4 Boolean Factoring

GF can be further improved by replacing $ALGE_DIV$ with boolean division $BOOL_DIV$ [14]. Using the above example,

$$GF(F) = (af + b + c)(d + e) + ag(f + c + b),$$

If $BOOL_DIV$ were used to divide the divisor $(af + b + c)$ into F, the quotient and the remainder would have been $(ag + d + e)$ and 0, which would have lead to the following factored form

$$(af + b + c)(ag + d + e)$$

with only 8 literals. Notice that this is no longer an algebraic factored form because of the term $afag$. This version of $GFACTOR$ is called Boolean factoring BF.

$BF(F)$
  return $GFACTOR(F, BEST_KERNEL, BOOL_DIV)$

Since $BOOL_DIV$ involves a two-level logic minimization step, it is a much more expensive operation than $ALGE_DIV$. Consequently, BF takes considerably more time than QF and GF. But, because it uses Boolean division, it is able to find some Boolean

factored forms with significantly fewer literals, as shown by the following example

$$\begin{aligned}
F &= abc'd' + abe'f' + a'b'cd + a'b'ef + cde'f' + c'd'ef, \\
QF(F) &= abc'd' + a'b'cd + ef(c'd' + a'b') + e'f'(cd + ab), \\
GF(F) &= cde'f' + c'd'ef + a'b'(ef + cd) + ab(e'f' + c'd'), \\
BF(F) &= (e'f' + c'd' + a'b')(ef + cd + ab).
\end{aligned}$$

It should be pointed out that this is a contrived example intended to show the potential power of Boolean factoring algorithms. In practice, as indicated in the results section of this chapter, most of the functions found in real circuits can be factored just as well by algebraic methods as by Boolean methods and algebraic methods are much faster.

3.4.5 Complement Factoring

The principle of duality in Boolean algebra implies that to complement a factored form is to exchange "*" and "+" and change the polarities of literals ($x \rightarrow x'$ and $x' \rightarrow x$). As pointed out earlier, one of the advantages of a factored form is that it represents both a function and, by duality, its complement. To state it differently, if function F has a factored form with n literals, then there is a factored form of F' with the same number of literals. However, sum-of-products forms do not have this nice property. All the previous factoring algorithms QF, GF, and BF start from a given sum-of-products representation, or more precisely a kernel of that expression. So, the results depend on the phase of the initial representation. For example, the following function

$$F = a'b + a'c + a'd + b'a + b'c + b'd + c'a + c'b + c'd + d'a + d'b + d'c$$

when factored using QF gives

$$QF(F) = (d + c)(a' + b') + (a + b)(c' + d') + c'd + a'b + cd' + ab'.$$

But, if F is complemented, it becomes

$$F' = abcd + a'b'c'd'$$

which is already an optimum factored form. Complementing it again using duality, we get

$$F = (a + b + c + d)(a' + b' + c' + d').$$

So, complement factoring CF simply complements a function, uses one of the QF, GF, or BF to obtain a factored form, and complements the factored form again using duality.

```
CF(F)
    return DUAL(XF(F')) /* XF is either QF, GF, or BF */
```

3.4.6 Dual Factoring

Instead of factoring either a function or its complement separately, duality can be further explored at each level of recursion during factoring. The idea is to use the function and its complement to help find divisors, and choose a better one before factoring each of the factors. Dual factoring DF uses this idea.

```
DF(F)
    D1 = DIVISOR(F)
    (Q1, R1) = DIVIDE(F, D1)
    D2 = DIVISOR(F')
    (Q2, R2) = DIVIDE(F', D2)
    if |D1| + |Q1| + |R1| < |D2| + |Q2| + |R2|
        return DF(Q1)DF(D1) + DF(R1)
    else
        return DUAL(DF(Q2)DF(D2) + DF(R2))
```

The algorithm derives $F = Q_1D_1 + R_1$ and $F' = Q_2D_2 + R_2$ and chooses a smaller one to continue the factoring. The size is measured by $|X|$ which could either be the number of product terms of X or the number of sum-of-products literals of X.

3.4.7 Factoring Incompletely Specified Functions

In multi-level logic synthesis and optimization, functions encountered are usually incompletely specified. Factoring an incompletely specified function is a problem of finding a cover in the factored form with a minimum number of literals. The following procedure outlines one possible algorithm.

$IF(F, DC)$
  $D = DIVISOR2(F, DC)$
  $(Q, R) = DIVIDE2(F, DC, D)$
  $D = IF(D, DC + Q' + R)$
  $Q = IF(Q, DC + D' + R)$
  $R = IF(R, DC + QD)$
  return $DQ + R$

The procedure factors the incompletely specified function (F, DC) with ON-set F and don't-care set DC. $DIVISOR2$ is used to find a divisor of (F, DC). Then, $DIVIDE2$ is used first to derive an initial factoring $QD + R$. When D is 0 or R is 1, the value of Q does not effect the output value of F. So, $DC + D' + R$ can be used as the don't-care set to recursively simplify Q. Similarly, D can be simplified using don't-care set $DC + Q' + R$. Finally, when QD is 1, F is 1 regardless of the value of R. So, the last step is to simpilfy R using the don't-care set $DC + QD$. Notice that this is not a complete algorithm because $DIVISOR2$ and $DIVIDE2$ are not specified.

3.4.8 Summary of Factoring Algorithms

To summarize, the factoring algorithms presented in this section are all based on a recursive paradigm: find first a divisor as the initial seed, and then use several divisions to try to improve the factors before recursively factoring them. All the algorithms use heuristics, i.e. at each step, the quality of the factoring is estimated by the literals in the sum-of-products form of each factor. Furthermore, because the initial divisors generated by $DIVISOR$ are restricted to kernels only, the results of factoring largely depend on the initial sum-of-products forms. The experiments as presented in the results section show that the heuristic works well for most of the functions and the results are in general quite good.

In multi-level logic optimization, the cost of a Boolean network has to be evaluated again and again. So, factoring algorithms are used constantly to estimate the number of literals in the factored forms of functions. In this application environment, the speed of the factoring algorithm is essential. QF seems to be particularly useful in this environment. In later stages of the optimization, a more accurate measure of network size and exact

implementation of functions are needed. Therefore, more expensive algorithms such as GF or BF can be used. In fact, since all the algorithms presented are heuristic based, there is no theoretical guarantee that one can always outperform the others. So, when the quality of results is essential, one should try all the algorithms and keep the best result. Table 3.1 of the results section of this chapter summarizes the quality and performance for various factoring algorithms presented in this section.

3.5 Properties of Optimum Factored Form

Understanding the properties of optimum factored forms can help us to design better factoring algorithms, to employ heuristics appropriately, and to justify the quality of results by deriving good lower bounds. In this section, conditions are derived under which the optimum factored form of a function consists of optimum factored forms of its sub-functions. These conditions can then be used whenever possible to break a problem of factoring a large function into factoring a set of smaller functions without loss of optimality. Next, the unateness properties of logic functions and its implications on their factored forms are studied. Then, properties of optimum factored forms of incompletely specified functions are investigated. Tests will be derived to identify "essential" and "redundant" variables.

3.5.1 Optimum Factoring Theorems

When a function has certain properties, its optimum factored form consists of optimum factored forms of sub-functions. More precisely, if a function is a sum or a product of several sub-functions whose supports are mutually disjoint, then the optimum factored form consists of the sum or the product of optimum factored forms of the sub-functions.

DEFINITION 3.5.1 *Let f be a completely Boolean function, $\rho(f)$ be the minimum number of literals in any factored form of f. Recall that $\rho(F)$, when F is a factored form, is the number of literals in F. Let $sup(f)$ be the variable support of f, i.e. set of variables f depends on. Two functions f and g are said to be orthogonal, denoted by $f \perp g$, if $sup(f) \cap sup(g) = \phi$.*

LEMMA 3.5.2 *Let $f = g + h$ such that $g \perp h$, then $\rho(f) = \rho(g) + \rho(h)$.*

Proof. Let F, G and H be the optimum factored forms of f, g and h respectively. Since $G + H$ is a factored form of f, we have trivially

$$\rho(F) \leq \rho(G) + \rho(H)$$

Let c be a minterm of $\overline{g}$. Since g and h have disjoint support, we have

$$f_c = (g+h)_c = g_c + h_c = \overline{\overline{g}_c} + h_c = \overline{1} + h_c = 0 + h_c = h_c = h$$

and similarly if d is a minterm of $\overline{h}$ we have $f_d = g$. Because $\rho(h) = \rho(f_c) \leq \rho(F_c)$ and $\rho(g) = \rho(f_d) \leq \rho(F_d)$, we have

$$\rho(g) + \rho(h) \leq \rho(F_c) + \rho(F_d).$$

Now, let m (n) be the number of literals in F which are from $sup(g)$ ($sup(h)$). When computing F_c (F_d), we replace all the literals from $sup(g)$ ($sup(h)$) by the appropriate values and simplify the factored form by eliminating all the constants and possibly some literals from $sup(h)$ ($sup(g)$) using the Boolean identities. So, we have $\rho(F_c) \leq n$ and $\rho(F_d) \leq m$. Since $\rho(F) = m + n$, we have

$$\rho(F_c) + \rho(F_d) \leq n + m = \rho(F)$$

Putting everything together, we have

$$\rho(f) \leq \rho(g) + \rho(h) \leq \rho(F_c) + \rho(F_d) \leq \rho(F) = \rho(f)$$

■

Lemma 3.5.2 shows that the optimum factoring of f can be obtained by optimally factoring g and h separately. But this does not imply that all minimum literal factored forms of f are sums of the minimum literal factored forms of g and h.

COROLLARY 3.5.3 *Let $f = gh$ such that $g \perp h$, then $\rho(f) = \rho(g) + \rho(h)$.*

Proof. Let $\overline{F}$ denotes the factored form obtained using DeMorgan's law. Then, we have $\rho(F) = \rho(\overline{F})$ and therefore $\rho(f) = \rho(\overline{f})$. Plus the above theorem we have

$$\rho(f) = \rho(\overline{f}) = \rho(\overline{g} + \overline{h}) = \rho(\overline{g}) + \rho(\overline{h}) = \rho(g) + \rho(h)$$

■

THEOREM 3.5.4 *Let $f = \sum_{i=1}^{n} \prod_{j=1}^{m} f_{ij}$ such that $f_{ij} \perp f_{kl} \forall i \neq k$ or $j \neq l$, then $\rho(f) = \sum_{i=1}^{n} \sum_{j=1}^{m} \rho(f_{ij})$.*

Proof. Use induction on m and then n, and Lemma 3.5.2 and Corollary 3.5.3. ∎

3.5.2 Essential Variables

In factoring an incompletely specified function, knowing that certain variables must appear in every factored form may help to trim the search space and deriving a divide-and-conquer type algorithm to break a large problem into a set of smaller ones.

DEFINITION 3.5.5 *For an incompletely specified function (F, D), variable x is essential if x must appear in every factored form of (F, D).*

THEOREM 3.5.6 *For an incompletely specified function (F, D), x is essential if and only if $F_x \not\subseteq F_{\overline{x}} + D_{\overline{x}}$ or $F_{\overline{x}} \not\subseteq F_x + D_x$.*

Proof. Without loss of generality, assume $F_x \not\subseteq F_{\overline{x}} + D_{\overline{x}}$. Then there is a minterm m of F_x not covered by $F_{\overline{x}} + D_{\overline{x}}$, i.e. $xm \in F$ and $\overline{x}m \notin F + D$. Any function g that is independent of x and covers xm must also cover $\overline{x}m$, therefore it cannot be a cover of (F, D).

If $F_x \subseteq F_{\overline{x}} + D_{\overline{x}}$ and $F_{\overline{x}} \subseteq F_x + D_x$, then $F_x + F_{\overline{x}}$ is a cover of (F, D) independent of x. ∎

This Theorem gives the exact condition for a variable to be essential. Knowing that certain variables of a function are essential, we can in some cases restrict out attention to the sub-functions in searching for an optimum factored form.

THEOREM 3.5.7 *Let (F, D) be an incompletely specified function. Let g be an optimum factored form of (F_x, D_x). If $F_{\overline{x}} = 0$ and $F_x \not\subseteq D_{\overline{x}}$, then xg is an optimum factored form of (F, D).*

Proof. Let h be an optimum factored form of (F, D). By Theorem 3.5.6, h must contain literal x because $F_{\overline{x}} = 0$ and $F_x \not\subseteq D_{\overline{x}}$. Since $F_{\overline{x}} = 0$, xh_x is also a cover of (F, D) because h_x covers (F_x, D_x). Now, we have

$$\rho(xg) \leq \rho(xh_x) \leq \rho(h).$$

So, xg is an optimum factored form of (F, D). ■

In this theorem, the condition $F_{\overline{x}} = 0$ and $F_x \not\subseteq D_{\overline{x}}$ is equivalent to saying that x is essential. So, under the condition of the theorem, optimum factoring (F, D) is to optimally factoring (F_x, D_x). Even if x is not essential, the optimum factored form of (F_x, D_x) can still help us to find an optimum factored form of (F, D).

THEOREM 3.5.8 *For an incompletely specified function (F, D), let g be an optimum factored form of (F_x, D_x). If $F_{\overline{x}} = 0$, then xg is a factored form of (F, D). Furthermore, if (F, D) has an optimum factored form that depends on x then xg is also an optimum factored form.*

Proof. xg covers (F, D) because $F_{\overline{x}} = 0$. Let h be an optimum factored form of (F, D). Then, h_x covers (F_x, D_x). Because g is an optimum factored form of (F_x, D_x), $\rho(g) \leq \rho(h_x)$. Since h may not contain x, we have $\rho(xh_x) \leq 1 + \rho(h)$. Combining both inequalities, we have

$$\rho(xg) \leq \rho(xh_x) \leq 1 + \rho(h).$$

If h depends on x, then $\rho(h_x) < \rho(h)$. From the above, we have

$$\rho(xg) \leq \rho(xh_x) < 1 + \rho(h)$$

which implies

$$\rho(xg) \leq \rho(h).$$

■

The following Theorem shows exactly how a variable x appears in an optimum factored form if certain conditions are true.

THEOREM 3.5.9 *If an optimum factored form h of (F, D) depends on x, and $F_{\overline{x}} = 0$, then x must appear exactly once as literal x in h, i.e. any optimum factored form must be unate in x.*

Proof. Let g be an optimum factored form of (F_x, D_x). Variable x must appear once because otherwise $\rho(h_x) < \rho(h) - 1$, which implies

$$\rho(xg) \leq \rho(xh_x) < \rho(h).$$

Since xg covers (F, D) because $F_{\overline{x}} = 0$, this contradicts the optimality of h.

Variable x must appear as literal x, not $\overline{x}$. Assume that h contains a literal $\overline{x}$. By previous argument, h cannot contain literal x. So, h is negative unate in x. So, $h = h_x + \overline{x}h_{\overline{x}}$. Since $F_{\overline{x}} = 0$ and h_x covers (F_x, D_x), then h_x covers (F, D). Since $\rho(h_x) < \rho(h)$, this contradicts the optimality of h. So, the original assumption is wrong and h does not contain literal $\overline{x}$. ■

In some cases, an optimum factored form of (F, D) is also an optimum factored form of either (F_x, D_x) or $(F_{\overline{x}}, D_{\overline{x}})$.

THEOREM 3.5.10 *Let (F, D) be an incompletely specified function. Let g be an optimum factored form of (F_x, D_x). If g covers $(F_{\overline{x}}, D_{\overline{x}})$ also, then g is an optimum factored form of (F, D).*

Proof. Let h be any optimum factored form of (F, D). Then, h_x is a cover of (F_x, D_x). Because g is an optimum factored form of (F_x, D_x), we immediately have

$$\rho(g) \leq \rho(h_x) \leq \rho(h).$$

So, g must be an optimum factored form of (F, D). ■

These theorems suggest the following procedure which reduces the optimum factoring of certain functions to optimum factoring of some smaller functions. If (F, D) has a cover of form xg, i.e. $F_{\overline{x}} = 0$, then we can find an optimum factored form of (F, D) by optimally factoring one or two smaller functions. Algorithm $OF1$ handles this case. Notice that $OF1$ is not an optimum factoring algorithm, it is simply one possible step in an optimum factoring procedure OF which is not defined yet.

$OF1(F,D,x)$ /* (F,D) has a cover xg, i.e. $F_{\overline{x}} = 0$ */

1. If $F_x \not\subseteq D_{\overline{x}}$, return $xOF(F_x, D_x)$.
2. If $F_x + D_x \subseteq D_{\overline{x}}$, then return $OF(F_x, D_x)$.
3. $p = OF(F_x, D_x)$.
4. If $p \subseteq D_{\overline{x}}$, return p.
5. $q = OF(F_x, D_x D_{\overline{x}})$.
6. If $\rho(q) = \rho(p)$, return q.
7. return xp.

In $OF1$, if the condition at line 1 is true, then by Theorem 3.5.7 $xOF(F_x, D_x)$ is an optimum factored form of (F,D). If the condition at line 2 is true, then any cover of (F_x, D_x) is a cover of $(F_{\overline{x}}, D_{\overline{x}})$ since $F_{\overline{x}} = 0$. By Theorem 3.5.10, an optimum factored form of (F_x, D_x) is an optimum factored form of (F,D). Otherwise, an optimum factored form of $(F_{\overline{x}}, D_{\overline{x}})$ is found. If the condition at line 4 is true, then by Theorem 3.5.10 again, p is an optimum factored form of (F,D). Next, let q be an optimum factored form of $(F_x, D_x D_{\overline{x}})$. Since $D_x D_{\overline{x}} \subseteq D_x$, q is a cover of (F_x, D_x). By now, we know that the condition at line 1 is false, i.e. $F_x \subseteq D_{\overline{x}}$. So, q is also a cover of $(F_{\overline{x}}, D_{\overline{x}})$. By Theorem 3.5.10, if q has the same number of literals as p does, then q is an optimum factored form of (F,D). Otherwise, the optimum factored form of (F,D) is at least one literal more than p and we've already got one which is xp (notice that by now $\rho(p) < \rho(q)$.

Using duality, the above procedure can be applied to (F,D) having a cover of form $x + g$.

$OF2(F,D,x)$

1. $h = OF1(\overline{F+D}, D, \overline{x})$.
2. return $\overline{h}$.

The following set of propositions provide theoretical support for various steps of procedure $OF1$.

PROPOSITION 3.5.11 *If (F, D) has a cover xg and x is essential, then $xOF(F_x, D_x)$ is an $OF(F, D)$.*

Proof. Because xg is a cover of (F, D), $F_{\overline{x}} = 0$. By Theorem 3.5.7, when x is essential, $xOF(F_x, D_x)$ is an optimum factored form of (F, D). ∎

PROPOSITION 3.5.12 *If (F, D) has a cover cg where c is a cube and all variables in c are essential, then $cOF(F_c, D_c)$ is an $OF(F, D)$.*

Proof. Induction on the number of variables in c and previous proposition. ∎

The following Lemma is the principle of duality applied to optimum factored forms of incompletely specified functions.

LEMMA 3.5.13

$$\overline{OF(F, D)} = OF(R, D)$$

Proof. The complement of a cover of (F, D) is a cover of (R, D). By DeMorgan's Law, complementing a factored form does not change the number of literals. So, let p and q be the optimum factored forms of (F, D) and (R, D) respectively. If $\rho(p) > \rho(q)$, then $\overline{q}$ would be a better factored form of (F, D). So, $\rho(p) = \rho(q)$. ∎

PROPOSITION 3.5.14 *If (F, D) has a cover $x + g$ and x is essential, then $x + OF(F_{\overline{x}}, D_{\overline{x}})$ is an $OF(F, D)$.*

Proof. (F, D) has a cover $x + g$ implies (R, D) has a cover $\overline{x}\,\overline{g}$. By previous proposition, $\overline{x}OF(R_{\overline{x}}, D_{\overline{x}})$ is an $OF(R, D)$. By DeMorgan's Law, $x + \overline{OF(R_{\overline{x}}, D_{\overline{x}})}$ is an $OF(F, D)$. By previous Lemma, $x + OF(F_{\overline{x}}, D_{\overline{x}})$ is an $OF(F, D)$. ∎

PROPOSITION 3.5.15 *If (F, D) has a cover $x + y + \ldots + z + g$ and $x, y, \ldots, z$ are essential, then $x + y + \ldots + z + OF(F_{\overline{x}\,\overline{y}\,\ldots\overline{z}}, D_{\overline{x}\,\overline{y}\,\ldots\overline{z}})$ is an $OF(F, D)$.*

Proof. Induction on number of variables in $x, y, \ldots, z$ using previous proposition. ∎

3.5.3 Redundant Variables

Finding out redundant variables can also restrict the search space when deriving optimum factored forms.

DEFINITION 3.5.16 *For an incompletely specified function (F, D), if no optimum factored form of (F, D) can contain variable x, then x is said to be redundant.*

LEMMA 3.5.17 *If f covers (F, D), then f_c covers (F_c, D_c) for any cube c.*

Proof. Easy induction on number of variables in c. ∎

THEOREM 3.5.18 *Let $(F, D+E)$ be an incompletely specified function such that $sup(F+D) \cap sup(E) = 0$. Then, variables in $sup(E)$ are redundant, i.e., $OF(F, D+E) = OF(F, D)$.*

Proof. Let f be an optimum factored form of $(F, D+E)$. Assume $E \neq 1$, and let c be a minterm of $\overline{E}$. By previous Lemma, f_c covers $(F_c, D_c + E_c)$. But, $F_c = F$, $D_c = D$, and $E_c = 0$. So, f_c also covers (F, D). Since f is optimum, we must have $\rho(f_c) = \rho(f)$, which implies that $sup(f) \cap sup(c) = sup(f) \cap sup(E) = 0$. So, $OF(F, D+E) = OF(F, D)$. Notice that this does not imply that no factored form would contain some variables in E. ∎

The above Theorem can be interpreted by Figure 3.6. If F, D, and E are represented as sum-of-product expressions in a positional cube notation [16] and the matrix has the structure indicated in the figure, then the set of cubes in E can be removed from the don't-care set without affecting the final result.

<table>
<tr><td>F</td><td rowspan="2">2</td></tr>
<tr><td>D</td></tr>
<tr><td>2</td><td>E</td></tr>
</table>

Figure 3.6: Certain structure of an incompletely specified function

A natural question to ask at this point is: is E unique? If yes, how to find it? The following Theorem answers the questions.

THEOREM 3.5.19 *Let f be a Boolean function, if there is a sum-of-product representation whose cube matrix has the structure as indicated in Figure 3.7 where the support of A and B are X and Y respectively, then no prime p contains variables from both X and Y.*

f		
	A	2
	2	B
	X	Y

Figure 3.7: Special structure of the cube matrix of f

Proof. Let p be a prime of f. Suppose p does contain variable from both X and Y. Let $p = p_x p_y$ where p_x and p_y are parts of p containing only variables from X and Y respectively. Taking cofactors of f and p with respect to p_y gives us $f_{p_y} = A$ and $p_{p_y} = p_x$ from the structure of the matrix. So, $p \subseteq f$ implies $p_{p_y} \subseteq f_{p_y}$, which in turn implies $p_x \subseteq A$. This contradicts to the assumption that p is a prime because all literals of p_y can be left off from p. So, p must contain variables either entirely from X or entirely from Y. ■

Not only did the Theorem show the uniqueness of the partition (if we make all cubes prime), it also indicated a procedure for obtaining it. Given an incompletely specified function (F, D), the following procedure finds the unique block partitioning of the matrix, and removes the redundant blocks.

```
PARTITION(F,D)
  Expand all cubes of F + D to primes.
  M = cube matrix of all primes.
  P = partitions of M.
  for each block B ∈ P {
    if B ⊆ D {
      remove B from M.
  } }
```

First, the algorithm expands all cubes to primes, builds a cube matrix M, then partitions M into blocks of disjoint supports. There are many existing algorithms for finding the partitions [16]. Finally, for each block of the partition, if it is entirely contained in the don't-care set, it is redundant and is hence removed by Theorem 3.5.19.

3.6 Experiments and Results

Several heuristic factoring algorithms have been implemented in the multi-level logic optimization MIS. They are: quick factoring (QF), good factoring (GF), boolean factoring (BF), quick factoring using complement (QFC), good factoring using complement (GFC), and Boolean factoring using complement (BFC). These algorithms, particularly quick factoring, have become important components in the optimization system. Quick factoring has been very effective and efficient as shown in the results table, and is used continuously throughout the optimization process to provide estimate for area and delay of Boolean networks. The objective of this section is to show the relative effectiveness and efficiencies of all the proposed algorithms by running them on a large set of logic functions from all available benchmark circuits.

The examples are taken from MIS benchmark set consisting of one hundred circuits. They are either from the MCNC logic synthesis workshop benchmark set, or actual circuits from industry. For the purpose of experiment and comparison, we want logic functions with reasonably large sum-of-product representations. In addition, we want those functions to have reasonably interesting factored forms, (e.g. $ab + cd + ef + gh + \ldots$ is hardly interesting to us because it has no factoring except the sum-of-product form itself). Therefore, a function is chosen if

1. it has at least 20 sum-of-product literals.
2. the number of literals in its factored form obtained by QF is at least 20% less than the sum-of-product literals.

This process is done indiscriminately for all functions in all the benchmark circuits. The results are 153 logic functions.

In addition to these real examples, there is another set of contrived examples which are obtained from good factored forms, usually Boolean, in order to test various specific features of the factoring algorithms.

The first experiment is to show the relative efficiencies and effectiveness of all the algorithms. Due to the size of the example set, it is not possible to present individual data on all the functions. Instead, the summary is presented in Table 3.1. There are total of 153 logic functions used. Each row in the table is the result of a particular algorithm. The abbreviations used are interpretated as following:

SOP Total number of literals in sum-of-product forms.

Algorithm	SOP	FCT	LB	time	best	min
QF	4059	1686	1403	14.1	136	79
GF	4059	1683	1403	28.6	138	80
BF	4059	1681	1403	182.4	140	79
QFC	4059	1725	1403	12.9	130	81
GFC	4059	1716	1403	21.9	132	81
BFC	4059	1688	1403	137.3	137	81

Table 3.1: Summary of factoring algorithms on 153 functions

FCT Total number of literals in factored forms.

LB Lower bound on the number of literals in factored forms.

time Seconds on VAX-8650.

best Number of best results obtained by an algorithm (best amount all algorithms tested).

min Number of minimum results obtained by an algorithm (equal to lower bound).

The results show that QF and GF are much faster than BF. Even though the overall results of *BF* is better, the amount of improvement is insignificant. Factoring on the complement in general yields worse results. However, the difference is much greater for QF and GF than BF, which shows that QF and GF depend more on the initial phase of functions than BF does. More than 50% of the time one of the algorithms obtained the optimum solutions, as the last column shows. To examine more closely the difference between the algorithms, we remove all the examples for which all algorithms generated the same results, and present data for each function individually in Table 3.2. As we can see from the table, in general, factoring a function or its complement gives very close results. However, there are few exceptions, e.g. $g8$, $k9$, $l9$, and $l10$. On these examples, *BF* is able to do much better than both QF and GF. These examples show that there are functions which have good algebraic factored forms if factoring is performed on a correct phase. However, there are also functions which do not have good algebraic factored forms on either phases. A example of such functions is $y10$ which QF and GF were able to obtain the same results on both phases but neither is as good as results of BF.

Next, we do the same experiment on the set of contrived examples. The best

ex	SOP	QF		QFC		GF		GFC		BF		BFC	
	lit	lit	time	lit	time	lit	time	lit	time	lit	time	lit	time
g8	20	15	0.09	21	0.21	15	0.19	18	0.27	15	2.78	17	1.39
h8	24	15	0.10	14	0.11	15	0.24	14	0.21	14	1.59	15	1.44
k9	22	14	0.10	20	0.20	14	0.26	21	0.46	14	1.78	19	2.57
l9	22	14	0.12	20	0.19	14	0.25	20	0.34	14	1.57	18	2.17
m9	40	10	0.10	9	0.06	9	0.26	9	0.08	9	1.48	9	0.61
p9	21	14	0.09	14	0.12	14	0.18	13	0.17	15	0.97	13	0.88
q9	25	16	0.10	15	0.09	17	0.23	15	0.18	16	1.34	15	1.15
r9	21	15	0.08	13	0.07	15	0.19	13	0.15	13	0.73	13	1.50
u9	20	14	0.07	15	0.10	14	0.18	15	0.20	14	0.91	15	1.30
w9	22	15	0.12	15	0.15	15	0.22	15	0.29	15	1.44	17	1.00
a10	20	11	0.11	12	0.04	11	0.21	12	0.07	11	1.25	12	0.45
b10	25	14	0.14	15	0.07	14	0.30	15	0.14	14	1.72	15	1.13
c10	25	14	0.13	15	0.07	14	0.30	15	0.13	14	1.69	17	1.12
d10	25	15	0.11	16	0.10	15	0.24	16	0.19	15	1.53	14	1.44
e10	20	11	0.11	12	0.04	11	0.22	12	0.07	11	1.22	12	0.45
f10	20	11	0.11	12	0.04	11	0.22	12	0.07	11	1.25	12	0.45
g10	25	14	0.14	15	0.08	14	0.30	15	0.13	14	1.76	15	1.23
h10	25	15	0.12	16	0.11	15	0.26	16	0.21	15	1.50	14	1.44
k10	25	15	0.11	16	0.11	15	0.25	16	0.20	15	1.49	14	1.46
l10	24	16	0.09	26	0.20	16	0.19	26	0.55	16	1.62	18	1.82
m10	22	17	0.09	18	0.16	16	0.23	18	0.31	18	1.49	17	1.32
n10	20	14	0.09	14	0.10	14	0.29	12	0.20	14	1.08	12	1.21
y10	56	16	0.13	16	0.15	15	0.58	15	0.59	13	2.09	13	2.06
i11	38	10	0.14	8	0.05	8	0.25	8	0.06	10	1.88	8	0.61

Table 3.2: Comparison of various factoring algorithms

factoring of each one is listed below.

$$O = a'b'c'd'e'f' + abcdef + (e'f' + c'd' + a'b')(ef + cd + ab)$$
$$P = (e'f' + c'd' + a'b')(ef + cd + ab)$$
$$Q = (ag + e + d)(af + c + b)$$
$$R = (a + b)(c + d)(e + f) + (a' + b')(c' + d')(e' + f')$$
$$S = a'((e' + c')(cf'gh' + d) + (c'h + d'e)(f' + c))$$

As we can see that all of them have good Boolean factored forms. As we see from the following tables, BF did perform a lot better than both QF and GF on these examples.

Algorithm	SOP	FCT	LB	time	best	min
QF	151	100	54	0.3	1	0
GF	151	95	54	0.6	1	0
BF	151	74	54	2.7	4	1
QFC	151	100	54	0.4	2	0
GFC	151	96	54	0.9	2	0
BFC	151	87	54	4.0	1	1

Table 3.3: Comparison of factoring algorithms on contrived examples

ex	SOP	QF		QFC		GF		GFC		BF		BFC	
	lit	lit	time	lit	time	lit	time	lit	time	lit	time	lit	time
P	36	28	0.04	34	0.16	28	0.10	34	0.47	24	0.47	33	1.81
Q	24	20	0.03	26	0.11	20	0.07	22	0.18	12	0.38	16	0.90
O	23	12	0.05	8	0.02	11	0.08	8	0.05	8	0.24	11	0.21
R	44	26	0.11	18	0.05	22	0.18	18	0.09	16	0.89	12	0.38
S	24	14	0.04	14	0.04	14	0.12	14	0.08	14	0.70	15	0.59

Table 3.4: Comparison of factoring algorithms on contrived examples

Notice that a fast algorithm that does well is to do QF, QFC and take the best provided that the complement can be obtained efficiently.

3.7 Future work

In a multi-level logic network, every function has certain don't-care conditions. The existing heuristic in factoring an incompletely specified function is to minimize the sum-of-products form for the function and then factor the optimized sum-of-products form.

But, a minimum-cube sum-of-products form cover is in general not related to a minimum-literal factored form cover. So, there is a need to factor an incompletely specified function directly (problem DCF).

As shown in Section 3.3, many problems encountered in manipulating factored forms are or can be reduced to simplifying a factored form using a don't-care set also in the factored form. Solutions to this problem are essential to manipulating functions entirely on factored forms.

These two problems are related. To factor an incompletely specified function, one can use existing algorithms to factor the on-set and don't-care set separately and then simplify the result using FSIM. On the other hand, to simplify a factored form with the dont'-care set also in factored form, one can first collapse the two factored forms into sum-of-products forms and then use DCF to obtain a new factored form.

Chapter 4

Boolean Simplification

4.1 Introduction

Two-level minimization is a much more developed science than multi-level minimization, and very efficient algorithms exist for finding minimal two-level representations of Boolean functions. Two-level logic minimization plays an important role in optimal multi-level logic synthesis. Recall that a Boolean network is simply a collection of inter-connected logic functions and each one can be represented as a sum-of-products expression or a factored expression. The sum-of-products form of each function in a Boolean network can be minimized, i.e. replaced by an equivalent but smaller sum-of-products form, using two-level logic minimization algorithms. Two-level minimization can be made more powerful in this context by providing the minimizer with various don't-care sets derived from the immediate environment of a function. Just how much of the don't-care sets to derive depends on how thorough and how fast this minimization process is expected to be.

The ultimate goal of simplifying a function in a Boolean network is to replace it with another equivalent, but minimal, function which has the fewest number of literals in the factored form. At the moment, there is no algorithm for minimizing a function with this objective. Instead, two-level minimization algorithms are used to minimize the number of literals in sum-of-products form of a function as an approximation to the number of literals in the factored form.

As we will see later on, don't-care sets of a function can and do become very large in the sum-of-products representation. Furthermore, the dimension of the Boolean space in which a function and its don't-care set are defined can also become very large.

Together, they limit seriously the effectiveness of our simplification procedure. The emphasis of this chapter is to study ways of reducing the size of a don't-care set by finding and removing the useless or "almost" useless cubes from the don't-care set. Doing so not only reduces the size of a don't-care set in the sum-of-products representation, but also restricts the dimension of the Boolean space. Experiments in MIS have shown that this approach improves significantly the performance of the simplification process, but at the potential expense of small loss in the quality of results.

Section 2 gives basic definitions. Section 3 outlines a simplification procedure and its variations. Section 4 studies two ways of restricting the size of a don't-care set, one by removing certain cubes from the don't-care set, and the other by generating selectively the don't-care set in the first place. Section 5 suggests an alternative way of reducing the size of a don't-care set by representing a certain part of it in the complemented form. The last two sections present experimental results and discussions of some open problems.

4.2 Basic Definitions

Let f be a logic function. the character f is often used to denote certain expression of f (e.g., $f = a+b'c$), or the literal corresponding to the function f which is used to compose other functions (e.g., $g = f'h + fh'$). To avoid confusion, we explicitly define f_l to be the literal corresponding to function f, and f_e to be any expression of function f. The default convention for interpreting the meaning of character f when f_l and f_e are not used is: f is a literal if it appears on the right-hand side of an equation and is an expression otherwise.

The operator $\oplus$ denotes the operation exclusive-OR, i.e., $a \oplus b = a'b + ab'$. The operator $\overline{\oplus}$ denotes operation exclusive-NOR, i.e, $a \overline{\oplus} b = ab + a'b'$. Notice that $\overline{\oplus}$ is equivalent to $\equiv$, the identity relation.

Some operations described in this chapter modify a network by deleting certain nodes. Let D be a set of nodes in a Boolean network η. $\eta - D$ represents a new network obtained by deleting all nodes of D from η and considering all edges from D to $\eta - D$ as additional primary inputs of $\eta - D$ and all edges from $\eta - D$ to D as additional primary outputs of $\eta - D$.

Fan-in Don't-care Set

For any completely specified function f, the value of its literal f_l is always equal to the value of its expression f_e. So, the new function represented by the expression $f_l \oplus f_e$ represents combination of values of variables f_l and $sup(f)$ which can never occur. This new function is denoted by DI_f, i.e.,

$$DI_f = f_l \oplus f_e$$

and is called the ***intermediate don't-care set*** posed by function f for the reason that it can never be evaluated to 1 and therefore can be used as the don't-care set for other functions. An example of f and its corresponding DI_f is

$$\begin{aligned} f &= a + b \\ DI_f &= f'(a+b) + f(a+b)' = f'a + f'b + fa'b'. \end{aligned}$$

Since function f can have, in general, many different expressions, it remains to show that DI_f is well-defined, i.e., its value is independent of the expression f_e used in $f_l \oplus f_e$.

PROPOSITION 4.2.1 *Let f_e^1 and f_e^2 be two different expressions of function f. Then, $f_l \oplus f_e^1 = f_l \oplus f_e^2$.*

Proof. Since f_e^1 and f_e^2 both are expressions of f, we have

$$(f_e^1 \overline{\oplus} f_e^2) = 1.$$

So,

$$\begin{aligned} f_l \oplus f_e^1 &= (f_l \oplus f_e^1)(f_e^1 \overline{\oplus} f_e^2) \\ &= (f_l \overline{f_e^1} + \overline{f_l} f_e^1)(f_e^1 f_e^2 + \overline{f_e^1}\,\overline{f_e^2}) \\ &= \overline{f_l} f_e^1 f_e^2 + f_l \overline{f_e^1}\,\overline{f_e^2} \\ f_l \oplus f_e^2 &= (f_l \oplus f_e^2)(f_e^1 \overline{\oplus} f_e^2) \\ &= (f_l \overline{f_e^2} + \overline{f_l} f_e^2)(f_e^1 f_e^2 + \overline{f_e^1}\,\overline{f_e^2}) \\ &= \overline{f_l} f_e^1 f_e^2 + f_l \overline{f_e^1}\,\overline{f_e^2} \end{aligned}$$

Therefore, $f_l \oplus f_e^1 = f_l \oplus f_e^2$. ∎

In a given Boolean network, each function poses an intermediate don't-care set. Together, they form the intermediate don't-care set for the entire network. Let η be a

Boolean network, we define

$$DI(\eta) = \sum_{f \in FUNCTION(\eta)} DI_f.$$

When there is only one Boolean network involved, we simply use DI instead of $DI(\eta)$ for brevity.

Fan-out Don't-care Set

Let η be a single output Boolean network. Let p be the primary output and x be an intermediate function in η. Define p_x to be the network obtained by replacing everywhere the literal x with constant 1 and literal x' with constant 0, i.e., p_x is p when x is forced to be 1. Similarly, p_x is the network obtained by replacing everywhere the literal x with constant 0 and literal x' with constant 1. Define DO_x^p as

$$DO_x^p = p_x \overline{\oplus} p_{x'}.$$

Thus, DO_x^p is the condition under which p_x and $p_{x'}$ are both 1 or both 0, i.e., the value of p does not depend on the value of x. For this reason, DO_x^p is called *fan-out* don't-care set of x. Notice that if p is independent of x, then $p_x = p_{x'}$ which implies $DO_x^p = 1$.

To generalize this concept to a multiple-output Boolean network, define

$$DO_x(\eta) = \prod_{p \in PO(\eta)} DO_x^p.$$

Notice that $PO(\eta) \cap TFO(x)$ is the subset of primary output which is contained in the transitive fanout of x, $TFO(x)$. Clearly, DO_x is the condition under which no primary output of η can be affected by the value of x. Therefore, DO_x is called the *fanout don't-care set* of x.

Internal Don't-care Set

The intermediate and fan-out don't-care sets together form a don't-care set for a function in a Boolean network. For a Boolean network η and a function $f \in FUNCTION(\eta)$, define

$$D_f(\eta) = DI(\eta) + DO_f(\eta),$$

or simply

$$D_f = DI + DO_f.$$

D_f is called *internal don't-care set* of f.

4.3 Simplification procedure

This section outlines and studies a simplification procedure. It begins with an example as an introduction to the concept of internal don't-care sets, Boolean substitution, simultaneous Boolean substitution, and simplification. A general simplification procedure is presented along with several specific variations which have been implemented and tested in MIS.

4.3.1 An Example

Let a Boolean network contain two functions:

$$\begin{aligned} f &= a + bc + bd \\ g &= a + c + d. \end{aligned}$$

It is not immediately obvious but can be easily verified that f can be re-expressed as $f = a+bg$, because when $a+bg$ is multiplied out, the term ba is contained in a and therefore can be removed. However, to discover this simplification is a much harder problem, and is described step by step next.

Function $f = a+bc+bd$ can be re-expressed as $f = a+b(c+d)g+b(c+d)g'$ for any g. Because the particular g we are using is equal to $a+c+d$, $(a+c+d)g'$ is always 0 which implies $(c+d)g'$ is always 0. So the last term can be dropped and now $f = a + b(c+d)g$. For the same reason, $(a+c+d)'g$ is always 0, so the term $ba'c'd'g$ can be added into the expression. We can also add term $bac'd'g$ into the expression because it is covered by a. Together, these cubes can then be merged into $a+bg$. The entire process is summarized by

$$\begin{aligned} f &= a + b(c+d) & & \\ &= a + b(c+d)g + b(c+d)g' & & x = xy + xy' \\ &= a + b(c+d)g & & (c+d)g' = 0 \\ &= a + b(c+d)g + ba'c'd'g & & a'c'd'g = 0 \\ &= a + b(c+d)g + ba'c'd'g + abc'd'g & & abc'd'g \subseteq a \\ &= a + b(c+d)g + bc'd'g & & xy + xy' = x \\ &= a + b(c+d)g + b(c+d)'g & & c'd' = (c+d)' \\ &= a + bg & & xy + xy' = x \end{aligned}$$

4.3.2 Boolean Substitution

The essential fact used in the previous example to achieve the simplification is that g is always equal to $a+c+d$. Because of this, the condition $(a+c+d)'g$ can never be evaluated to 1 and nor can $(a+c+d)g'$. Together, the condition $D_g = (a+c+d) \oplus g$ can never be evaluated to 1 and therefore can be used as the don't-care set for simplifying any f. Indeed, if ESPRESSO were used to minimize f with D_g as the don't-care set, the result would have been $f = a + bg$ also. This process can be viewed as a Boolean substitution of g into f. It is made of two related problems.

PROBLEM 1: (question of existence) Given logic functions f and g, are there any p, q, and r where at least one of p and g is non-zero such that $f = pg + qg' + r$?

THEOREM 4.3.1 *Let (F, D) be an incompletely specified function. Let $f = ph + r$ be a cover of (F, D) and g be any other function. If $g \oplus h \subseteq D$, then $\tilde{f} = pg + r$ is also a cover of (F, D).*

Proof. Let $f = ph + r$ and $\tilde{f} = pg + r$. We need to prove $\tilde{f} \oplus f \subseteq D$.

$$\begin{aligned}
\tilde{f} \oplus f &= (pg + r) \oplus (ph + r) \\
&= (p'r' + g'r')(ph + r) + (pg + r)(h'r' + p'r') \\
&= g'r'ph + pgh'r' \\
&= r'p(g'h + gh') \\
&= r'p(g \oplus h) \\
&\subseteq g \oplus h \\
&\subseteq D
\end{aligned}$$

■

COROLLARY 4.3.2 *$pg_e + r$ is a cover of $(f, g_l \oplus g_e)$ if and only if $pg_l + r$ is a cover of $(f, g_l \oplus g_e)$.*

COROLLARY 4.3.3 *$pg_e' + r$ is a cover of $(f, g_l' \oplus g_e')$ if and only if $pg_l' + r$ is a cover of $(f, g_l' \oplus g_e')$.*

Therefore, the answer to **PROBLEM 1** is: p, q and r exist if and only, by forcing variable g to be used either as literal g_l or literal g_l', we can obtain a cover of $(f, g_l \oplus g_e)$. [18] provides an algorithm for doing so.

PROBLEM 2: (optimization) Given logic functions f and g, find p, q, and r such that $f = pg_l + qg_l' + r$ is minimum.

In multi-level logic optimization, the size of a function is measured by the number of literals in its factored form. So, the objective of PROBLEM 2 is to find a factored form of $pg_l + qg_l' + r$ with the minimum number of literals. However, there is no known algorithm for solving this problem. Therefore, p, q and r are minimized for the total number of product terms as an approximation to the total number of factored-form literals. In general, the number of product terms of p, q, and r are not independent of each other. However, since we are only interested in minimizing the total number of product terms, the objective is simplified because of the one-to-one correspondence between the total number of product terms in p, q and r and the number of product terms in the expression $pg_l + qg_l' + r$, i.e.,

$$|p| + |q| + |r| \equiv |pg_l + qg_l' + r|$$

because g_l and g_l' are literals. This objective can be achieved by minimizing the incompletely specified function $(f, g_l \oplus g_e)$ using any two-level minimization algorithm. Notice that if the objective is to minimize the total number of literals in the sum-of-products forms of p, q, and r, then minimizing the sum-of-products literals in $pg_l + qg_l' + r$ would only serve as an approximation because of the following observation: suppose function f has two different covers, i.e $f = pg_l + r$ and $f = \tilde{p}g_l + r$, such that $\tilde{p}$ has fewer literals but more terms. A two-level minimum literal minimizer may not pick $\tilde{p}g_l + r$ because literal g_l appears in too many terms.

Putting everything together, we have the following Boolean substitution algorithm which tries to minimize function f using an existing function g. The fact that g is an existing function is important because the cost of g is not taken into account when minimizing $(f, g_l \oplus g_e)$. The problem of finding a g such that the total cost of $pg + qg' + r$ and g is minimized is a harder problem and is not addressed here.

$BOOL_SUB1(f,g)$
  $DI_g = g_l \oplus g_e$
  $MINIMIZE(f, DI_g)$
  return f

The routine $MINIMIZE$ can be ESPRESSO or any other two-level minimization program. Notice that the goal of $BOOL_SUB1(f,g)$ is to simplify f using an existing g. If function g does not help simplify f, then the literal g_l and g_l' will not appear in the resulting f.

4.3.3 Simultaneous Boolean Substitution

When there are more than two functions in a Boolean network, a function may be simplified by trying to substitute all other functions into it one by one. A new question arises: in which order should the substitution be carried out? Even if we can enumerate all possible orderings, there is still a more fundamental question of whether additional simplification can be made by simultaneously substituting a set of functions into f. The following example answers the questions. We add one more function $h = a + b$ to the previous example and see how we can use both g and h to simplify f.

$$\begin{aligned} f &= a + bc + bd \\ g &= a + c + d \\ h &= a + b \end{aligned}$$

We try all possible orders in which to substitute g and h into f and the results are summarized as:

$$\begin{aligned} BOOL_SUB1(f,g) &\Rightarrow a + bg_l \\ BOOL_SUB1(f,h) &\Rightarrow a + (c+d)h_l \\ BOOL_SUB1(BOOL_SUB1(f,g),h) &\Rightarrow a + bg_l \\ BOOL_SUB1(BOOL_SUB1(f,h),g) &\Rightarrow a + g_l h_l. \end{aligned}$$

If g is substituted into f first, no further simplification can be made by substituting h. If h is substituted into f first, f can be simplified again with g to $a + g_l h_l$. But, $a + g_l h_l$ can be further simplified to $g_l h_l$ because a is contained in $gh = (a+b)(a+c+d)$. The reason this was not discovered by the procedure is because the relationship between a and h_l was not known when substituting g into $a + (c+d)h_l$ and the relationship between a and g_l was not known when substituting h into $a + bg_l$. Had we tried to substitute both g and h into f simultaneously with the don't-care set $(g_l \oplus g_e) + (h_l \oplus h_e)$, f would have been simplifed to $g_l h_l$. Therefore, not only did we achieve more simplification by simultaneously substituting g and h into f, we also obtained the result $(g_l h_l)$ which could not otherwise be obtained by substituting g and h into f individually in any order and we did it with only

one simnplification. So, the Boolean substitution routine has to be generalized to allow this more powerful simplification. Let X be a set of functions. The following procedure simplifies function f by substituting all functions of X into f simultaneously.

$BOOL_SUB2(f,X)$
  $DI_X = \bigcup_{x \in X}(x_l \oplus x_e)$
  $MINIMIZE(f, DI_X)$
  return f

$BOOL_SUB2$ simplifies the function f by substituting a set of functions X into f simultaneously. It is possible that some functions in X are not useful for simplifying f. During the course of simplification, $BOOL_SUB2$ implicitly selects an optimal subset of X which is the set of functions that ends up in the simplified f. By the nature of the minimization process, adding more functions into X can never hurt the simplification result. However, the run time of $BOOL_SUB2$ can be significantly affected if X is constructed in a un-justified manner. This issue will be addressed in the subsequent sections.

To put simultaneous Boolean substitution into the context of multi-level logic optimization, it is the process of simplifying a function using all other existing functions in the network. The don't-care set is just the intermediate don't-care set, DI, defined earlier and the simplification procedure is:

$SIMPLIFY_DI(\eta, f)$
  $DI = DI - \{f_l \oplus f_e\}$
  $MINIMIZE(f, DI(\eta))$

Notice that all the cubes in $f_l \oplus f_e$ have to be removed from DI in order to prevent f to be simplified to f_l because

$$\begin{aligned} f &= f_e \\ &= f_e f_l + f_e \overline{f_l} \\ &= f_e f_l + \overline{f_e} f_l \\ &= f_l \end{aligned}$$

4.3.4 Output don't-care conditions

Simultaneous substitution of a set of existing functions into another function is not the only way a function can be simplified. The following example illustrates this. Let

$$\begin{aligned} x &= af + bc \\ f &= a'd + b'c. \end{aligned}$$

In this example, f is used in function x. Notice that whenever bc is 1, x is 1 regardless of the value of f, so bc can be added to f without effecting the outcome of x. In addition, whenever a is 0, the value of f does not affect the outcome of x also, therefore a' can be added to f without effecting the outcome of x. Together, f can be simplified because

$$\begin{aligned} f &= a'd + b'c \\ &= a'd + b'c + bc + a' \\ &= a' + c. \end{aligned}$$

The condition $bc + a'$ is treated here as the don't-care set of f because any change of f using this don't-cares does not affect the outcome of x. It is trivial to verify that $bc + a'$ is equal to $x_f \overline{\oplus} x_{f'}$ and is, as we defined earlier, the *fan-out* don't-care set of f.

Ideally, one would like to simplify a function f using its entire fan-out don't-care set DO_f. However, there is one problem with this process. For an arbitrary function f in a Boolean network, if all fan-outs of f are primary outputs, DO_f can easily be computed because we know the *operation* of cofactoring an expression with respect to a literal. If some fan-outs of f are not primary outputs, the procedure for computing DO_f is not defined. For this reason, a subset of DO_f is computed and used to simplify f. This subset is defined as

$$DO1_f = \prod_{x \in FO(f)} x_f \overline{\oplus} x_{f'}$$

and is called *one-level* fan-out don't-care set of f. Notice that f is a variable in x because x is an immediate fan-out of f, therefore, the operation of obtaining x_f and $x_{f'}$ is defined. To see that $DO1_f$ is a subset of DO_f, we need

THEOREM 4.3.4 $DO1_f \subseteq DO_f$.

Proof. Trivial. Since $DO1$ is the condition under which the value of f does affect the values of the immediate fanouts of f, so it does not affect the primary outputs. Thus, $DO1 \subseteq DO$. ∎

The theorem is really a simple statement: if the value of a function does not affect the values of its fan-outs, it does not affect the values of the primary outputs.

Now, we have restricted the entire fanout don't-care set of a function to a subset which can actually be computed in a straight forward manner. The simplification procedure using this don't-care set is called $SIMPLIFY_DO1$.

$SIMPLIFY_DO1(f)$
  $MINIMIZE(f, DO1_f)$

4.3.5 Simplification

By now, it has become fairly clear that to simplify a function f is to minimize the function with an appropriately derived don't-care set. The don't-care set may consist of intermediate don't-cares (DI), a subset ($DO1_f$) of the fanout don't-cares as well as user specified don't-cares DC (also called external don't-cares). This procedure of simplification is called $SIMPLIFY$ and is simply $MINIMIZE(f, DI + DO1_f + DC)$. To obtain desired run-time versus quality tradeoffs, two parameters are added to the simplification procedure which result in the following:

$SIMPLIFY(f, MINIMIZE, DC_GEN)$
  $DC = DC_GEN(f)$
  $MINIMIZE(f, DC)$
  return f

The first parameter, $MINIMIZE$, specifies a two-level minimization program to be used. The second parameter DC_GEN specifies the type of don't-cares, which may be any combination of DI, $DO1_f$, and DC or their subsets. By selecting different $MINIMIZE$ and varying DC_GEN, one can achieve desired tradeoffs between the performance and the quality of results. In general, $ESPRESSO$ can be used as the minimization program and has been shown in MIS to produce high quality results except for very few cases. [36] provides another minimization routine which is a variation of ESPRESSO. The

second parameter is intended for controlling the size of the don't-care set to achieve desired run-time efficiency. It will be shown in the results section of this chapter that the don't-care set can get very large and there is currently no effective procedure that can make use of such a large don't-care set efficiently. In the next section, emphasis will be placed on how to find the useful or "most" useful part of a don't-care set.

4.4 Filtering Don't-care Sets

An ideal procedure for simplifying a given logic function f in a Boolean network is to generate first the entire don't-care set, D, including fanin don't-care set DI, the one-level fanout don't-care set $DO1_f$ [1], and user specified don't-cares, and then minimize the incompletely specified function (f, D). However, for most practical circuits this approach can hardly be carried through for two reasons. First, because the simplification is finally performed by a two-level minimizer, the don't-care set needs to be represented in a sum-of-products form. As will be shown in the result section of this chapter, the don't-care set is often too large, so large that it exceeds the capacity of most existing computer systems. Even when the don't-care set can be computed, most of the existing two-level minimization programs still take unacceptable amounts of time because the large amount of don't-care information.

The first solution is to generate the don't-care set in a selective fashion. By examining the circuit topology (connections) around the function to be simplified, one can leave out parts of the don't-care set derived from the functions which are "far away" and are unlikely to be used in the simplification. One of the theorems presented later on suggests a way to find out, based on circuit topology, those functions which have absolutely no use in the simplification.

Since a function in a Boolean network can be arbitrarily complex, parts of the circuit topology are actually hidden inside the function. So, the don't-care set generated by looking at any special circuit topology may still contain un-necessary components (cubes). Therefore, one needs to look into the sum-of-products representation of the don't-care set and remove the useless or "almost" useless cubes to further reduce its size. This process is called *filtering* because it filters out the useless part of the don't-care set. A filter is *exact* if it finds out the cubes which, when removed from the don't-care set, do not compromise

[1] The entire fan-out don't-care set DO_f should and would be used if it can be computed efficiently.

the quality of the final result. A filter is *heuristic* if it finds out the "unlikely to be useful" part of the don't-care set.

In the remaining of this section, the emphasis is on understanding the two-level minimization algorithms and from that understanding deriving conditions which can be used to filter out the "useless" cubes in the don't-care sets. An exact filter is first derived based on a special structure in the matrix representation of the don't-care set. Then, its implication on the circuit topology is studied which can then be used to guide the don't-care set generation in the first place. In addition, a set of approximate filters are designed to further reduce the size of the don't-care set.

4.4.1 An exact filter

For an incompletely specified function where both the on-set and the don't-care set are represented in the sum-of-products forms, deleting a cube of the don't-care set may result in excluding certain variables from consideration in the minimization process, and consequently speeds up the simplification process. However, doing so may also limit the solution space and compromise the results.

DEFINITION 4.4.1 *Let (F, D) be an incompletely specified function. A variable x is said to be in-essential if there is an optimum cover of (F, D) which is independent of x. A cube $c \in D$ is in-essential if any optimum cover of $(F, D - \{c\})$ is an optimum cover of (F, D).*

So, if a cube c is in-essential, one can simply delete it from the don't-care set without worrying about losing any optimality. The next theorem helps remove some in-essential cubes and, as a by-product, some in-essential variables for a given incompletely specified function if the function has a special structure in its sum-of-products representation.

LEMMA 4.4.2 *Let C be the sum of two orthogonal functions, i.e. $C = A + B$ such that $sup(A) \cap sup(B) = \phi$. Let $p_A p_B$ be an implicant of C such that $sup(p_A) \subseteq sup(A)$ and $sup(p_B) \subseteq sup(B)$. Then, either $p_A \subseteq A$ or $p_B \subseteq B$.*

Proof. If $p_B \not\subseteq B$, then we must have $p_A \subseteq A$ because there must be a minterm of B' which is in the form $p_B q$ such that

$$\begin{aligned} p_A &= (p_A p_B)_{p_B q} && sup(p_A) \perp sup(p_b q) \\ &\subseteq C_{p_B q} && p_A p_B \subseteq C \\ &= A_{p_B q} + B_{p_B q} && C = A + B \\ &= A + 0 && p_B q \subseteq B' \\ &= A && \end{aligned}$$

■

THEOREM 4.4.3 *Let $(F, D+E)$ be an incompletely specified function and $|(sup(F) \cup sup(D)) \cap sup(E)| = 1$, then ALL cubes of E are in-essential and all variables in $sup(E) - sup(F+D)$ are in-essential.*

Proof. The structure of the function is illustrated in Figure 4.1. Let $sup(F+D)$ be

on-set | F | 2
don't-care set | D | 2 · E
X y Z

Figure 4.1: Special structure of an incompletely specified function $(F, D+E)$

$X \cup \{y\}$, and $sup(E)$ be $Z \cup \{y\}$. All we need to prove is: for any prime p of $(F, D+E)$ such that $sup(p) \cap Z \neq \phi$, there is another prime q such that $sup(q) \cap Z = \phi$ and any minterm of F covered by p is also covered by q.

Let p be a prime of $(F, D+E)$ containing some variables of Z. Let $p = p_x p_y p_z$ where p_x, p_y, and p_z contain literals of p from X, Y, and Z respectively.

Since $p_x p_z = p_{p_y} \subseteq F_{p_y} + D_{p_y} + E_{p_y}$ and $sup(F_{p_y} + D_{p_y}) \cap sup(E_{p_y}) = \phi$. By Lemma 4.4.2 we have either $p_x \subseteq F_{p_y} + E_{p_y}$ or $p_z \subseteq E_{p_y}$. If $p_x \subseteq F_{p_y} + E_{p_y}$, we have $p_x p_y \subseteq F + D$ which implies for p to be prime $p_z = 1$. Hence, $p_x p_y = p \subseteq F + D$. The other case is $p_x = 1$, hence $p = p_y p_z \subseteq E$, which is a don't-care and therefore redundant. Hence, all non-redundant cubes are contained in $F + D$ ■

The exact filter suggested by Theorem 4.4.3 is to discover the special structure illustrated in Figure 4.1 in the matrix representation of an incompletely specified function.

If the structure exists, all cubes in E can be removed from the don't-care set without any loss of optimality. This exact filter should always be applied prior to simplifying an incompletely specified function since it is easy to identify this structure. Notice that this exact filter identifies some in-essential don't-care cubes, but not all.

4.4.2 Topological interpretation of the exact filter

An example

A straight-forward way of using the exact filter when simplifying a function is to construct first the don't-care set and then apply the filter to remove in-essential cubes. However, it is advantageous not to generate those in-essential cubes in the first place in the presence of certain special circuit topologies. Therefore, it is important to understand the implications of the exact filter on the circuit topology and use that to guide the don't-care set generation.

The relationship between a certain structure of a cube matrix and the corresponding circuit topology can be best illustrated by a simple example.

EXAMPLE: Let η be a multi-level Boolean network with a single output. Let f be the primary output node, i.e. f is the root node of the decomposition circuit. Furthermore, there is no user-specified don't-care set. We would like to simplify node f.

Since f is a primary output, there is no fanout don't-care set, i.e. $DO_f = \phi$. The only don't-care set we can use to simplify f is

$$DI = \sum_{x \in (NODES(\eta) - \{f\})} (x_e \oplus x_l).$$

Suppose the circuit has a topology illustrated in Figure 4.2, i.e., there is an intermediate signal g such that the only connection between the circuit rooted at g, $\eta(g)$, and the rest of the network is through g. In other words, every path from a node of $\eta(g)$ to f has to pass node g. Thus f has been decomposed into $f(x_1, \ldots, x_k, g(x_{k+1}, \ldots, g_n))$. Let

$$\begin{aligned} E &= \textstyle\sum_{x \in NODES(\eta(g))} (x_l \oplus x_e) \\ D &= \textstyle\sum_{x \in NODES(\eta) - NODES(\eta(g))} (x_l \oplus x_e). \end{aligned}$$

It is clear that $DI = D + E$. Now, the only variable common to (f, D) and E is g. So, by Theorem 4.4.3 E is in-essential and therefore does not have to be generated. Notice that

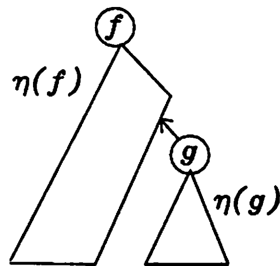


Figure 4.2: A special circuit structure

leaving E out implies that no variable of $\eta(g)$ can appear in the final minimized form of f except g.

The sub-circuit $\eta(g)$ having the above property is called a **disjoint component** of the network η. The don't-care set used to simplify f is generated only over those nodes of η which do not belong to any disjoint component of η.

Meaning of simplification

Having seen an example, it is appropriate to emphasize the meaning of simplification. To simplify a function f is to strong-divide a set of candidate factors simultaneously into f. Part of the don't-care set is the OR of DI_x's, each one stating a relationship of x with its fanins in a form that is useful for the simplification. If, for a given x, none of the fanins are needed to simplify f, then DI_x serves no purpose and can thus be omitted.

Region of interest (RI)

All the nodes which can not be excluded in generating the don't-care set for simplifying function f have potential values in simplifying f. These nodes form a region of interest around f which is formally defined below.

DEFINITION 4.4.4 *Let f be a node in a network η. A region of interest of f, denoted by $RI(f)$ is a sub-network consisting of a subset of nodes in η including f. All inputs to $RI(f)$ are considered as primary inputs and all outputs of $RI(f)$ are considered as primary outputs.*

$RI(f)$ is the environment around f in which f is to be simplified. It specifies a region over which the don't-care set is to be generated. Given a $RI(f)$, one can simplify

f in the sub-network $RI(f)$ rather than the entire original network. However, doing so may compromise the quality of the result. This leads to the following definitions for certain properties of RI.

DEFINITION 4.4.5 *A $RI(f)$ is DI-minimal if there is no proper subset of it, $RI'(f)$, in which f can get the same simplification using only the intermediate don't-care set. A $RI(f)$ is minimal if there is no proper subset of it, $RI'(f)$, in which f can get the same simplification using all internal don't-care set.*

For example, let η be the network used in Example 4.4.2. An example of $RI(f)$ is the entire network itself. However, this $RI(f)$ is not DI-minimal because f can get the same simplification in $RI(f) - \eta(g)$. This example also brings up another question: to simplify a function in a network, what is the minimal $RI(f)$ in which f can get as much simplification as in the network itself?

DEFINITION 4.4.6 *A $RI(f)$ is DI-sufficient if f can be simplified in $RI(f)$ as much as in the entire network using only the intermediate don't-care set. A $RI(f)$ is sufficient if f can be simplified in $RI(f)$ as much as in the entire network using all the internal don't-care set.*

Since the entire network is an instance of $RI(f)$, there is always a sufficient $RI(f)$ for any function f in any Boolean network. However, our interest is to find a sufficient $RI(f)$ which is also minimal.

A necessary condition for a DI-minimal RI

Since a node in a Boolean network can be arbitrarily complex, it is never sufficient to remove in-essential cubes of a don't-care set just by examining the circuit topology. For this reason, any condition based on circuit topology is only necessary but not sufficient. To study the necessary condition for an RI to be minimal, we generalize the notion of *disjoint component* used in Example 4.4.2.

DEFINITION 4.4.7 *A* **path** *between a pair nodes in a Boolean network is a set of edges, not necessarily pointing in the same direction, connecting the two nodes.*

For example, if $f \in FO(g)$ and $g \in FO(h)$, then $h - g - f$ is a path consisting of only forward edges. If $f \in FO(g)$ and $h \in FO(g)$, $h - g - f$ is a path consisting of backward edge $h - g$ and forward edge $g - f$.

DEFINITION 4.4.8 *A non-empty subset of nodes, D, of a network, η, is* **disjoint** *if there is an edge e that is on every path from a node in D to a node outside of D. The edge e in this cases is called* **joining edge.**

Notice that if D is a disjoint component of η, so is $\eta - D$.

THEOREM 4.4.9 *A $RI(f)$ is not DI-minimal if it contains a disjoint component.*

Proof. Let $RI^d(f)$ be the disjoint component not containing f. By definition, f can never be in a disjoint component. The don't-care set derived from the nodes in $RI^d(f)$ can at most have one variable in common with the rest of the don't-care set and the on-set, because there is only one edge between $RI^d(f)$ and $RI(f) - RI^d(f)$. By Theorem 4.4.3, the part of the don't-care set derived from $RI^d(f)$ is in-essential. So, f can get the same simplification in $RI(f) - RI^d(f)$ as in $RI(f)$. Therefore, $RI(f)$ is not DI-minimal. ■

Figure 4.3 illustrates some of the circuit topology $RI(f)$ which are not minimal. In all cases, the shaded areas represent the disjoint component which can be excluded from the don't-care set generation. The only connect between the shaded area and the rest of the network is by edge e. In case (a) and (b), e is a backward edge. In case (c) and (d), e is a forward edge. If edge e were deleted, then the transitive fanout and fanin of g is completely disconnected from the rest of $RI(f)$.

CONJECTURE 4.4.10 *A $RI(f)$ is not minimal if it contains a disjoint component.*

A sufficient condition for a DI-sufficient RI

Next, we would like to find out a condition for a $RI(f)$ to be sufficient in simplifying f. By our definition, the entire network $\eta(f)$ is obviously sufficient. We are interested in finding as small a sufficient $RI(f)$ as possible.

THEOREM 4.4.11 *Let f be a function to be simplified, and $\eta(f)$ be the network which f belongs to. Let $RI(f)$ be a region of interest of f obtained by removing all the disjoint components from $\eta(f)$. $RI(f)$ is DI-sufficient.*

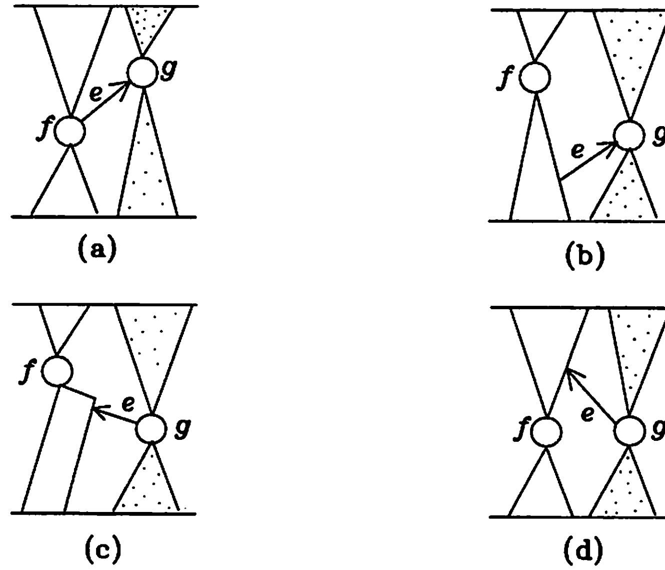


Figure 4.3: Several examples of non-minimal $RI(f)$

Proof. Given a f and $\eta(f)$, there is only a finite number of disjoint components. By applying Theorem 4.4.9 repeatedly, one can obtain the $RI(f)$ such that f can get the same simplification in $RI(f)$ as in $\eta(f)$. ■

COROLLARY 4.4.12 *Let $RI(f)$ be DI-sufficient in a network η.*

1. *Every out-going edge of $RI(f)$ is either a joining edge or a primary output of η.*
2. *Every in-coming edge of $RI(f)$ is either a joining edge or a primary input of η.*

CONJECTURE 4.4.13 *Let f be a function to be simplified, and $\eta(f)$ be the network which f belongs to. Let $RI(f)$ be a region of interest of f obtained by removing all the disjoint components from $\eta(f)$. $RI(f)$ is sufficient.*

The structure of a DI-sufficient $RI(f)$ is pictured in Figure 4.4. In the figure, the region around f is $RI(f)$. The edges going out of $RI(f)$ are considered as primary outputs of $RI(f)$, $PO(RI(f))$. The edges coming in to $RI(f)$ are considered as primary inputs of $RI(f)$, $PI(RI(f))$. Both e_g and e_h are joining edges. The shaded area covering g is part of the network reachable from g using both forward and backward edges excluding edge e_g. The fact that e_g is a joining edge implies that the shaded area covering g is a disjoint

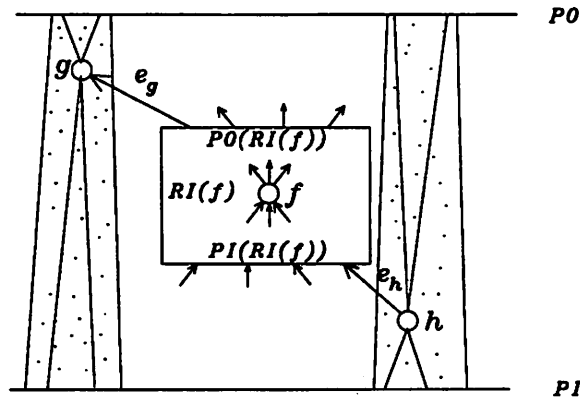


Figure 4.4: Structure of DI-sufficient $RI(f)$

component of $\eta(f)$ and therefore does not have to be part of $RI(f)$. Likewise, the shaded area covering h is part of the network reachable from h and is also a disjoint component of $\eta(f)$ with e_h being the joining edge. It too can be excluded from $RI(f)$.

Summary

To summarize, the exact filter is designed to remove certain in-essential cubes from the don't-care set when its cube matrix has a special structure. When simplifying a function f, the environment in which f is to be simplified is called the region of interest of f, $RI(f)$. The topological interpretation of the exact filter gives us a way to reduce the size of $RI(f)$, i.e. removing all of its disjoint components. Finally, a DI-sufficient $RI(f)$ can be obtained by removing maximally disjoint components from $\eta(f)$. A function f can get the same simplification in a DI-sufficient $RI(f)$ as in $\eta(f)$ using only the intermediate don't-care set.

4.4.3 Approximate filters

In practice, there are cases where the don't-care set filtered by the exact filter is still too large for existing two-level minimization programs to handle efficiently. So, further reduction of the don't-care set is needed to obtain a desired run time efficiency at the expense of possibly a little loss in optimality. Those reductions are called approximate filters. Approximate filters try to remove cubes from the don't-care set which are unlikely to be useful, but with no guarantees. The approximate filters were derived through experiments

and have been shown to be effective, i.e. they have been able to reduce run time drastically and still permit significant optimization.

Sub-support filter

This is a filter based on circuit topology [2]. When simplifying a function f, this filter tries to reduce the size of a don't-care set by restricting the region of interest to only those nodes whose supports are entirely contained by the support of f. The precise definition of the don't-care set generated by this filter is:

$$\begin{aligned} RI(f) &= \{x \in NODES(\eta(f)) | sup(x) \subseteq sup(f)\} \\ DC(f) &= \textstyle\sum_{x \in RI(f)-\{f\}} (x_l \oplus x_e). \end{aligned}$$

The don't-care set $DC(f)$ used for simplification contains fanin don't-care set only. This filter has been shown to be very effective and efficient, i.e. achieve good minimization quality with acceptable run time requirements [46].

Disjoint support filter

This filter is defined on the matrix representation of a don't-care set. Given an incompletely specified function $(F, D+E)$ such that its matrix has the structure indicated in Figure 4.5, then all cubes in E are removed by this filter. While this filter cannot guarantee

<table>
<tr><td>F</td><td>2</td></tr>
<tr><td colspan="2">D</td></tr>
<tr><td>2</td><td>E</td></tr>
</table>

Figure 4.5: Structure that disjoint support filter looks for

that the cubes in E are in-essential, it has been shown through experiment that they are unlikely to be used [3].

Maximum support filter

This is again a filter defined on the matrix representation. Each cube of the don't-care set introduces a set of possible new variables which may be used during simplification.

[2]This filter was suggested by Alex Saldanha

[3]The experiment was done by Mauro Pipponzi of SGS, Italy

The goal of this filter is to limit the size of the don't-care set by imposing a limit on the maximum support size. The limit can be specified as one of the two ways:

1. Specifying explicitly the *set* of new variables allowed to be used during simplification. Any cube of a don't-care set containing variables other than those specified is removed.

2. Specifying the *number* of new variables allowed during simplification. Given a don't-care set, the cubes are examined in certain sequence. A cube is kept if it does not introduce more variables to the existing support than allowed by the limit. By varying the limit, tradeoffs between run time and result quality can be obtained.

4.4.4 Putting it together

To simplify a function using various filters presented in this section, one should: 1) find the region of interest $RI(f)$, 2) restrict $RI(f)$ using the exact filter or sub-support filter, 3) generate the matrix representation of the don't-care set, 4) apply the exact filter on the matrix to remove more cubes, 5) apply appropriate approximate filters to reduce further the size of the don't-care set, and 6) then simplify function f with the filtered don't-care set. The reason for applying the exact filter on the matrix again is because the function at a node can be arbitrarily complex and certain in-essential cubes can not be discovered by looking at the circuit topology alone.

4.5 Don't-care Set Representation

The internal don't-care set of a function is a property of the Boolean network and is implicitly specified by the network topology and node functions. There have been several methods for simplifying functions using the don't-care sets implicitly [49] [4]. However, the simplification procedure presented in this chapter requires the don't-care sets to be expressed explicitly, in particular, in sum-of-products forms.

One drawback of representing the don't-care sets in sum-of-products forms is that their size (number of cubes) can get very large, sometime too large to be handled efficiently, or at all, within the capacity of current computer systems. For example, if the function at a node is:

$$x = ab + cd + ef + gh,$$

then D_x, computed by $x_l \oplus x_e$, is

$$\begin{aligned} D_x &= x'(ab + cd + ef + gh) + (ab + cd + ef + gh)'x \\ &= x'ab + x'cd + x'ef + x'gh + \\ &\quad xaceg + xaceh + xacfg + xacfh + \\ &\quad xadeg + xadeh + xadfg + xadfh + \\ &\quad xbdeg + xbdeh + xbdfg + xbdfh + \\ &\quad xbceg + xbceh + xbcfg + xbcfh. \end{aligned}$$

It is evident that the don't-care set D_x would be un-manageable if x contains few more disjoint cubes. The source of the size increase is clearly due to the complementation of expression x_e, and the multiplication of sum-of-products forms.

The situation can get even worse when computing fanout don't-care sets. In fact, the formula for computing the one-level fan-out don't-care set of a function f is:

$$DO_f = \prod_{x \in FO(f)} (x_f \overline{\oplus} x_{f'}).$$

Notice that in addition to the complementation, DO_f is a product of fanout don't-care conditions of all output nodes of f.

In this section, two methods are proposed to represent the don't-care sets in a more compact form, i.e., with fewer number of cubes. One method reduces the number of cubes in a don't-care set by introducing new variables, and the other by representing the don't-care set in its complemented form.

4.5.1 Reduction using new variables

As we have seen in the previous example, functions with many disjoint set of cubes have complements which are very large. In this case, the size of the complement can be reduced by introducing new variables.

Suppose there is a function $f = g + h$ such that g and h have disjoint support, i.e., $g \perp h$. Then

$$\begin{aligned} f &= g + h \\ DI_f &= f_l'(g + h) + f_l g'h' \\ |DI_f| &= |g| + |h| + |g'| \times |h'| \end{aligned}$$

If two new variables, x and y are used to represent g and h, then

$$\begin{aligned} f &= x + y \\ x &= g \\ y &= h \\ DI_f &= f_l'(x_l + y_l) + f_l x_l' y_l' + x_l' g + x_l g' + y_l' h + y_l h' \\ |DI_f| &= 3 + |g| + |h| + |g'| + |h'|. \end{aligned}$$

It is clear that $|g'| \times |h'|$ is avoided by introducing two new variables. As a by-product of this method, more simplification is possible using this representation of the don't-care set because the new intermediate functions introduced may be better than the existing ones.

So, the general procedure for computing DI is to introduce first a set of new variables, each one representing a set of disjoint cubes in a function, then carry out the usual computation of DI. Notice that there are functions which do not have a partition of disjoint cubes and still have exponential number of cubes in the complements. For example, the complement of the following function can still be very large even though it does not have partitions of disjoint cubes.

$$x = abc + cdef + fghi + ijkl + ...$$

One solution to this problem is to partition the cubes in x to "almost" disjoint sets and still use a new variable to represent each set. In this case, an analysis should be performed to see if he complement is indeed too large. [16] provides an algorithm, $ESTCOMP$, which can be used to estimate the size of the complement.

It should be pointed out that the introduction of new variables may put extra burden on the minimization program because of the extra information, but may allow better solutions to be found because of the availability of more intermediate variables. However, the main objective of this approach is to have a way of representing the don't-care set that we could not otherwise represent before. As long as we have a representation of the don't-care set, filters can be used to reduce its size.

4.5.2 Reduction by complementation

The second source of problem is the multiplication in the computing fanout don't-care sets. Again, an example is used to show a possible solution. $FO(x)$ and DO_x denote

the fanouts of x and the fanout don't-care of x respectively.

$$\begin{aligned} FO(x) &= \{f,g\} \\ DO_x^f &= f_x \overline{\oplus} f_{x'} \\ DO_x^g &= f_x \overline{\oplus} g_{x'} \\ DO_x &= DO_x^g DO_x^f \\ |DO_x| &= |DO_x^f| \times |DO_x^g| \end{aligned}$$

If DO_x is complemented, we would have

$$\begin{aligned} \overline{DO_x} &= \overline{DO_x^f} + \overline{DO_x^g} \\ |\overline{DO_x}| &= |\overline{DO_x^f}| + |\overline{DO_x^g}| \end{aligned}$$

Since

$$|\overline{DO_x^f}| = |f_x \overline{\oplus} f_{x'}| \approx |f_x \oplus f_{x'}| = |(DO_x^f)|$$

we have

$$|\overline{DO_x}| = |\overline{DO_x^f}| + |\overline{DO_x^g}| \approx |DO_x^f| + |DO_x^g|.$$

Computing DO_x involves multiplication. But, computing $\overline{DO_x}$ only involves addition. This example suggests that the complement of a fanout don't-care set is usually more compact. But, how do we use it in a minimizer?

4.5.3 Requirement for a new two-level minimizer

As defined earlier, the internal don't-care set of a function in a Boolean network has two components, the intermediate don't-care set and the fan-out don't-care set. The previous section has shown that the complement of the fan-out don't-care set has, in general, more compact representation. Therefore, using this approach, the don't-care set is in the form of $D + \overline{E}$ where D represents the intermediate don't-care and E represents the complement of the one-level fan-out don't-care set. This particular representation of the don't-care set poses a new requirement for the two-level minimization algorithms, i.e., being able to operate on this special representation of the don't-care set directly.

Currently, such a minimizer does not exist. However, simple modifications can be made to the existing algorithms to accommodate this new requirement. Instead of giving a complete description for such an algorithm, we show here that one of the basic operation, the containment operation, in two-level minimization can be carried out using this representation directly.

Suppose we want to check the condition $p \subseteq D+\overline{E}$. The following set of equivalent conditions can be derived,

$$\begin{aligned} p \subseteq D+\overline{E} &\Leftrightarrow (D+\overline{E})_p = 1 \\ &\Leftrightarrow D_p + \overline{E}_p = 1 \\ &\Leftrightarrow D_p + \overline{E_p} = 1 \\ &\Leftrightarrow E_p \subseteq D_p \end{aligned}$$

So, checking whether $p \subseteq D+\overline{E}$ is equivalent to checking that $E_p \subseteq D_p$. Thus, cube-containment operation can be carried out on the don't-care of form $D+\overline{E}$ without having to complement E at all. However, the drawback of this approach is that it avoids doing the complementation by doing more tautology checking in determining $E_p \subseteq D_q$. So, in implementing the new minimizer, one needs to dynamically choose between complementation and tautology strategy, based on estimating the size of the complement.

4.6 Literal Weight

The objective of this chapter is to reduce the size of a Boolean network by simplifying each function in the network using appropriate don't-cares. There is an assumption being made at each step of this process. In simplifying a function using internal don't-cares, it is assumed that all the intermediate functions exist in the network regardless of whether they are used in the final simplified form of the function, when in fact some of them can be deleted from the network if not used. For example, Figure 4.6 shows a case where not

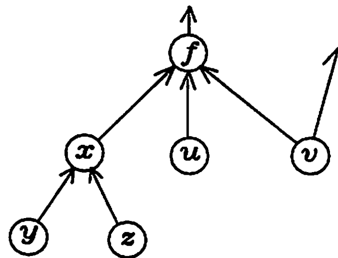


Figure 4.6: A case of non-equal literals of f

all intermediate variables can be treated as equal. Suppose in the process of simplifying f, a two-level minimizer discovers three equal size covers using variables $\{x,u\}$, $\{x,v\}$, and $\{u,v\}$ respectively. The two-level minimizer, without any other information, could choose

any one of them. Here, even though the choice does not affect the size of simplified f, it does affect the total size of the network. If the cover with variables $\{x, v\}$ is chosen, node u can be deleted. Better yet, if the cover with variables $\{u, v\}$ is chosen, not only can node x be deleted, nodes y and z can be deleted as well. However, if the cover with variable $\{x, u\}$ is used, no nodes can be deleted because v is used by some other nodes in the network. So, to simplify f correctly, each of the possible variables has to have a weight reflecting the amount of reduction if it is not used in the final cover of f. In the example, variable x should have the largest weight, and variable v should have the smallest weight.

To take into account the literal weights, current two-level minimizers need to be modified to find minimal weighted covers of incompletely specified functions. Nevertheless, there is a simple modification of procedure *SIMPLIFY* which avoids this problem of non-equal literal weights without requiring a new two-level minimizer.

Right before simplifying a function f, all of the inputs are examined. If an input x is used only in function f, it is collapsed into f. This process repeats until the fanins of f are either primary inputs or multiple-fanout nodes. After this, the normal simplification can be performed on f where every variable can be treated equally. This procedure is outlined as follows:

SIMPLIFY'(f)
  while $\exists x \in FI(f)$ such that $FO(x) = \{f\}$ {
    collapse(f, x)
  }
  $DC = DC_GEN(f)$
  $MINIMIZE(f, DC)$

4.7 Experiments and Results

The simplification procedure has been implemented in MIS with an extensive list of options designed to allow various experiments. Using the options, one can select various two-level minimization algorithms, specify criteria by which the simplification results are accepted, define the region of interest, and choose the type of don't-care sets. Most of the

results presented in this section were obtained by various combinations of these options. The objectives of the experiments are listed below:

1. To get a feel on the size of the problems we are dealing with, in particular the size of don't-care sets.
2. To show the effect of filters, exact and heuristic, on the size of don't-care sets.
3. To show that the size of don't-care sets can be reduced by introducing new variables or by representing their complements.
4. To compare the quality of results obtained by using various don't-care sets.

The examples used in the experiments are taken from the MIS benchmark set consisting of over one hundred circuits. They are either from the MCNC logic synthesis workshop benchmark set, or actual circuits from industry. The circuits are first optimized using MIS with the standard algebraic script. In all experiments the sizes of a don't-care set is measured by the number of cubes.

The first experiment simply generates the entire fanin don't-care set for all the benchmark circuits. Some of the size of these don't-cares are tabulated in Table 4.1. The second column, DI, lists the number of cubes in the fanin don't-care set. the don't-care sets are generally quite large as compared with the kind of don't-care sets we see in two-level logic design. In addition, the technique of reducing the size of fanin don't-care sets by introducing new variables is explored. Before generating the fanin don't-care set, the functions are first decomposed into disjoint components, each one being represented by a new variable. The size of don't-care sets in this new representation is listed in the last column, DI with new variables. It was intended to show that disjoint decomposition may help reduce the size of fanin don't-care sets. However, the results showed that in all but a few examples, the size went up rather than down. There are three examples, 9sym, rd73, and yannis, whose don't-care sizes went down significantly with disjoint decomposition. The conclusion is that disjoint decomposition should only be used on the nodes whose complements are potentially large. [16] has an algorithm, ESTCOMP, for estimating the size of the complement.

The next experiment studies the fanout don't-care set. In this experiment, we restrict ourselves to the immediate fanout don't-care sets only, because the correct computation of general fanout don't-care sets has not yet been implemented in MIS. Since fanout don't-care sets are associated with nodes, sizes of fanout don't-care sets are presented in

example	DI	DI with new var.
9sym	719	500
f51m	216	343
misex2	159	185
rd73	328	271
sao2	304	364
cafmt	127	154
cfeed	327	327
ciadr	104	149
cntmux	103	150
count4	251	359
dinmux	121	193
gprdec	712	712
iseset	133	167
maskgn	322	530
neeeds	122	182
pbo2	343	466
pbonew	534	661
pcdmux	131	131
resmux	128	128
runset	163	183
s0mux	186	276
setpmr	205	311
yannis	152	104
ampms	402	597
chest	195	263

Table 4.1: Sizes of the entire fanin don't-cares and their alternative representation

terms of average and maximum sizes. Again, a related experiment is to show how the sizes are reduced by representing the complement of the don't-care sets. The results are summarized in Table 4.2. It is evident from the table that even though the average size of the

example	ave. DO	max. DO	ave. DO in comp.	max. DO in comp.
5xp1	4	58	2	13
9sym	374	14362	28	160
alupla	5	48	3	9
bw	10	196	4	20
f51m	6	86	3	12
rd73	62	342	25	83
sao2	44	592	7	40
vg2	4	20	2	12
cafmt	3	16	1	6
cfeed	2	27	1	3
cntmux	9	127	2	10
iseset	7	22	3	9
maskgn	2	48	2	14
neeeds	6	15	3	9
pbo2	24	391	4	37
pbonew	83	2594	7	50
setpmr	4	65	2	15
yannis	19	50	7	16
amprx	11	660	2	14
chest	394	7072	3	32
ctcucon	19	388	2	16

Table 4.2: Sizes of one level fanout don't-cares and their complements

fanout don't-care sets are small, it becomes occasionally very large as indicated by examples 9sym, pbonew, and chest. The next two column show the size of the fanout don't-care sets when represented as their complements. As expected, the sizes of the complements are significantly smaller.

Numbers in previous tables show that the don't-care sets found in a multi-level Boolean network are almost always too large to be handled efficiently by existing two-level minimization programs such as ESPRESSO. So, the filters, both exact as well as heuristic, are applied to reduce the don't-care sizes. The results are reported in Table 4.3. The first two columns list example names and the size of the fan-in don't-care sets. The next two columns report the average and maximum sizes of don't-care sets after the exact filter. The

example	DI	ave. DI exact filter	max. DI exact filter	ave. DI heuristic filter	max. DI heuristic filter
alupla	253	236	247	24	54
bw	312	300	308	50	99
f51m	216	75	86	20	37
misex1	93	83	89	24	44
misex2	159	150	156	20	36
rd53	81	69	75	22	47
rd73	328	277	311	34	93
sao2	304	289	301	50	116
vg2	132	105	124	16	42
alui	576	570	573	29	71
cafmt	127	113	121	33	55
camux	18	2	5	2	4
cbmux	93	80	86	28	48
ccrup	105	61	84	16	39
cfeed	327	307	321	13	18
ciadr	104	94	99	35	52
cntmux	103	56	82	18	37
dinmux	121	44	81	30	66
gprdec	712	132	143	23	26
iseset	133	114	127	34	65
maskgn	322	314	319	32	68
mdrmux	192	186	186	155	155
neeeds	122	111	116	41	68
pbo2	343	314	328	67	140
pbonew	534	503	523	68	162
runset	163	31	105	6	40
s0mux	186	163	174	136	145
setpmr	205	133	179	19	49

Table 4.3: Effect of exact and heuristic filters on the sizes of fanin don't-care set

last two columns report the average and maximum sizes of don't-care sets after the heuristic filters. Fair amounts of reductions are obtained by the exact filter on all the examples. In some cases, the reduction is quite significant as demonstrated by example f51m, z5ml, and pmoset. However, the results are still too large. So, heuristic filters are applied to reduce further the don't-care sizes. With few exceptions, the don't-care sets after being reduced by heuristic filters are much smaller and can be reasonably handled by two-level minimization tools like ESPRESSO.

The same experiment is performed to filter the fanout don't-care sets. The results are summarized in Table 4.4. It is worth noting that the exact filter did not perform as

example	ave. DO	max. DO	ave. DO exact filter	max. DO exact filter	ave. DO heuristic filter	max. DO heuristic filter
5xp1	4	58	2	41	0	1
9sym	374	14362	180	5418	1	4
bw	10	196	3	164	0	1
f51m	6	86	2	19	0	6
misex1	4	16	1	9	0	2
rd53	4	18	4	18	1	3
rd73	62	342	37	342	0	5
sao2	44	592	19	536	1	8
vg2	4	20	2	20	0	0
z4ml	1	3	1	3	1	3
afmt	3	16	1	7	0	7
cafmt	3	16	1	7	0	7
cbmux	4	29	3	29	0	4
cfeed	2	27	0	0	0	0
ciadr	3	11	1	9	1	4
cntmux	9	127	0	0	0	0
crbus	4	11	2	11	1	3
iseset	7	22	6	19	1	3
neeeds	6	15	2	15	1	6
pbo2	24	391	11	329	0	4
pbonew	83	2594	63	2582	0	9

Table 4.4: Effect of exact and heuristic filters on fanout don't-cares

well as it did for fanin don't-cares because if is a product. On the other hand, the heuristic filters did very well on the fanout don't-cares. In fact, they may filter out too much.

The last set of experiments compare results of simplification using various don't-care sets. Table 4.5 summarizes the results. Column SBS lists the results of using the

example	SBS		INOUT		INOUT-E	
5xp1	145	2.5	145	6.3	145	6.1
9sym	35	0.6	35	1.0	35	0.8
alupla	24	0.8	24	4.1	24	2.2
bw	265	26.0	279	36.0	279	46.8
con1	0	0.1	1	0.2	1	0.1
duke2	876	24.6	1409	685.8	1409	632.5
f2	8	0.2	8	0.3	8	0.3
f51m	126	1.5	126	6.0	126	5.2
misex1	68	0.4	88	1.3	88	1.4
misex2	39	1.4	41	18.1	41	23.1
rd53	44	0.3	44	0.5	44	0.4
rd73	427	2.0	427	2.8	427	2.8
rd84	920	11.4	920	15.0	920	14.6
sao2	294	2.4	294	3.4	294	3.2
z4ml	166	0.7	166	1.6	166	1.5

Table 4.5: Simplification results using various don't-care sets

subset-support filter. Column INOUT lists the results of using don't-cares generated over one-level of fanin and then one level of fanout of the fanins. Column INOUT-E is the same as INOUT column with don't-cares being filtered by the exact filter. In each case, there are two columns, the first indicates the literal reductions and the second indicates the CPU time. The literal reduction is the sum of literal reductions of all the nodes. The literal reductions of a node is measured independent of other simplifications, i.e. each node is simplified, measured, and put back to its original form. The reason for doing so is to remove the effect of simplification orderings. The results show that subset-support filter in general performs quite well, i.e. obtains comparable results (except duke2) with less time. The INOUT filter allows more nodes to be used in generating the don't-cares. Consequently, it could only perform better, as shown by the result of duke2. As the theorem predicts, using exact filter on the don't-cares does not change the results quality, as verified by the table, but may affect the run time. On some examples, the CPU time went down because the exact filter reduced the don't-care sets. In other cases, the CPU time went up, an indication that the exact filter itself is a quite expensive operation and, when no reductions are possible, the time spent on filtering is wasted, as shown by the results of bw and misex2.

4.8 Future Work

Since simplification is a form of Boolean substitution, it is possible to have a situation where any one of the two functions f and g can be substituted into the other, resulting in two different networks. This is the simplest situation where the order of simplification is important. The problem gets even more complicated when more nodes are involved. Trying out all possible orderings is simply too expensive to be a viable solution. Preliminary experiments have shown that significant improvement can be made by choosing the right orderings. The subject is well worth further investigation.

A heuristic used in the simplification process is to use the size of sum-of-products forms as approximation to the size of factored forms, when in reality the two quantities are not related well enough. So, there is a strong need for a simplification program which optimizes an incompletely specified function for factored literals.

Section 4.5 has shown that some don't-care sets have more compact sum-of-products representation in the complement. Consequently, the don't-care set used for simplifying a function may have both the un-complemented and complemented components, i.e., $DC = DI + \overline{DOC}$ where $DOC = \overline{DO}$. Modifications are needed to the existing two-level minimization algorithm which handle the don't-care set expressed in both un-complemented and complemented form.

Chapter 5

Phase Assignment

5.1 Introduction

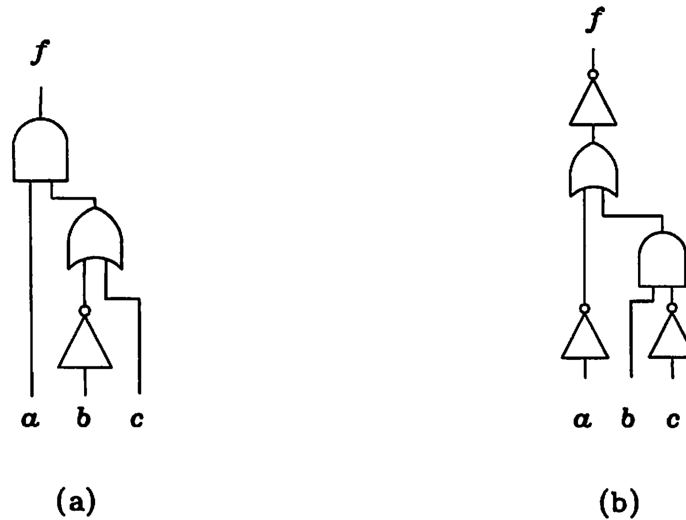
The principle of duality is an important property of Boolean functions. It states that every algebraic expression deducible from the basic axioms of Boolean algebra remains valid if both operators (i.e. AND and OR) and identity elements (i.e. 1 and 0) are interchanged. The duality principle is formally expressed as the following specific form of De Morgan's law:

$$\overline{x+y} = \overline{x}\;\overline{y}$$

$$\overline{xy} = \overline{x} + \overline{y}$$

The duality principle makes it possible to implement a Boolean function in either uncomplemented or complemented form provided that necessary inverters are supplied at the inputs as well as the output of the function. For example, the function $a(\overline{b}+c)$ can be realized by either of the circuits in Figure 5.1.

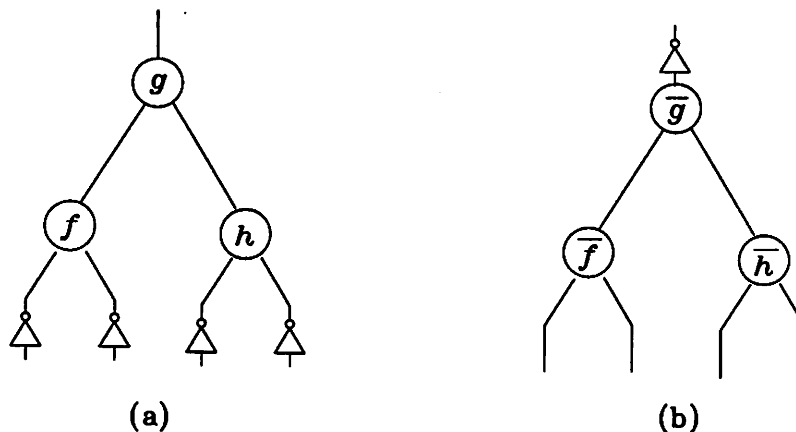
In a given multi-level combinational logic network, each function can be implemented in its present or complemented form provided that appropriate inverters are supplied. However, the costs of implementing a function or its complement may differ. For example in static CMOS technology, NAND gates are generally preferred over NOR gates due to performance considerations. More importantly, the choice of implementing a function in its present or complemented form affects the number of inverters needed at the inputs and output of the function, which may in turn affect how other functions are to be implemented. For example in Figure 5.2, circuit (b) can be derived from circuit (a) by



(a) (b)

Figure 5.1: Equivalent implementations of $a(\overline{b}+c)$

complementing functions f, g, and h and has fewer number of inverters than circuit (a) does.



(a) (b)

Figure 5.2: Reducing inverters by complementing certain functions

For a Boolean network, whether the present or the complemented form of logic functions should be implemented to minimize the cost of implementing the network is a global problem, and is therefore defined as **global phase assignment** problem or simply **phase assignment**. The cost of a Boolean network may vary, depending on the optimization criteria. Typical cost functions are:

1. Total area occupied by the logic in the network.

2. Total number of inverters in the network.

3. Longest delay in the network.

The first cost function is used during the area optimization phase, and for certain technologies it can be reduced to the second cost function. Whereas the third cost function is often used in conjunction with other performance optimization algorithms. This chapter is dedicated to solving the problem of minimizing the total number of inverters. However, the formulations and algorithms can easily be extended to accommodate more complex cost functions.

To solve the global phase assignment problem, the first question to be answered is whether an algorithm exists to solve the problem optimally in a reasonable amount of time. It will be shown in Section 2 that the problem is NP-complete. Only when the network is in a very specific form can the phase assignment problem be solved optimally in polynomial time. A dynamic programming algorithm will be designed to minimize the total number of inverters in a tree network.

To offer a practical solution to the phase assignment problem, several heuristic algorithms have been developed, tested on a large set of examples, and experimentally shown to be very efficient and effective. They will be discussed in Section 3.

When used in a cell-based design style, e.g., standard cell or gate array technology, the phase assignment problem can be extended to allow more general modifications to functions than just complementation. New modifications will be formally defined and heuristic algorithms, developed in Section 3, will be extended to solve this **generalized phase assignment** problem in Section 4.

To evaluate the heuristic algorithms and to compare their relative performance, experimental results will be presented and examined in Section 5.

Finally, open problems and future directions will be discussed in Section 6.

5.2 Basic Definitions

In a given Boolean network, every signal has two *phases*. The presently available phase of a signal is referred to as its *positive phase*. The complement of a signal is referred to as its *negative phase*. Notice that these definitions are relative to the **current** configuration of a given Boolean network.

Inverters in a Boolean network can be specified implicitly by allowing signals to appear in logic equations as either literals or complement of literals. For example, when given two equation $f = \overline{x}a$ and $x = bc$, it is implicitly understood that an inverter is needed in the final implementation to provide $\overline{x}$. A Boolean network is said to be *phase-consistent* if all the inverters needed are explicitly present in the network.

The *dual* of a Boolean expression f, denoted by d_f, is another expression obtained by switching AND's and OR's in f. For example, if $f = a(\overline{b} + c)$ then $d_f = a + \overline{b}c$. It is obvious that the dual of f is not the complement of f. By De Morgan's law, the complement of f can be obtained by switching all AND's and OR's and invert all literals. Using the same example, if $f = a(\overline{b} + c)$, then $\overline{f} = \overline{a(\overline{b} + c)} = \overline{a} + b\overline{c}$. To formalize these definitions, let $F(*, +, x, y, \ldots, z)$ be an expression for $f(x, y, \ldots, z)$. Then f and d_f are

$$d_{f(x,y,\ldots,z)} = F(+, *, x, y, \ldots, z)$$

$$\overline{f(x, y, \ldots, z)} = F(+, *, \overline{x}, \overline{y}, \ldots, \overline{z}).$$

PROPOSITION 5.2.1 *The operation* dual *is self-inverse, i.e.,*

$$d_{d_f} = f$$

Proof.

$$\begin{aligned} d_{d_{f(x,y,\ldots,z)}} &= d_{d_{F(*,+,x,y,\ldots,z)}} \\ &= d_{F(+,*,x,y,\ldots,z)} \\ &= F(*, +, x, y, \ldots, z) \\ &= f(x, y, \ldots, z) \end{aligned}$$

■

PROPOSITION 5.2.2 *Every function can be replaced by its dual with inverters each input and the output, i.e.,*

$$f(x, y, \ldots, z) = \overline{d_f(\overline{x}, \overline{y}, \ldots, \overline{z})}$$

Proof.

$$\begin{aligned} f(x,y,\ldots,z) &= \overline{\overline{f(x,y,\ldots,z)}} \\ &= \overline{\overline{F(*,+,x,y,\ldots,z)}} \\ &= \overline{F(+,*,\overline{x},\overline{y},\ldots,\overline{z})} \\ &= \overline{d_f(\overline{x},\overline{y},\ldots,\overline{z})} \end{aligned}$$

■

The proposition can be interpreted as follows: if a function is replaced by its dual, then inverters have to be added to all the inputs and the output in order to preserve the logic, as indicated in Figure 5.3. Notice that buffers and inverters are the self-dual.

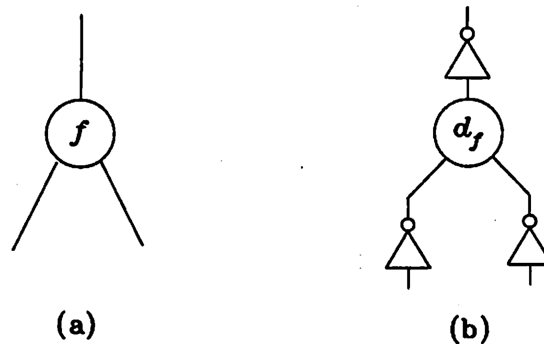


Figure 5.3: Replacing f by its dual d_f

Sometimes, replacing a node f with its dual d_f is referred to as *negating* f or *flipping* f.

5.3 Complexity and Formulation

Is phase assignment a difficult problem? Can we solve the problem exactly in practice? If an exact algorithm is unlikely to be found, can we solve the problem exactly when the networks possess some special properties? These are the questions to be studied and answered in this section.

First, the complexity of the phase assignment problem is investigated and the decision problem associated with the phase assignment problem will be shown to be NP-complete. In addition, the problem is formulated as a mathematical programming problem with a linear objective function and non-linear constraints. Finally, a dynamic programming

algorithm running in polynomial time is proposed to find the optimum solution of the phase assignment problem for tree networks.

5.3.1 Complexity of Phase Assignment

Given a Boolean network with n nodes, the size of the solution space of the phase assignment problem is bounded by 2^n simply because each node has the choice of being replaced by its dual or not, i.e. the problem is in NP. Unfortunately, it will be shown that it is very unlikely to find an algorithm that is substantially better than searching through all possible solutions, i.e. the phase assignment problem is NP-complete.

To simplify the proof, it is convenient to look at the decision problem associated with phase assignment. The decision problem of phase assignment can be informally stated as: given an instance of a Boolean network with n nodes, is there a phase assignment requiring no more than K inverters? This decision problem can be no harder than its corresponding optimization problem. Clearly, any algorithm that solves the minimum-inverter phase-assignment problem also answers the question of the decision problem at the same time.

The decision problem of phase assignment can be proved to be NP-complete by transformation from the **MAX CUT** problem which is defined as

MAX CUT
INSTANCE: Graph $G = (V, E)$, weight $w(e) \in Z^+$ for each $e \in E$, positive integer K.
QUESTION: Is there a partition of V into two disjoint sets V_1 and V_2 such that the sum of the weights of the edges from E that have one endpoint in V_1 and one endpoint in V_2 is at least K?

The **MAX CUT** problem is NP-complete [30] and remains NP-complete if $w(e) = 1$ for all $e \in E$ (the SIMPLE MAX CUT problem) [25].

The decision problem of phase assignment is formally defined as:

PHASE ASSIGNMENT
INSTANCE: Boolean network η, dual d_f for each $f \in NODES(\eta)$, positive integer L.
QUESTION: Is there a subset of functions $F \subseteq NODES(\eta)$ such that replacing each $f \in F$

by d_f results in a Boolean network with at most L inverters?

The following lemma is needed to prove that PHASE ASSIGNMENT is NP-complete.

LEMMA 5.3.1 *Given a graph $G = (V, E)$, assigning directions to the edges in E such that the resulting directed graph is acyclic is a polynomial time operation in the number of edges .*

Proof. The following algorithm assigns directions to edges of G in polynomial time. Without loss of generality and for the sake of simplicity, G is assumed to be connected. In the algorithm, $SUB_GRAPH(G, S)$ is a graph whose vertices are $S \subseteq V$ and whose edges are all edges of G with both endpoints in S.

$ASSIGN_DIRECTION(G = (V, E))$

1. Let $U = \phi$, and $W = V$.
2. Pick any $v \in W$, let $U = U \cup \{v\}$ and $W = W - \{v\}$.
3. Let $S = \phi$.
4. For each edge (u, w) such that $u \in U$ and $w \in W$, assign direction $u \rightarrow w$, and $S = S \cup \{w\}$.
5. $ASSIGN_DIRECTION(SUB_GRAPH(G, S))$.
6. Let $W = W - S$, and $U = U \cup S$.
7. If $W \neq \phi$ goto step 3, otherwise done.

In the algorithm, U contains the partial result. Each time when S is added to U, all edges between U and S are pointing from U to S. Because $|S| < |V|$, by simple induction on the number of nodes in the graph, $SUB_GRAPH(S)$ can be made acyclic by recursive call to $ASSIGN_DIRECTION$. Eventually, U will contain all the nodes of G. So, the resulting directed graph is acyclic. The algorithm is $O(|E|)$ because each edge is encountered exactly once. ∎

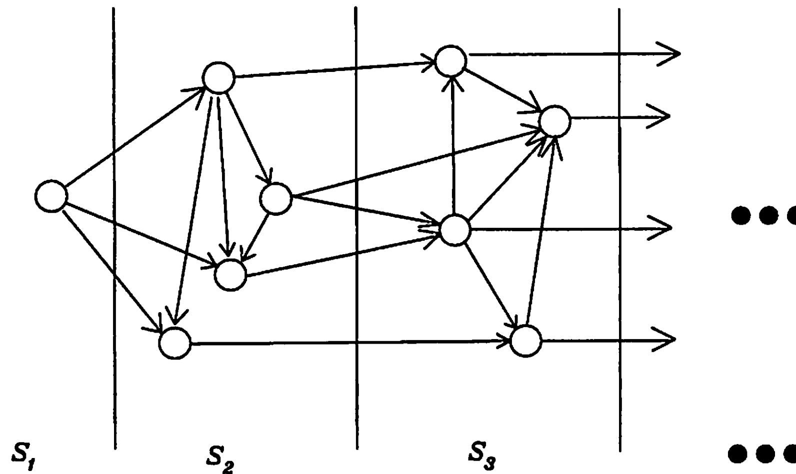


Figure 5.4: Steps in constructing a DAG from a graph

The algorithm used in the proof can be visualized in Figure 5.4, where $S_1, S_2, \ldots$ refer to S constructed at step 4 in each iteration.

To prove that **PHASE ASSIGNMENT** is NP-complete, the **SIMPLE MAX CUT** is transformed to **PHASE ASSIGNMENT**. Recall that **SIMPLE MAX CUT** is to partition the nodes of a graph into two sets, V_1 and V_2, such that the number of crossing edges is maximized. The transformation maps the graph G to a phase-consistent Boolean network η. V_1 is mapped to the set of nodes of η which are to be replaced by their duals. V_2 corresponds to the un-changed nodes. Maximizing the edges across the partition is transformed to minimizing the number of inverters.

THEOREM 5.3.2 *PHASE ASSIGNMENT is NP-complete.*

Proof. It is easy to see that PHASE ASSIGNMENT $\in$ NP, because a nondeterministic algorithm need only guess a set of functions $f \in NODES(\eta)$, replace them with their duals, and check in polynomial time that the number of inverters needed in the network η is less than or equal to L.

The SIMPLE MAX CUT can be transformed to PHASE ASSIGNMENT in polynomial time. Let an arbitrary instance of SIMPLE MAX CUT be given by the graph $G = (V, E)$ and the positive integer $K \leq |E|$. The following steps are used to construct a Boolean network from G.

GRAPH_TO_NETWORK(G):

1. *ASSIGN_DIRECTION*(G).
2. Associate each vertex $v \in V$ a node n_v.
3. Associate each edge $e \in E$ with a node i_e.
4. Each $i_{(x,y)}$ is assigned an inverter function. $i_{(x,y)} = \overline{n_x}$.
5. If a node n_v has no incoming arcs, it is a primary input.
6. Otherwise, let $n_v = \prod_{(u,v) \in E} i_{(u,v)}$ (AND function).

It is easy to see how the construction can be accomplished in polynomial time, because the first step is polynomial by Lemma 5.3.1 and each of the remaining steps is also polynomial.

The correspondence between solutions of SIMPLE MAX CUT and PHASE ASSIGNMENT is established as follows. Let N be the set of nodes, given by the PHASE ASSIGNMENT algorithm, to be replaced by their duals. V_1 is associated to F by

$$V_1 = \{v | n_v \in N\}.$$

Now, it is essential to show that there is a partition V_1 and V_2 of V with a number of edges between V_1 and V_2 being no less than K if and only if there is a set of functions N which, when replaced by their duals, result in a network with no more than L inverters, where $L = |E| - K$.

If there is a partition V_1 and V_2 of V with K crossing edges, then the dual replacement of all the nodes corresponding to vertices in V_1 will result in a network with $|E| - K$ inverters. The reason is quite simple. For each edge $(u, v) \in E$, there is a corresponding inverter $i_{(u,v)}$ in the network. This inverter can be eliminated if and only if exactly one of the nodes, n_u or n_v, is replaced by its dual, i.e. the edge (u, v) is a crossing edge. On the other hand, if (u, v) is not a crossing edge, then either both nodes n_u and n_v or none of them are replaced by their duals. So the inverter corresponding to (u, v) remains. Thus, the number of crossing edges, K, corresponds to the number of inverters being eliminated, which implies that the number of inverters left in the network is $|E| - K$.

If there is a phase assignment resulting in a network with L inverters, then let V_1 contain all the vertices whose corresponding nodes are replaced by their duals, and let V_2 contain the rest of the vertices. The resulting partition will have no more than $|E| - L$ crossing edges. The proof is again by establishing a one-to-one correspondence between a crossing edge and an eliminated inverter.

Now, we have established a one-to-one correspondence between a solution of SIMPLE MAX CUT and that of the corresponding PHASE ASSIGNMENT, i.e. if there is a partition with at least K crossing edges, then there is a phase assignment with at most $L = |E| - K$ inverters. ■

5.3.2 A Mathematical Programming Formulation

The phase assignment problem has a natural mathematical programming formulation: a 0-1 integer programming problem with linear objective function, and non-linear constraints.

The mathematical programming problem is setup as follows: Each of the primary inputs and internal signals is associated with a binary variable. A value of 1 implies the presence of an inverter to provide the negative phase of the signal. Each internal node is associated with an additional binary variable whose value being 1 implies that the node is to be replaced by its dual. Obviously, the objective is to minimize the sum of all the variables associated with the signals.

The constraints are used to ensure the consistency of the network. Let f be a primary input or an internal node. $NFO(f)$ and $PFO(f)$ partition the fanouts of f, $FO(f)$, as indicated in Figure 5.5. $NFO(f)$ contains all the fanouts of f which presently depend on the negative phase of f. $PFO(f)$ contains all the fanouts of f which presently depend on the positive phase of f. The inverter can be eliminated if all the nodes in $NFO(f)$ are negated. In addition, if f is an internal node, the inverter can also be eliminated by negating all nodes in $PFO(f)$ and f. These are the basis for deriving the constraints. It is worth mentioning that at the presence of the inverter, the constraint associated with f should be automatically satisfied simply because both phases of f are available for the fanouts of f to use.

Formally, let η be a phase-consistent Boolean network. Two sets of binary vari-

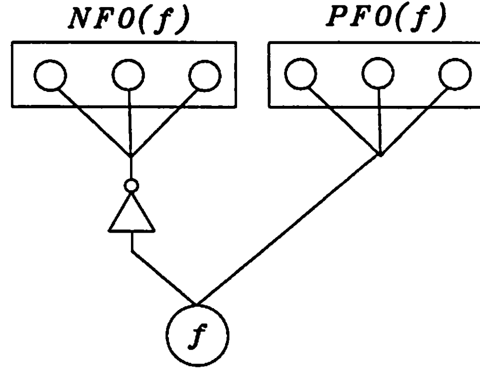


Figure 5.5: Partition of $FANOUTS(f)$ into $NFO(f)$ and$PFO(f)$

ables, X and Y, are created.

$$X(\eta) = \{x_s \in \{0,1\} | s \in SIGNALS(\eta)\}$$

$$Y(\eta) = \{y_n \in \{0,1\} | n \in NODES(\eta)\}.$$

Value of 1 for x_s means an inverter is needed to provide the opposite phase of signal s, 0 means no inverter is needed. Value of 1 for y_n means that node n is to be replaced by its dual, d_n. The objective function is

$$\sum_{s \in SIGNALS(\eta)} x_s.$$

Each signal $s \in SIGNALS(\eta)$ yields a constraint C_s

$$C_s = \begin{cases} (1 - x_s)(\sum_{t \in FO(s)} z_s^t + N_s y_f - N_s) = 0 & f \notin PI(\eta) \\ (1 - x_s)(\sum_{t \in FO(s)} z_s^t - N_s) = 0 & s \in PI(\eta) \end{cases}$$

where $N_s = |FO(s)|$ is the number of fanouts of s, f is the node generating signal s, and z_s^t is defined as

$$z_s^t = \begin{cases} 1 - y_t & y \in PFO(s) \\ y_t & y \in NFO(s) \end{cases} \tag{5.1}$$

We use an example to illustrate how the constraints are derived. Figure 5.6 shows part of a Boolean network where s is the signal generated by node f. o, p, q, and r are the fanouts of f. o and p depend on the negative phase of f. q and r depend on the positive phase of f. To eliminate the inverter, we need to either negate o and p and keep the current phase of f, q, and r, i.e.

$$y_o + y_p + (1 - y_q) + (1 - y_r) = N_f \textit{ and } y_f = 0$$

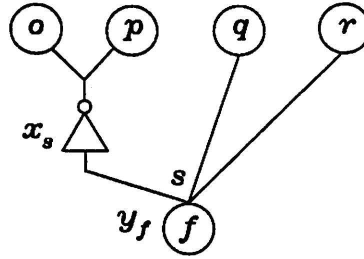


Figure 5.6: Associate constraint variables with nodes and inverters

or keep the current phase of o and p and negate f, q, and r, i.e.

$$y_o + y_p + (1 - y_q) + (1 - y_r) = 0 \text{ and } y_f = 1.$$

Combining the above two constraints, we get

$$y_o + y_p + (1 - y_q) + (1 - y_r) = (1 - y_f)N_f,$$

which is

$$y_o + y_p + (1 - y_q) + (1 - y_r) + N_f y_f - N_f = 0.$$

Since the constraint exists only if the inverter is to be eliminated, the constraint should be multiplied by $1 - x_s$, i.e.

$$(1 - x_s)(y_o + y_p + (1 - y_q) + (1 - y_r) + N_f y_f - N_f) = 0.$$

When the variables defined as in equation 5.1 are used,

$$(1 - x_s)(z_f^o + z_f^p + z_f^q + z_f^r + N_f y_f - N_f) = 0.$$

5.3.3 Exact Solution on Trees

For networks with certain special structure, it is possible to derive an efficient algorithm for solving the phase assignment problem optimally. In particular, a dynamic programming algorithm will be presented in this section to find the optimum phase assignment for networks which are trees.

First, let's look at a simple example to see why a simple greedy algorithm fails to yield the optimum solution. In Figure 5.7, (a) is a phase-consistent tree Boolean network with five nodes and five inverters. Any greedy algorithm stops after the dual-replacement

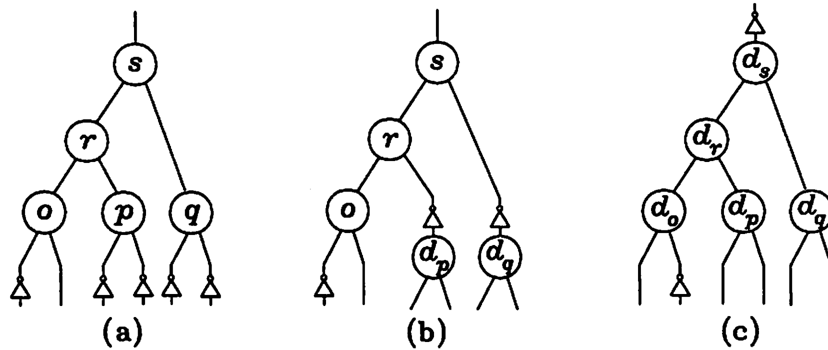


Figure 5.7: Greedy algorithm falls into local minimum (d)

of nodes p and q, leaving the network, (b), with three inverters. However, if all nodes are dual-replaced, the network, (c), would have only two inverters, an optimum solution. To achieve this optimum configuration by a sequence of dual replacements, some of the dual replacements must be allowed to increase the number of inverters temporarily.

Additional definitions are needed to describe a dynamic programming algorithm.

Given a tree network η, let $ROOT(\eta)$ be the root node of η. Let η_f be the network rooted at node f. Let $NI(\eta)$ be the number of inverters in the network η. A phase assignment of η is given by function $P(\eta) : NODES(\eta) \longrightarrow \{0,1\}$, i.e.

$$P(\eta) = \{(n,b)|n \in \eta \text{ and } b \in \{0,1\}\}$$

where a node is assigned a 1 if it is replaced by its dual, and is assigned 0 otherwise.

The algorithm has two major components, $OPA_I(\eta)$ finds the optimum phase assignment of η with an inverter in front of the root node $ROOT(\eta)$, $OPA_N(\eta)$ finds the optimum phase assignment with no inverter in front of $ROOT(\eta)$. The two components are described as two co-routines: each one calls itself or the other with a smaller tree, and terminates at the primary inputs of η.

```
OPA_N(η)
   P = {(ROOT(η), 0)}
   for each f ∈ FI(ROOT(η)) and f ∉ PI(η)
      if ROOT(η) ∈ PFO(f)
         if NI(OPA_N(η_f)) < NI(OPA_I(η_f))
            P = P ∪ OPA_N(η_f)
         else
            P = P ∪ OPA_I(η_f)
```

```
    else
        if NI(OPA_N(η_f)) + 1 < NI(OPA_I(η_f)) - 1
            P = P ∪ OPA_N(η_f)
        else
            P = P ∪ OPA_I(η_f)
return P
```

OPA_I is very similar to *OPA_N*. It assigns "1" to the root and inverts the dependencies of the root on its fanins.

```
OPA_N(η)
P = {(ROOT(η), 1)}
for each f ∈ FANINS(ROOT(η)) and f ∉ PI(η)
    if ROOT(η) ∈ NFO(f)
        if NI(OPA_N(η_f)) < NI(OPA_I(η_f))
            P = P ∪ OPA_N(η_f)
        else
            P = P ∪ OPA_I(η_f)
    else
        if NI(OPA_N(η_f)) + 1 < NI(OPA_I(η_f)) - 1
            P = P ∪ OPA_N(η_f)
        else
            P = P ∪ OPA_I(η_f)
return P
```

Now, to find the optimum phase assignment of a tree network, algorithm *OPA* simply evaluates the results of *OPA_N* and *OPA_I* and picks the better one.

```
OPA(η)
if NI(OPA_N(η)) < NI(OPA_I(η))
    return OPA_N(η)
else
    return OPA_I(η)
```

Figure 5.8 helps explaining *OPA_N*. f is the current root with two fanins g

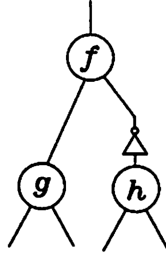


Figure 5.8: Finding $OPA_N(\eta_f)$ and $OPA - P(\eta_f)$

and h, and depends on the positive phase of g and the negative phase of h. To find $OPA_N(\eta_f)$, f must not be flipped. Since f depends on the positive phase of g, the optimum phase assignment of η_g should be used. This is found by comparing $NI(OPA_N(\eta_g))$ with $NI(OPA_I(\eta_g))$. If f depends on the negative phase of an input, e.g. h, then the number of inverters from this branch is either $NI(OPA_N(\eta_h)) + 1$ or $NI(OPA_I(\eta_h)) - 1$. This justifies the last 'if' statement in the algorithm.

It is a dynamic programming algorithm because it uses the principle of optimality and assumes that at each step the optimum solutions of sub-problems have already been found.

THEOREM 5.3.3 *The algorithm $OPA_N(\eta)$ finds the optimum phase assignment without an inverter at $ROOT(\eta)$. The algorithm $OPA_I(\eta)$ finds the optimum phase assignment with an inverter at $ROOT(\eta)$.*

Proof. By induction on the height of the tree network. If the network η is of height one, then there is no other choice than not flipping the node in OPA_N and flipping the node in OPA_I. The algorithms work correctly in this case.

If the network η is of height n, let $f \in FI(ROOT(\eta))$, then the input network $\eta(f)$ is of height less than n. By the induction hypothesis, OPA_I and OPA_N find optimum phase assignment of $\eta(f)$ with and without inverters at the root of $\eta(f)$. The algorithm will choose either $OPA_I(\eta(f))$ or $OPA_N(\eta(f))$ depending on which one results in less inverters. Since η is a tree, the phase assignment of $\eta(f)$ is independent of all other $\eta(g)$, $g \in FI(ROOT(\eta))$. Therefore, the result is optimum. ∎

COROLLARY 5.3.4 *Algorithm OPA finds the optimum phase assignment for tree networks.*

Proof. Obvious from Theorem 5.3.3 ∎

The algorithm is also very efficient. In fact, we have

THEOREM 5.3.5 *Algorithm OPA is $O(N)$ where N is the number of nodes in the tree network.*

Proof. Each node in the network is processed exactly twice, one by OPA_N and one by OPA_I. Each time, a fixed amount of operations is performed. In particular, since the phase assignment of sub-trees is independent of each other, the union operation in the algorithm OPA_N and OPA_I can be done in constant time. ∎

5.4 Heuristic Algorithms

Since the phase assignment problem is NP-complete, we have to resort to heuristic algorithms. One operation used repeatedly in the heuristic algorithms is to compute *inverter savings* which is the number of inverters saved by flipping a node. Based on this operation, a greedy algorithm can be easily derived. To avoid falling into a local minimum far removed from the global minimum, the greedy algorithm is improved to allow partial hill-climbing to find better solutions at the expense of more computing time. These algorithms can be further improved by allowing look-a-head in selecting nodes to be processed. Each of the above steps are described in details in the following subsections.

5.4.1 Computing Inverter Saving

Inverter Saving of a node f is defined as the number of inverters saved if the node f is replaced by its dual d_f. Correctly computing Inverter Savings of nodes is essential in the heuristic algorithms described later on. Figure 5.9 shows what is involved in computing Inverter Saving of node f.

There are two sets of nodes associated with the fanouts of node f, $NFO(f)$ containing all nodes $n \in FO(f)$ which depend on the negative phase of f, $PFO(f)$ containing all nodes $n \in FO(f)$ which depend on the positive phase of f. A node n belongs to both groups $NFO(f)$ and $PFO(f)$ if n depends on both phases of f.

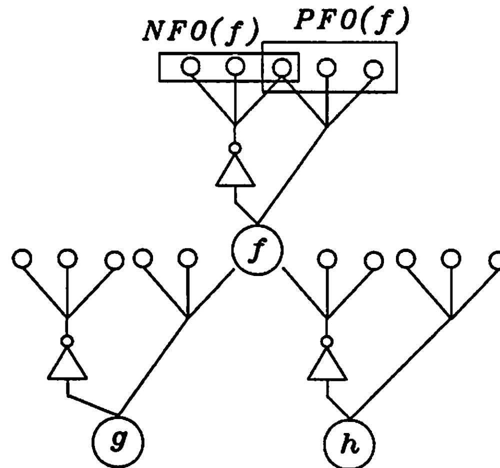


Figure 5.9: Computing inverter savings of flipping f

The inverter savings of f in the figure has two components, savings at the output of f and savings at each input of f. When replacing f by its dual d_f, an inverter can be saved at the output of f if and only if all outputs of f depend on the negative phase of f and no output of f depends on the positive phase of f, i.e.

$$NFO(f) = FO(f) \text{ and } PFO(f) = \phi.$$

An inverter has to be created at the output of f if and only all outputs of f depend on the positive phase of f and no output of f depends on the negative phase of f, i.e.

$$PFO(f) = FO(f) \text{ and } NFO(f) = \phi.$$

At an input x of f, an inverter can be saved if and only f is the only node in $FO(x)$ that depends on the negative phase of x and f does not depend on the positive phase of x, i.e.

$$NFO(x) = \{f\} \text{ and } PFO(x) = FO(x) - \{f\}.$$

An inverter has to be created if and only if no output of x depends on the negative phase of x, i.e.

$$NFO(x) = \phi.$$

The procedure $OS(f)$ computes the inverter savings at the output of f if f is replaced by its dual d_f.

$OS(f)$

```
    if NFO(f) = FO(f) and PFO(f) = φ
        saving = 1
    else if PFO(f) = FO(f) and NFO(f) = φ.
        saving = −1
    else
        saving = 0
    return saving
```

The procedure $IS(f, x)$ computes the inverter savings at input x of f if f is replaced by its dual d_f.

```
IS(f,x)
    if NFO(x) = {f} and PFO(x) = FO(x) − {f}
        saving = 1
    else if NFO(x) = φ
        saving = −1
    else
        saving = 0
    return saving
```

Using the two procedures above, the inverter savings of f, when it is replaced by its dual d_f, can be computed quite easily:

```
S(f)
    saving = OS(f)
    for each x ∈ FI(f)
        saving = saving + IS(f,x)
    return saving
```

5.4.2 Incremental Update of Inverter Saving

Inverter Saving is used throughout our heuristic algorithms selecting nodes to flip. It is too expensive to compute the Inverter Saving repeatedly. In fact, inverter savings of

nodes need to be computed only once at the beginning. When a node is flipped, only the inverter savings of certain nodes around it need to be re-computed. It is essential to be able to update incrementally Inverter Savings after each flip of a node.

First of all, when a node is flipped, it is important to find out all the other nodes whose inverter savings could be affected. The example in Figure 5.10 illustrates some of the possibilities. When f is flipped, node o is affected because one of its input inverter savings

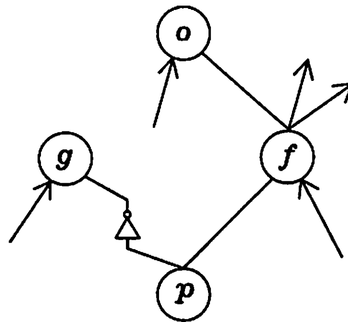


Figure 5.10: Nodes affected when f is flipped

$IS(o, f)$ is changed from -1 to 0. Node p is also effected because its output inverter saving $OS(p)$ is changed from 0 to 1. Furthermore, node g is affected because one of its input Inverter Saving $IS(g, p)$ is changed from 1 to 0. So, if node f is flipped, the nodes which need to be updated are all of its fanouts, all of its fanins, and all the fanouts of the fanins, i.e. the following set of nodes

$$AFFECTED(f) = FO(f) \cup FI(f) \bigcup_{x \in FI(f)} FO(x)$$

Since Inverter Saving of a node f depends on its fanouts, fanins, and fanouts of the fanins, $AFFECTED(f)$ are all the nodes whose Inverter Savings could possibly be changed. In addition, it is not even necessary to re-compute Inverter Savings of all the nodes in $AFFECTED(f)$ from scratch. In the above example, Inverter Saving of node o is changed due to the changes at its input branch from f. So, it can be updated by subtracting $IS(o, f)$ and then adding $IS(o, d_f)$. All the nodes in $AFFECTED(f)$ can be updated in a similar way. The procedure for doing so is called $FLIP_AND_UPDATE$. It replaces node f by its dual d_f and updates incrementally Inverter Savings of all the affected nodes.

$FLIP_AND_UPDATE(f)$

```
for each g ∈ FO(f)
    S(g) = S(g) − IS(g, f)
for each g ∈ FI(f)
    S(g) = S(g) − OS(g)
    for each h ∈ FO(g)
        S(h) = S(h) − IS(h, g)
replace f by d_f
S(d_f) = −S(f)
for each g ∈ FO(d_f)
    S(g) = S(g) + IS(g, d_f)
for each g ∈ FI(d_f)
    S(g) = S(g) + OS(g)
    for each h ∈ FO(g)
        S(h) = S(h) + IS(h, g)
```

5.4.3 Quick Phase

A greedy algorithm for phase assignment consists of flipping the node with largest inverter savings until all nodes in the given network have zero or negative inverter savings. Because the algorithm runs very fast when compared to other algorithms, it is named **QUICK_PHASE.**

```
QUICK_PHASE(η)
    for each node f ∈ NODES(η)
        compute S(f)
    repeat {
        f = argmax_{x∈NODES(η)} {S(x)}
        if S(f) ≤ 0 done
        FLIP_AND_UPDATE(f)
    }
```

Notice that *QUICK_PHASE* never flips a node whose Inverter Saving is zero. This is to avoid falling into an infinite loop. However, allowing flips of these nodes may lead to further inverter reduction. Algorithm *QUICK_PHASE′* explores this idea by allowing nodes with zero Inverter Savings to flip, but once only until the inverter count is reduced again.

```
QUICK_PHASE'(η)
    for each node f ∈ η
        compute S(f)
        unmark f
    repeat {
        f = argmax_{x∈NODES(η) and unmarked}{S(x)}
        if S(f) = 0
            mark f
        else if S(f) > 0
            unmark all marked nodes in η
        else /* now S(f) < 0 */
            done
        FLIP_AND_UPDATE(f)
    }
```

5.4.4 Good Phase

The problem with both *QUICK_PHASE* and *QUICK_PHASE'* is that the cost function, the inverter count, never increases. Consequently, the results are often just local minimum and largely depend on the initial configuration of the Boolean network. In this section, another algorithm is presented which allows the cost function to increase in a limited fashion.

The idea used in *QUICK_PHASE'* is to allow a node to flip even if the cost function stays the same. But a zero-inverter-saving node can only be flipped once before a better solution is found. The generalization of the idea is to allow a node to flip even if the cost function increases. The order in which nodes are selected is again determined in a greedy fashion, i.e. selecting a node which decreases maximally or increases minimally the cost function. But, flipping of a node is restricted to only once before a better configuration is found. This algorithm resembles the ideas originated by Kernighan and Lin in solving the graph partitioning problem [31].

```
GOOD_PHASE(η)
    repeat {
        unmark all f ∈ NODES(η)
        best = η
```

```
    QUICK_PHASE(η)
    if NI(η) < NI(best)
        best = η
    repeat {
        if all f ∈ NODES(η) are marked
            return best
        else
            f = argmax_{x∈NODES(η) and unmarked} {S(s)}
            FLIP_AND_UPDATE(f)
            mark f
    } until NI(η) < NI(best)
}
```

The algorithm is named **GOOD_PHASE** because it always gives better result than *QUICK_PHASE*. *GOOD_PHASE* always keeps the best network found so far. The call to *QUICK_PHASE* initially reaches a local minimum. Then, *GOOD_PHASE* enters the hill-climbing stage by flipping nodes even though the number of inverters in the network may increase. If, at any point, a solution better than the current best is found, that solution is accepted and the whole process starts again. The algorithm terminates when all of the nodes have been flipped once and no better solution has been found. This is a partial hill-climbing scheme because each function is flipped only once in a greedy order (minimum inverter increase).

5.4.5 Look-ahead

Both *QUICK_PHASE* and *GOOD_PHASE* can be improved by choosing the order in which nodes are selected better. So far, the nodes have been chosen in a greedy way, i.e. selecting the node that maximally decreases or minimally increases the cost function. This may lead to a poor local minimum solution, as the example in Figure 5.11 shows. Using a greedy strategy, node f is chosen leading to a solution with five inverters. However, if node g is chosen instead, knowing before hand that it would lead to flipping node h with two inverter savings and that, in turn, would lead to flipping node i with one more inverter saving, the resulting circuit would have four inverters. This prompts the idea of k-step look-ahead. Instead of choosing the current best move, this strategy chooses a node to flip which, when combined with the next $k-1$ flips of the connecting nodes, leads to the best

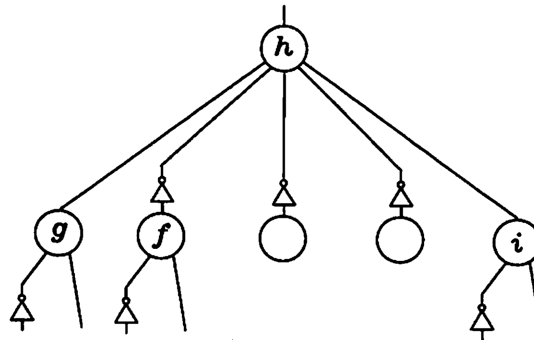


Figure 5.11: look-a-head gives better result on this circuit

overall k moves. The total number of inverters saved in these k moves will be assigned as the value of the first flipped node. Following is a procedure for computing the value, S_K, of a node, which takes a node n and a positive integer k as inputs.

```
S_K(n, k)
    saving = S(n)
    FLIP_AND_UPDATE(n)
    T = {n}
    N = FI(n) ∪ FO(n)
    repeat k − 1 times {
        f = argmax_{x∈N} {S(x)}
        saving = saving + S(f)
        FLIP_AND_UPDATE(f)
        T = T ∪ {f}
        N = N ∪ FI(f) ∪ FO(f) − T
    }
```

Now, function S_K can be used to replace function S everywhere in $QUICK_PHASE$ and $GOOD_PHASE$. k can be used to obtain desired tradeoffs between the quality of results and CPU time.

5.5 Generalized Phase Assignment

Until now, the basic step in global phase assignment is replacing a node f by its dual d_f. However, in standard-cell or gate-array design, there are manually designed cells

which have some input signals inverted internally in order to achieve better performance or smaller area. Thus, there are many other modifications to the nodes which could affect the number of inverters. For example, Figure 5.12 shows three cells found in a standard cell library. If gate A is replaced by gate B, two inverters should be added at the inputs to

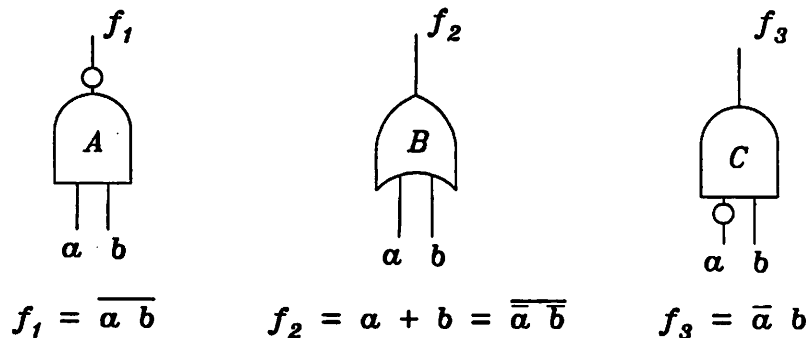


Figure 5.12: Three cells in a standard cell library

preserve the logic, because $\overline{ab} = \overline{\overline{\bar{a}\bar{b}}}$. If A is replaced by C, one inverter should be added to the output and another one should be added to the input a.

As the example shows, replacing A by either B or C is more general than simply replacing A by its dual d_A. Minimizing the number of inverters using such changes is the **generalized phase assignment** problem. The objective of this section is to define this new class of transformations formally, to show that the problem is not significantly different from the original global phase assignment problem, and to extend the heuristic algorithms to handle the generalized phase assignment problem.

Since the new modifications are possible only by allowing internal inverters of library cells, the focus of this section is to minimize the inverters for a library-based target technology. Circuits under consideration in this section are assumed to be mapped. Each node has associated with it a gate from a given library.

5.5.1 NN-class and Related Definitions

Given a gate g in a library, we would like to find out all the other gates that can be used to replace g by adding or subtracting some inverters at the inputs and output of g. What do those gates have in common? How can they be found in a given library? When replacing a gate by another, how many inverters can be saved? Let's start with the following definitions.

An *NN-transformation* of a gate g consists of negating certain inputs of g, and/or negating the output of g. For example, in Figure 5.13, gate (b) can be obtained from (a)

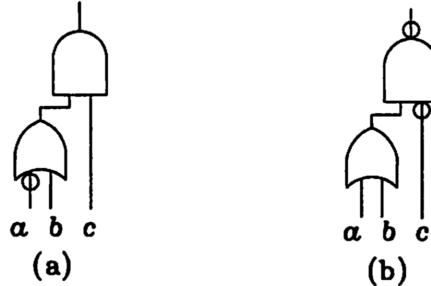


Figure 5.13: (b) can be derived from (a) by negating some inputs and output

by negating inputs a and c and negating its output.

Two gates are said to be *NN-equivalent* if one can be NN-transformed to another. We use $g\overset{nn}{\leftrightarrow}h$ to denote the NN-equivalence of gate g and h. It is trivial to see that $\overset{nn}{\leftrightarrow}$ is reflexive, symmetric, and transitive, i.e.

$$g\overset{nn}{\leftrightarrow}g$$

$$g\overset{nn}{\leftrightarrow}h \Longleftrightarrow h\overset{nn}{\leftrightarrow}g$$

$$g\overset{nn}{\leftrightarrow}h \text{ and } h\overset{nn}{\leftrightarrow}i \Longrightarrow g\overset{nn}{\leftrightarrow}i$$

So, $\overset{nn}{\leftrightarrow}$ is an equivalence relation. Cells in a standard cell library can therefore be partitioned into NN-equivalent classes, or simply NN-classes. Let $NN(g)$ denote the NN-class gate g belongs to, i.e.

$$NN(g) = \left\{h | h \in LIB(g) \text{ and } h\overset{nn}{\leftrightarrow}g\right\}$$

where $LIB(g)$ is the underlying technology library.

A cell g is a **tree** if its function in factored form has exactly one literal from each input (e.g. $a(\overline{b}+c)$ is a tree and $a\overline{b}+\overline{a}b$ is not).

Let g be a tree gate. Since $NN(g)$ is the set of all the gates NN-equivalent to g, it is possible to choose a representative function for $NN(g)$. Any function f satisfying the following condition can be chosen to be the representative of $NN(g)$.

- f is positive unate in all the inputs,
- Each input is used exactly once in f,

- The output is not complemented.

For example, the representative of $NN(a(\overline{b}+c))$ could be $a(b+c)$, or $a+bc$. Every cell in $NN(g)$ can then be expressed as the representative function plus some phase information of its inputs and output. In the previous example, if $a(b+c)$ is chosen to be the representative function, cell $a(\overline{b}+c)$ would have $phase(a(\overline{b}+c)) = 0$, $phase(a) = 0$, $phase(b) = 1$, and $phase(c) = 0$. If the representative function is $a+bc$, $a(\overline{b}+c)$ would have $phase(a(\overline{b}+c)) = 1$, $phase(a) = 1$, $phase(b) = 0$, and $phase(c) = 1$, because $a(\overline{b}+c) = \overline{\overline{a}+b\overline{c}}$. As the example shows, the phase of a cell or its input is determined by the choice of the representative function. Given a library, a preprocessing step can be performed to choose a representative function for each NN-class so that the phases of all the cells and all of their inputs are uniquely determined.

Two inputs of a cell are said to be permutable if the connections to these two inputs can be switched without changing the functionality of the circuit (e.g. an $AO22$, $f = ab + cd$, has four inputs. inputs a and b are permutable. So are inputs c and d).

Let p be an input of a gate g in a mapped circuit. Let v_p be the signal connected to the p. $phase(v_p)$ is 0 if the positive phase of v_p is used when $phase(p) = 0$. $phase(v_p)$ is 1 if the negative phase of v_p is used when $phase(p) = 0$. Notice that the phase of an input variable v_p is defined with respect to the un-inverted input p of the representative function of $NN(g)$, regardless of the current phase of the input p.

Finally, the inputs of a gate are sometimes called pins.

5.5.2 Choosing A Move

A basic step, a move, in the generalized phase assignment algorithms presented here is replacing a gate g by another gate in $NN(g)$. In the original simple phase assignment, procedure S is used to compute Inverter Saving obtained by replacing a gate by its dual. The goal of this section is to develop a procedure for computing Inverter Saving obtained by replacing a tree-gate by another gate in its NN-class. There are several reasons for restricting the attention only to those gates which are trees. First of all, fast procedures are available for computing Inverter Saving obtained by replacing a tree by another NN-equivalent tree. Second, the majority of the gates in a library are trees.

We first use an example to show what is involved in computing inverter savings in this case, to point out some of the difficulties, and to motivate the algorithm developed

later on.

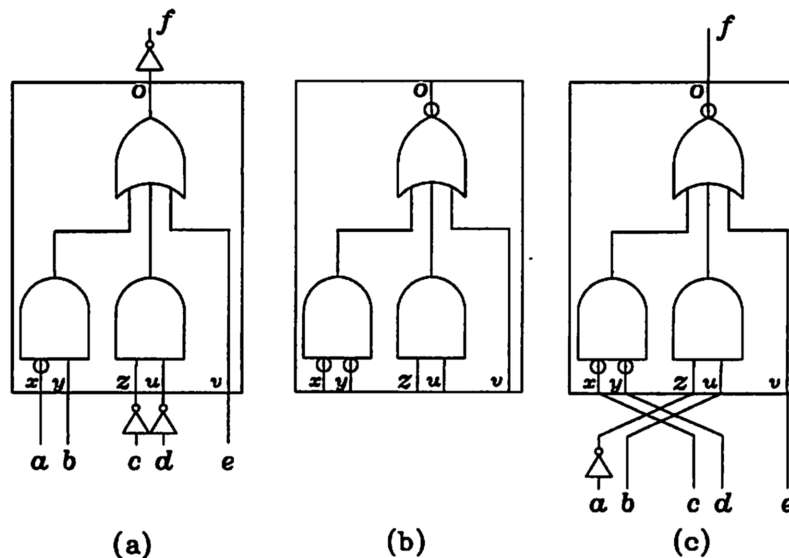


Figure 5.14: Replacing (a) by (b) to save two inverters in (c)

Part (a) of Figure 5.14 is a node of a Boolean network where $f = \overline{\overline{a}b + \overline{c}\overline{d} + e}$ is required. The gate $AO221$ with inverted input x is used. Two inverters are needed at inputs z and u and one inverter is needed at the output. Part (b) is an NN-equivalent gate $AOI221$ with two inverted inputs, x and y. If the gate in (a) is replaced by the gate in (b), two inverters can be saved. Part (c) shows the new connection.

Several simple facts are used in deriving the new configuration. Pin x and y are permutable, and so are z and u. In addition, the pin group $\{x, y\}$ and $\{z, u\}$ are permutable. For each set of permutable pins (e.g. x and y), there is a problem: finding the permutation which saves the most inverters. A similar problem exists in finding a permutation of pin groups. If cells are allowed to have arbitrary levels of logic, there is a hierarchy of the permutability information about the pins. Figure 5.15 shows a gate with three levels of logic. Each pair of inputs to the first level AND gates are permutable. In addition, $\{a, b\}$ is permutable with $\{c, d\}$ and $\{e, f\}$ is permutable with $\{g, h\}$. Finally, pin groups $\{a, b, c, d\}$ and $\{e, f, g, h\}$ are permutable.

To describe the algorithm formally , additional definitions are needed. A **level-0 permutable pin group** contains two or more permutable pins. A **level-n permutable pin group**, level-n group for short, is a set containing two or more level-$n-1$ permutable pin

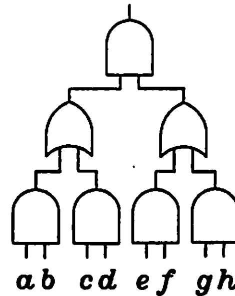


Figure 5.15: Hierarchy of permutable pins and pin groups

groups. In Figure 5.15, $\{a,b\}$ is a level-0 group, $\{\{a,b\},\{c,d\}\}$ is a level-1 group, and $\{\{\{a,b\},\{c,d\}\},\{\{e,f\},\{g,h\}\}\}$ is a level-2 group.

There is another way of representing the permutability information. The internal of a tree-gate can be viewed as a tree of AND and OR nodes. Two inputs of an internal node are permutable if the sub-trees rooted at these two inputs are isomorphic. So, the inputs of each internal node can be partitioned into permutable groups. For the example in Figure 5.15, all the inputs of all the internal nodes are permutable (this is not the case in general).

The procedure $IIC(A,B)$ (Input Inverter Count) determines how many inverters are needed at the inputs if a gate A is replaced by an NN-equivalent gate B. The input Inverter Saving obtained by replacing A by B can be computed by subtracting the current number of inverters from $IIC(A,B)$. Since the permutability information is arranged in a strictly hierarchical fashion, i.e. each internal node has permutable inputs which in turn have their own permutable inputs, it is natural to describe the procedure recursively.

$IIC(A,B)$
    if A and B are pins
        return $\overline{phase(v_A) \oplus phase(B)}$
    $count = 0$
    for each permutable input group G_A of $ROOT(A)$
        let G_B be the corresponding group of $ROOT(B)$
        create a bipartite graph $K = (S,T,E)$
        for each node $g_A \in G_A$
            create a node $n_A \in S$
            for each node $g_B \in G_B$
                create a node $n_B \in T$
                create an edge $(n_A, n_B) \in E$

```
            let w(n_A, n_B) = IIC(TREE(g_A), TREE(g_B))
        }
    }
    let M be the minimum weighted complete matching of K
    let count = count + Σ_{e∈M} w(e)
}
return count
```

$IIC(A, B)$ is a recursive procedure. It stops when A and B are pins. An inverter is needed at the new pin B if and only if the exclusive-nor of $phase(v_A)$ and $phase(B)$ is 1. If A and B are internal nodes, the inputs of root nodes of A and B, $ROOT(A)$ and $ROOT(B)$ are partitioned into permutable groups. The total input inverter count is the sum of the inverter count from each group, because the inputs of the groups are mutually disjoint. For any single group G_A and its corresponding G_B, any subtree rooted at $g_A \in G_A$ can be replaced by a subtree rooted at $g_B \in G_B$. So, the bipartite graph K is complete. The weight of an edge (n_A, n_B) is the number of inverters needed at the input of tree $TREE(g_B)$, if g_A is replaced by g_B. The minimum inverter permutation is therefore given by the minimum weighted complete matching of K.

Now, IIC can be used to compute the Inverter Saving obtained by replacing a gate A with an NN-equivalent gate B. The procedure is called GS (Generalized Saving).

```
GS(A, B)
    saving = output_saving(A, B)
    saving = saving + IIC(A, A) − IIC(A, B)
    return saving
```

The first step of the procedure is to determine the inverter savings at the output. This can be computed quite easily knowing the relative phase of the outputs of A and B and how the output is used. The next step is to determine the inverter savings at the inputs. It is the difference of current inverter count $IIC(A, A)$ and new inverter count $IIC(A, B)$. Notice that the current inverter count can not be obtained from the current usage of A directly because it may be connected in a non-optimum way. However, by replacing A by itself in $IIC(A, A)$, the optimum usage of A and its associated inverter count can be found.

The Matching Problem

The matching problem in *IIC* is to find a complete matching with minimum total weight. It can be transformed to the usual maximum-weighted-bipartite-matching problem.

Let $K = (S, T, E)$ be an instance of the graphs created in *IIC*. A new bipartite graph K' is created by copying K. Then, $w(e')$ is changed to $-w(e')$ for all $e' \in E'$.

PROPOSITION 5.5.1 *If M' is the maximum weighted matching of K', then its corresponding M is the minimum weighted complete matching of K.*

Proof. Obvious. ∎

Because the graph K' is complete, it is not necessary to solve the matching problem using general weighted matching algorithms. In fact, the maximum weighted bipartite matching of K' is the Gale-Shaply matching which can be solved in $O(n^2)$ time [24].

5.5.3 Generalization of Heuristic Algorithms

The heuristic algorithms *QUICK_PHASE* and *GOOD_PHASE* can be generalized to accommodate new moves, i.e. replacing a gate by an NN-equivalent gate. The basic step of both algorithms is choose a node n, choose a gate g in $NN(n)$ and replace n with g. To evaluate each move, a generalization of S, called *SAVING*, is proposed.

```
SAVING(n)
   if n is not a tree
      saving = S(n)
   else
      saving = max_{m∈NN(n)} {GS(n,m)}
   return saving
```

If n is not a tree gate, *SAVING* uses the old S to compute the inverter savings. If n is a tree gate, *SAVING* searches through all gates in $NN(n)$ to find one with maximum inverter savings.

With *SAVING*, the generalized Quick Phase and Good Phase algorithms can be described.

```
GENERAL_QUICK_PHASE(η)
   repeat {
      f = argmax_{x∈NODES(η)} {SAVING(x)}
      if SAVING(f) ≤ 0 done
      REPLACE(f, BEST(NN(f))
   }
```

In the procedure, $BEST(NN(f))$ returns a gate in $NN(f)$ which, when replacing gate f, maximally reduces the number of surrounding inverters. $REPLACE$ does the actual replacement.

$GOOD_PHASE$ is extended similarly to $GENERAL_GOOD_PHASE$.

```
GENERAL_GOOD_PHASE(η)
   unmark f for all f ∈ NODES(η)
   repeat {
      best = η
      GENERAL_QUICK_PHASE(η)
      if NI(η) < NI(best)
         best = η
      repeat {
         if all f ∈ NODES(η) are marked
            return best
         else
            f = argmax_{x∈NODES(η) and x is unmarked} {SAVING(s)}
            REPLACE(f, BEST(NN(f)))
      } until NI(η) < NI(best)
   }
```

5.6 Experiments and Results

The heuristic algorithms presented in the previous subsection for solving the global phase assignment problem have been implemented and tested on a large set of examples. The objectives of these experiments are:

1. To show the speed of *QUICK_PHASE* and *GOOD_PHASE* and the quality of results obtained by the algorithms.

2. To compare the relative speed and result quality of the heuristic algorithms.

3. To explore the possibility and the effect of having random starting points.

The examples are taken from the MIS benchmark set consisting of over one hundred circuits. They are either from the MCNC logic synthesis workshop benchmark set, or actual circuits from industry. The examples are optimized by MIS using an algebraic script prior to phase assignment.

The first experiment is to compare the quality and run time of *QUICK_PHASE* and *GOOD_PHASE*. Table 5.1 shows the results of *QUICK_PHASE* and *GOOD_PHASE*. For compactness, only the examples with more than thirty inverter savings are selected. The second column is the number of nodes in the circuits excluding inverters. The results show that Inverter Savings obtained by either *QUICK_PHASE* and *GOOD_PHASE* are significant in the total cost of the circuits. The next two columns and the last two columns show the Inverter Savings obtained by *QUICK_PHASE* and *GOOD_PHASE* and their CPU time in seconds on a DEC μVAX-III which is a three MIPS machine. The table shows that *GOOD_PHASE* generally spends about 30% more cpu time, but always achieves as good or better results and can sometimes obtain much better results than *QUICK_PHASE* does, as indicated by example *alui* and *merge*.

To see closely the deficiencies of *QUICK_PHASE*, Figure 5.16 is used to show how the inverter count reduces as *GOOD_PHASE* flips the nodes in example *merge*. The horizontal axis is the number of flips made by the algorithm and the vertical axis is the inverter count. One can imagine the curve growing from left to right as the algorithm proceeds. It is clear that *QUICK_PHASE* would have stopped at 170 inverters. Even the modified *QUICK_PHASE'* would have stopped at 166 inverters. Since *GOOD_PHASE* allows partial hill-climbing, it is able to find a much better solution. Figure 5.17 shows the entire process followed by *GOOD_PHASE*.

To explore the possibility and effect of using random starting points in conjunction with *QUICK_PHASE*, the following algorithm is designed which combines random starting points with *QUICK_PHASE*.

RANDOM_GREEDY(η, n)

		QUICK_PHASE		*GOOD_PHASE*	
examples	no. nodes	saving	time	saving	time
alui	161	109	5.2	152	10.0
aluii	113	68	5.4	69	6.2
amux	32	100	3.6	100	3.9
bfmt	39	11	3.9	40	5.1
bmux	36	132	10.0	132	10.5
brngen	40	88	3.0	106	2.8
cfeed	86	69	2.7	69	3.1
count4	80	45	1.9	45	2.2
gprdec	224	58	4.5	64	6.3
maskgn	93	27	2.2	42	3.5
merge	106	68	9.9	200	13.6
pcfmux	43	73	3.3	73	3.6
rbusmx	34	150	8.6	150	9.2
rotate	104	70	7.0	72	7.0
s1mux	40	37	2.4	53	2.7
smux	32	100	3.7	100	4.0
ampdv	161	67	7.1	75	9.7
ampiod	92	50	5.0	56	9.3
ampms	231	101	12.8	112	18.9
amprx	88	41	2.9	42	3.4
amptmm	212	94	13.1	94	15.1
ampxhd	121	52	5.5	70	7.4
txvlsi	120	58	4.5	64	5.6
9sym	55	39	9.2	41	3.8
alupla	56	33	1.8	45	2.5
duke2	113	53	4.1	56	5.3
misex3	162	81	8.4	91	11.0
misex3c	133	82	17.5	85	21.5

Table 5.1: Comparison of *QUICK_PHASE* and *GOOD_PHASE*

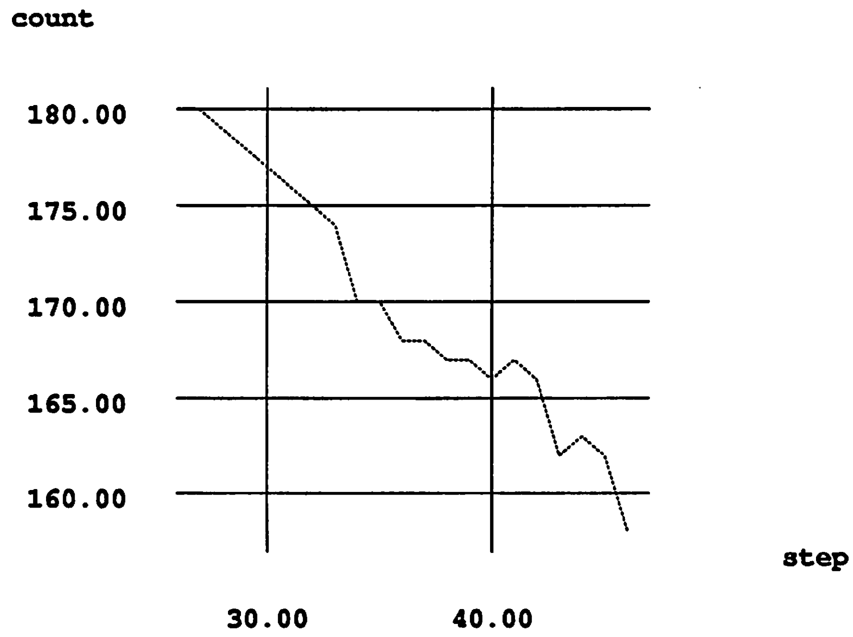


Figure 5.16: Deficiency of *QUICK_PHASE* and *QUICK_PHASE'*

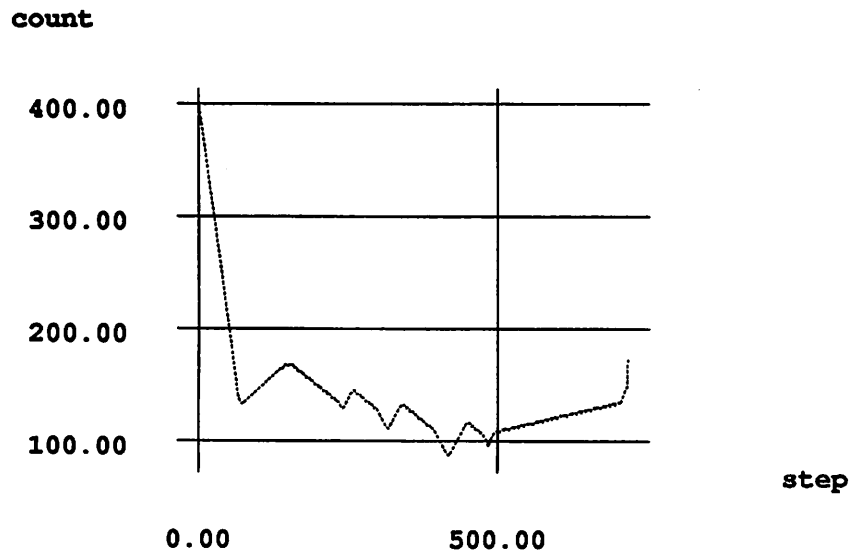


Figure 5.17: Partial hill-climbing of *GOOD_PHASE*

```
best = η
for i = 1 to n {
    RANDOM_ASSIGN(η)
    QUICK_PHASE(η)
    if η is better than best
        best = η
}
return best
```

The algorithm takes a network and a number n, and generates n random starting points. Each time, it uses the greedy algorithm to get to a local minimum. The best assignment is returned at the end.

The next experiment compares the quality of the results and CPU time of *GOOD_PHASE* and *RANDOM_GREEDY* with fifty random starting points. Figure 5.18 is the scatter plot of inverter savings of *GOOD_PHASE* versus *RANDOM_GREEDY* on all the benchmark examples. Almost all the points are below the 45-degree line. This shows that one path of *GOOD_PHASE* almost always gives better results than that of fifty *RANDOM_GREEDY*. Furthermore, Figure 5.19 compares the CPU time of *GOOD_PHASE*

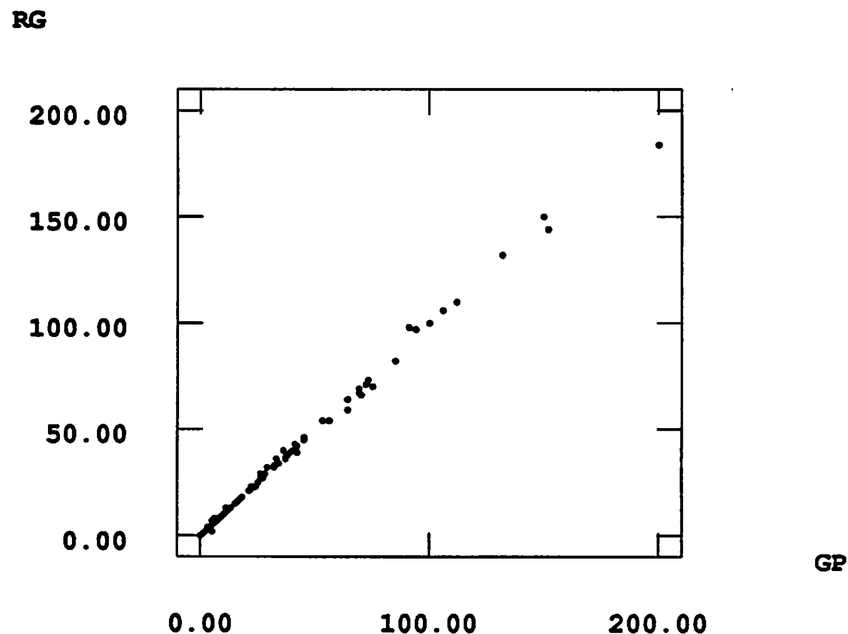


Figure 5.18: Quality of *Random_Greedy* vs. *Good_Phase*

and *RANDOM_GREEDY*. All the points are way above the 45-degree line, which shows that *GOOD_PHASE* takes less time on all the examples. Finally, let us take a closer

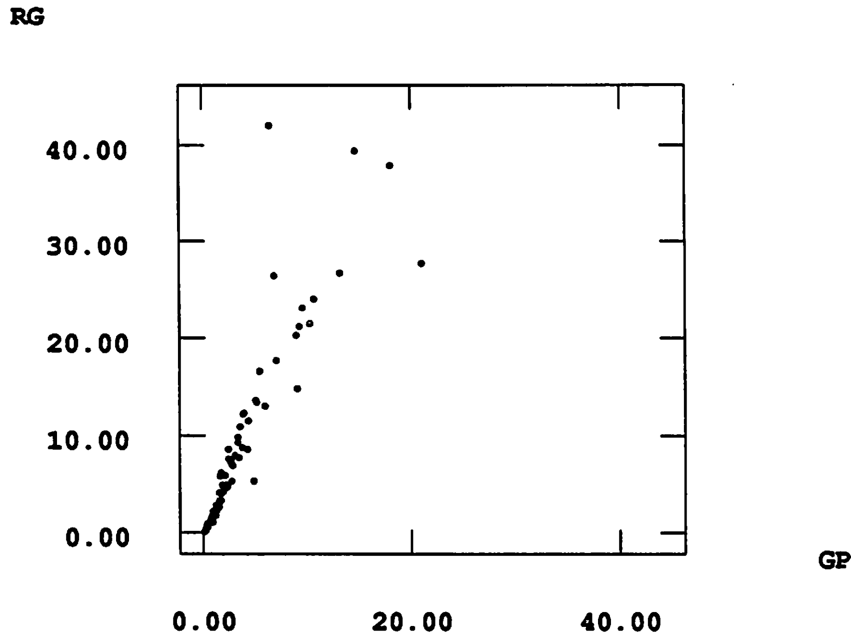


Figure 5.19: CPU time of *Random_Greedy* vs. *Good_Phase*

look at one of the examples. In figure 5.20, the line on the bottom is the result given by *GOOD_PHASE*. The line on the top shows the results of each random-greedy assignment, clustering within a certain region. This example shows that hill-climbing is essential in achieving good results for the phase assignment problem, and *GOOD_PHASE* is very cost-effective.

5.7 Future Work

In addition to the heuristic algorithms presented in this chapter, several other algorithms are applicable to the global phase assignment problem. Given that inverter savings are updated incrementally, simulated annealing is a viable alternative. Several variations of the Kernighan and Lin algorithm are also applicable here. Recently, Daniel Brand [8] proposed another probabilistic hill-climbing algorithm. It should be interesting to apply the algorithm to the phase assignment problem.

The generalized phase assignment problem is fairly new. The treatment presented

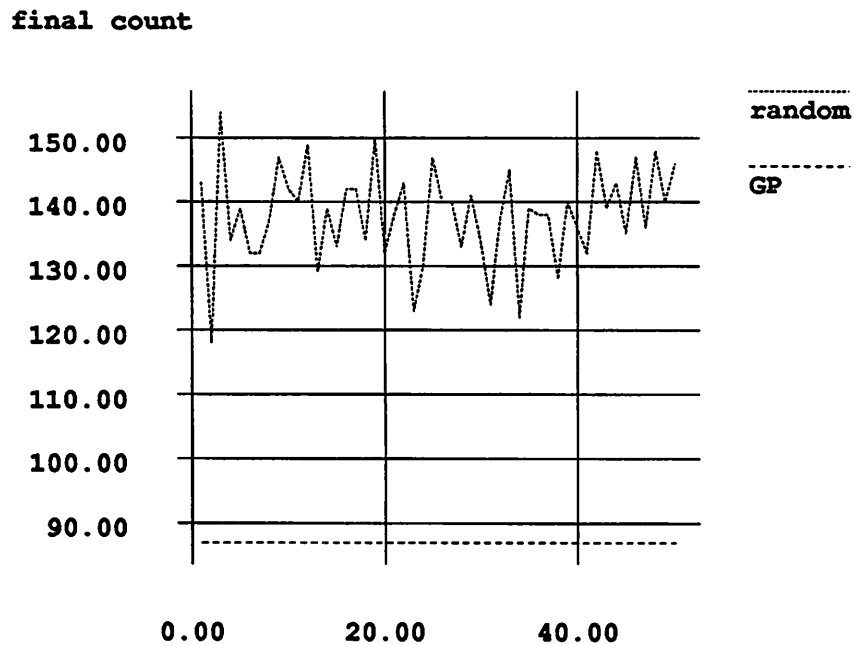


Figure 5.20: Quality of *Random_Greedy* vs. *Good_Phase*

in this chapter is preliminary. More work remains to be done. First of all, the generalized phase assignment algorithm has yet to be implemented. This requires close interaction with technology mapping and a new interface to the technology library. Given a standard-cell library, the gates in it need to be grouped into NN-classes. It should be a fairly straightforward exercise to check for NN-equivalence of the gates given that the cells are, in general, quite small, and, in addition, only NN-classes of tree-cells are used in the algorithm.

Chapter 6

Timing Optimization

6.1 Introduction

Being able to meet performance requirements is absolutely essential in synthesizing digital logic circuits. As the circuit complexity increases, many of the manual methods for performance improvement have become impractical. Automatic performance optimization of digital circuits has played and will be playing a more and more important role in any synthesis system. Such performance optimization systems must be able to work with different levels of circuit hierarchy and at various steps of the design process (e.g. re-timing, reducing delay in combinational logic, delay-driven layout, etc.). This chapter deals exclusively with performance optimization of combinational logic circuits (timing optimization). The results can be used as a component in a performance driven synthesis system.

Timing optimization of combinational circuits can be viewed as a three-phase process. In the first phase, circuits are globally restructured to have better "timing properties". As a simple example, Figure 6.1 shows two equivalent circuits. If the arrival times of all the inputs are the same, circuit (b) is preferred over circuit (a) for it reduces the output arrival time. On the other hand, if input u is the critical signal, circuit (a) becomes superior. It is evident from this example that even though the two circuits have the same area, one is better than the other when speed is important. Here, the quality of the circuits is judged not by the detailed timing figures, but rather, by their structure. A more sophisticated example of global restructuring is the conversion from a carry-ripple adder to a carry-look-ahead adder. This phase is characterized by its independence of the target technology. The objective here is to look for global structural changes of circuits to achieve delay reductions

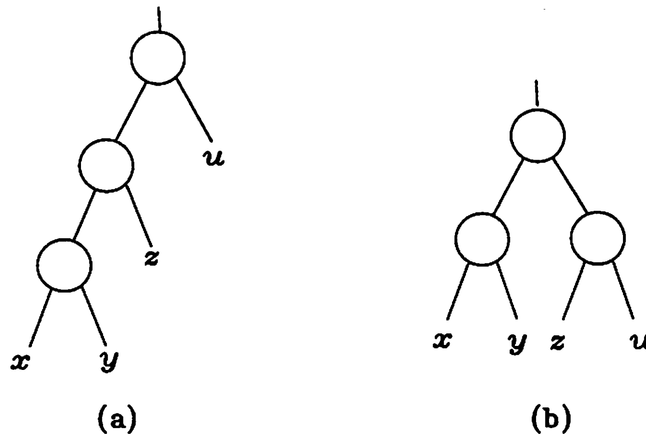


Figure 6.1: Equivalent circuits with different timing property

that can not be obtained by lower level techniques such at transistor sizing or buffering.

The second phase of timing optimization is performed during the physical design process. Here, the target technology is known and more accurate timing information is available. Optimization involves transistor sizing, buffering, delay-driven placement, etc. This phase is characterized by its dependence on a particular target technology and on the existence of fast and relatively accurate timing simulators.

The third and last phase of timing optimization is performed when actual designs are available. There, much more accurate timing analyzers are used to fine tune the circuit parameters. This phase serves both the optimization and verification purpose.

This chapter is dedicated to the first phase of timing optimization. There have been several previous attempts to solve this problem. SOCRATES [26] uses a rule based approach and tries to achieve global restructuring through a sequence of local transformations. Even though the system is very flexible in adapting to various cell libraries and target technology, it is heavily dependent on the rule set and the order in which the rules are applied. More recently, an algorithmic-based restructuring technique was developed in the Yorktown Silicon Compiler [10]. Even though the work lacks detailed algorithms and theoretical analysis, it had several interesting ideas from which some ideas presented in this chapter were originated. Section 2 gives some basic definitions to be used later on in the chapter. Section 3 defines exactly the problem to be solved along with appropriate abstractions. Section 4 discusses step by step the timing optimization algorithm. Section 5 focuses on an important step of the algorithm, the timing-driven decomposition, and provides some

theoretical support for the algorithm. Section 6 studies incremental delay trace, a technique that can not only significantly increases the speed of the timing optimization algorithm but also has applications in other parts of performance optimization systems. The last two sections present some experimental results and discussions on future work in this area.

6.2 Basic Definition

The *arrival time* of a signal is the time at which the signal settles to its steady state value. A_s is used to denote the arrival time of signal s. (All times are relative to an arbitrary, but common, reference point) The *required time* of a signal is the time at which the signal is required to be stable. R_s is used to denote the required time of signal s.

The *slack* of a signal is the difference between its required time and arrival time. S_s is used to denote the slack of signal s and is defined as

$$S_s = R_s - A_s.$$

It is clear that the slack value of a signal measures its criticality, i.e., signals with negative slacks are considered to be critical. Unlike arrival times and required times, slacks have no reference point. For these reasons, slacks are some time more convenient to use.

In order to trace circuit delays, certain delay models of logic gates have to be used. The *linear delay model* breaks the delay through a gate into two components, the block (intrinsic) delay and the fanout delay. Let x be an input of a gate. The delay from x to the output of the gate is modeled by

$$d = B_x + D * NFO$$

where B_x is the block delay from input x to the output, D is the fanout driving ability of the gate, and NFO is the number of gates which this gate drives.

Two specific forms of the linear delay model are often used. The *unit-fanout* delay model is a linear model with B_x and D being non-zero constants for all gates. This delay model is intended to capture the timing properties of circuits in a technology independent fashion.

A unit-fanout model is called the *unit* model if $D = 0$.

These models subsume some of the commonly used models. For example, a unit model with $B_x = 1$ measures circuit delays as number of levels of logic gates.

6.3 Technology Independent Timing Optimization

Technology independent timing optimization is also referred to as global circuit re-structuring. It aims at delay reductions which are beyond the capabilities of low level techniques such as transistor sizing or signal buffering. More specifically, the purpose of re-structuring is to discover alternative decompositions of circuits such that critical paths are minimized, and at the same time keep the area increase as small as possible. Since re-structuring is independent of any target technology and implementation style, it is important to judge correctly the timing quality of the circuits to be optimized. Without an actual implementation, any existing method for measuring timing information of the circuits at this stage is necessaryly inaccurate. For this reason, technology-independent timing quality of the circuits should be justified by one of the following methods:

1. Develop a technology-independent delay model. Since a node in a Boolean network can be arbitrarily complex, the delay model should take into account the logic function at the node and its factored form. This is best used when the target technology is random static CMOS, in which the transistor net-list of a gate is given directly by the factored form of the logic function. This delay model can be used to judge the quality of the circuit and to identify its critical sections before the actual implementation is available.

2. Represent the circuits in some canonical form, e.g. using unlimited fanin NAND functions. Because of the specific representation, it is possible to correlate some simple circuit parameters (e.g. number of logic levels or number of fanins of a function) with delays of final implementations. Such approachs could reduce the circuit delay to a very simple form and greatly simplify the re-structuring algorithm. The simplified delay model does not have to give accurate timing numbers and may even use completely different unit for measuring delays. The numbers given by the simplified delay model are not to be used as quantitative measurement of delays, but rather as qualitative measurement, i.e. the larger the simplified delay number is, the longer the actual delay will be in the final implementation.

The timing optimization presented in this chapter uses the second approach. It is specifically designed to be used in conjunction with technology mapping in MIS. The general approach taken in MIS is to minimize first the area of a network without concern for the

delay. Then at the beginning of the timing optimization phase, the area of the network is minimized (e.g. all the global common factors have been extracted out). Next, the network is decomposed into 2-input NAND gates and inverters, the input format for the technology mapping algorithms. At this point, timing optimization is invoked to re-structure the circuit into an alternative 2-input NAND gate and inverter form in which critical paths are reduced at the possible expense of area. The output of timing optimization is then fed directly to the technology mapping stage.

In this environment, the input circuits are assumed to have already being decomposed into 2-input NAND gates and inverters, and the output circuits are also required to be in the same format. The unit-fanout delay model serves as the simplified delay model for computing the circuit delays. It is chosen because it takes into account two of the major delay components in the circuit, the number of levels of logic from an input to an output and the number of fanouts of a signal. Timing constraints are specified as input arrival times of primary inputs and output required times of primary outputs. The goal of timing optimization is to meet the timing constraints while keeping the area increase to its minimum.

6.4 Re-structuring Algorithm

Given a Boolean network in 2-input NAND gate and inverter representation, the algorithm uses first the simplified delay model to identify the critical section of the network. In general, the unit-fanout delay model should be used because it takes into account both the number of logic levels as well as the number of fanouts of each signals. However, one can optimize only for the number of logic levels by simply replacing the unit-fanout delay model with the unit delay model. No other part of the algorithm needs to be changed.

The critical section of a Boolean network is made of critical paths from primary inputs to primary outputs. Given a critical path, the total delay on the path can be reduced if any section of the path is speeded up. For example, in Figure 6.2, part (a) shows a critical path, $a - x - y$. The critical path can be reduce by first collapsing x and y and then re-decomposing in a different way to minimize the critical path, as shown in part (b). This method (first collapsing along a critical path and then re-decomposing to shorten the critical path) is the basic step taken in the re-structuring. The nodes to be collapsed and re-synthesized form the *re-synthesis region*.

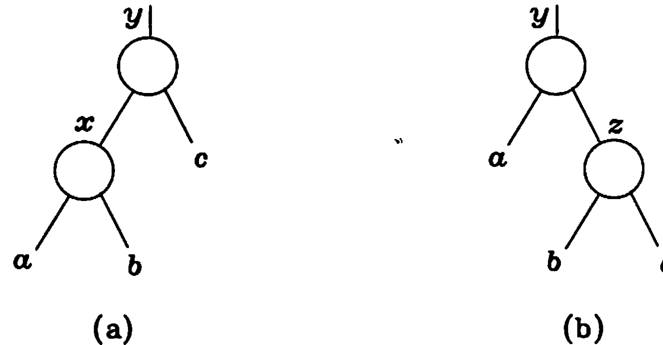


Figure 6.2: Reducing delay by collapsing and re-decomposition

Since a critical section usually consists of several overlapping critical paths, the algorithm looks at all of them at once and selects a minimum set of sub-sections, *re-synthesis points*, which when speeded up will reduce the delays on all the critical paths. In order to take into account of possible area increase during the collapsing and re-decomposing step and the total number of re-synthesis points needed to reduce all the critical paths, a weight is assigned to each candidate re-synthesis point. The goal is to select a set of points which cuts all the critical paths and has minimum total weight.

Once the re-synthesis points are chosen, they are speeded up by the collapsing-decomposing procedure. The simplified delay model is then used to find the new critical section of the network. The algorithm proceeds iteratively until the timing requirement is satisfied or no more improvement can be made. The following is an outline of the algorithm.

$SPEED_UP(\eta)$

1. Computing the arrival and required times for all the nodes in η.
2. Find all the critical nodes in η.
3. Compute a weight for each critical node.
4. Find the minimum weighted cut-set of all the critical paths.
5. Partially collapse along the critical path at each node on the cut-set.
6. Re-decompose each collapsed node into 2-input NAND gates and inverters.
7. If the timing requirement is satisfied, done.
8. If the circuit improved from the previous iteration, goto step 1.

Each of the steps in the algorithm will be described in detail in the following sub-sections. At the end of this section, the precise algorithm will be presented.

6.4.1 Identifying critical nodes

Given a linear delay model and timing constraints (input arrival times and output required times), a simple static timing analyzer is used to trace the delays through the network and compute for each signal its arrival time, required time, and slack. The following formula is used to compute the arrival time of signal s given that the arrival time of all the fanins are available.

$$A_s = max_{x \in FI(s)} \{A_x + B_x + D * NFO(s)\}$$

A_x is the arrival time of input x. B_x is the block delay from input x to the output s. D is the drive and $NFO(s)$ is the number of fanouts of s. Using this formula, the arrival time of all the signals in a network can be computed by the following recursive routine.

```
ARRIVAL(s)
  if A_s is not available at s {
    if s is not a primary input {
      For each x ∈ FI(s) {
        ARRIVAL(x)
      }
      A_s = max_{x∈FI(s)}{A_x + B_x + D * NFO(s)}
      store A_s at the node s
    }
  }
  return A_s
```

The routine computes recursively the arrival time of all the fanins and then uses that information to compute the arrival of s and store the arrival time at the node s so that it will not be re-computed. To compute the arrival times of all the signals in a network, simply calls the routine on each primary output of the network.

The required times are computed similarly. The formula for computing required time of a signal s is given below when the required times of all the fanout signals are known

is

$$R_s = min_{x \in FO(s)} \{R_x - B_s^x - D^x * NFO(x)\}.$$

The recursive routine is

```
REQUIRED(s)
   if R_s is not available {
      if s is not a primary output {
         For each x ∈ FO(s) {
            REQUIRED(x)
         }
         A_s = min_{x∈FO(s)}{R_x − B_s^x − D^x * NFO(x)}
         store R_s at the node s
      }
   }
   return R_s
```

The slacks are simply computed by

$$S_s = R_s - A_s.$$

Any node with negative slack is *critical* because the stable value of the signal is available later than it is required. A *critical path* in a network is a path from a primary input to a primary output such that all the nodes on the path are critical. More precisely, the set of critical paths of a network η is

$$CP(\eta) = \{(x_1, x_2, \ldots, x_n) | x_i \in NODES(\eta),\ S_{x_i} < 0,\ x_i \in FI(x_{i+1}),\ x_1 \in PI(\eta),\ x_n \in PO(\eta)\}.$$

In order for the algorithm to concentrate on speeding up the most critical signals, the notion of ϵ-criticality is introduced. A signal s is said to be ϵ-critical if $S_s < MS + \epsilon$ where MS is the minimum slack in the network. An ϵ-critical path is a path from a primary input to a primary output which consists of only the $\epsilon-critical$ nodes. The ϵ-critical section of a network η contains of all the ϵ-critical signals and can be computed by

---

```
CR(η, ε)
   for each f ∈ PO(η)
      ARRIVAL(f)
```

```
for each f ∈ PI(η)
    REQUIRED(f)
MS = ∞
for each f ∈ NODES(η)
    S_f = R_f − A_f
    MS = min{MS, S_f}
C = φ
for each f ∈ NODES(η)
    if S_f < ε + MS
        C = C ∪ {f}
return C
```

The following lemma shows a property of all ϵ-critical sections, i.e. an ϵ-critical section of a network is simply a collection of all the ϵ-critical paths. This property is stated formally as the following lemma and will be used in the subsequent sections.

PROPOSITION 6.4.1 *Let x be an ϵ-critical node in a network η. Then, there exist an $f \in FO(x)$ and a $g \in FI(x)$ such that both f and g are ϵ-critical.*

Proof. First, we show that there is at least one fanin of x whose slack is less than or equal to the slack of x. Since $A_x = max_{y \in FI(x)}\{A_y + B_y^x + D^x NFO(x)\}$, let g be the fanin that gives the maximum value of A_x, i.e.

$$A_x = A_g + B_g^x + D^x * NFO(x).$$

Also because $R_g = min_{y \in FO(g)}\{R_y - B_g^y - D^y * NFO(y)\}$, we have

$$R_g \leq R_x - B_g^x - D^x * NFO(x).$$

Combining the two, we have

$$A_x + R_g \leq A_g + R_x$$

$$R_g - A_g \leq R_x - A_x$$

$$S_g \leq S_x$$

So, if x is *epsilon*-critical, so is g.

Next, we show that there is at least one of the fanout of x whose slack is less than or equal to the slack of x. Since $R_x = min_{y \in FO(x)}\{R_y - B^y_x - D^y * NFO(y)\}$, let f be the fanout that gives the minimum value of R_x, i.e.

$$R_x = R_f - B^f_x - D^f * NFO(f).$$

Also because $A_f = max_{y \in FI(f)}\{A_y + B^f_y + D^f * NFO(f)\}$, we have

$$A_f \geq A_x + B^f_x + D^f * NFO(f).$$

Combining the two, we have

$$R_x + A_f \geq R_f + A_x$$

$$R_x - A_x \geq R_f - A_f$$

$$S_x \geq S_f$$

So, if x is ϵ-critical, so is f. ■

6.4.2 Computing weights of ϵ-critical nodes

Some ϵ-critical nodes are easier to speed up than others. For example, in part (a) of Figure 6.3, all the nodes are ϵ-critical. If node y is selected, collapsing its critical fanin

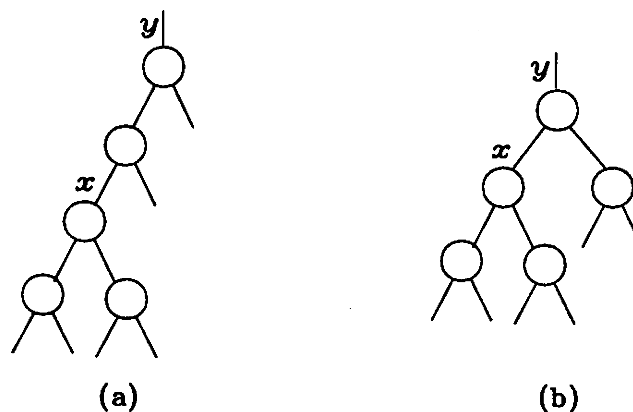


Figure 6.3: Node y is easier to be speed up than x is

into y will result in a node with one critical input, x, and two non-critical inputs. So, it is easy to decompose it such that the critical path is reduced, as indicated in part (b). If on

the other hand x is chosen, collapsing its critical fanins into x will result in a node with all of its fanins being critical. So, there is no decomposition that can reduce the critical paths in this case. The weight of an ϵ-critical node should reflect how easy it is to re-synthesis at the node.

It is also possible that, in order to reduce a critical path, certain nodes have to be duplicated. For example, part (a) of Figure 6.4 is part of a network with ϵ-critical signals

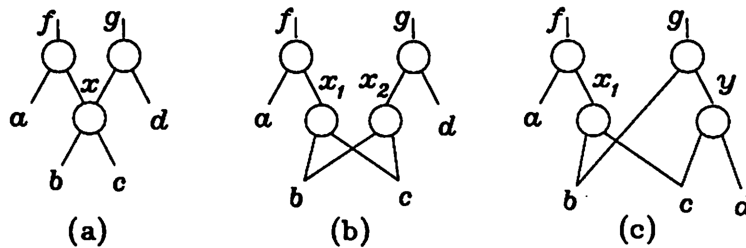


Figure 6.4: Area increase during re-synthesis

$b - x - g$. If g is chosen as a re-synthesis point, x needs to be collapsed into g and re-decomposed in a different way. Since f also depends on x, x needs to be duplicated before the collapsing, as indicated in part (b). Now, the critical path becomes $b - x_2 - g$ and can then be reduced as shown in part (c). This increase in area should be reflected in the weight of g. Also it may be that x existed initially because it had good area value. Now, however, its fanout has been reduced, so it should be examined again for its area value and eliminated it profitable.

Both the ease of re-synthesis and the area increase of an ϵ-critical node depend on the size of the *re-synthesis region* at the node. The re-synthesis region of a node consists of a set of ϵ-critical nodes which are at most distance d away from the node. d is a parameter for the global re-structuring algorithm, and is used to control the amount of speed up to be made in each iteration.

Now, the weight of an ϵ-critical node x if a function of d, the distance parameter, and can be defined as

$$W_x(d) = W_x^t(d) + \alpha * W_x^a(d)$$

where W_x^t is the delay component reflecting the ease of speed up, and W_x^a is the area component reflecting the area increase. α is used to control the tradeoff between delay reduction and area increase. Let $N(d)$ be the set of signals which are the inputs to the re-synthesis region of x and $M(d)$ be the nodes in the re-synthesis region. A node in $M(d)$

is *shared* if it is a transitive fanin of another node which is not a transitive fanin of x. Now, we can define

$$W_x^t(d) = \frac{|\{y \in N(d) | S_y \leq \epsilon\}|}{|N(d)|}$$

$$W_x^a d = \frac{|\{y \in M(d) | y \text{ is shared}\}|}{|M(d)|}.$$

It is clear that the easiness of speed up is inversely proportional to W^t and area increase is directly proportional to W^a.

The example in Figure 6.5 illustrates the weight computation. In the example, d

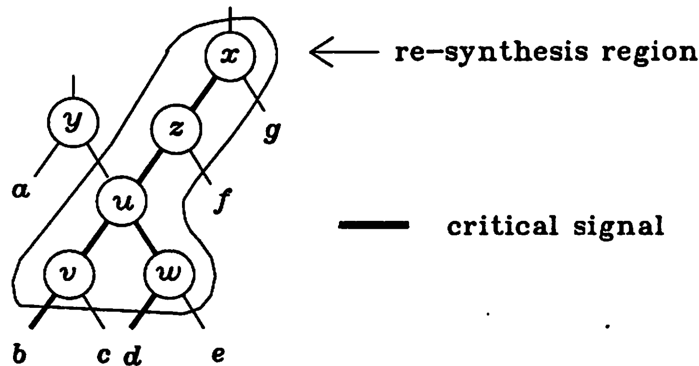


Figure 6.5: To re-synthesis at node x, nodes u, v, and w have to be duplicated.

is equal to 3. $\{x, z, u, v, w\}$ are the nodes in the re-synthesis region. $\{u, v, w\}$ are the shared nodes because they are also in the transitive fanin of y. b and d are the ϵ-critical input signals to the re-synthesis region. So, the weight of x is

$$W_x(3) = \frac{2}{6} + \alpha * \frac{3}{5}.$$

whereas for $d = 1$

$$W_x(1) = \frac{1}{3} + \alpha * \frac{0}{2}.$$

Hence, d affects the relative tradeoff potential of a node. This suggests d should be dynamically determined for each node, i.e., for each node various values of d are used to compute W_x and the one most favorable is used.

6.4.3 Finding minimum weighted cut-sets

After assigning weights to the ϵ-critical nodes, the next step is to select a set of nodes which, when speeded up, will reduce the delay through every ϵ-critical path in the

network. This means that at least one node on each critical path must be selected. To formalize the idea, we define

DEFINITION 6.4.2 *For network η, an ϵ-cut-set (or simply cut-set), $CS(\eta, \epsilon)$ is a set of nodes such that for every ϵ-critical path $p \in CP(\eta, \epsilon)$, $p \cap CS(\eta, \epsilon) \neq \phi$.*

The minimum weighted cut-set, $MWCS(\eta, \epsilon)$, is a cut-set such that the total weight of the nodes in the cut-set is minimum over all the cut-sets in η.

To find the minimum weighted cut-set of a Boolean network, we first construct a flow network [34], then use the Max-Flow Min-Cut algorithm to find the minimum capacity edge cut-set of the flow network, and finally map the edge cut-set back to a node cut-set of the Boolean network. We begin with the definition of flow network.

DEFINITION 6.4.3 *A flow network $N = (V, E, s, t, C)$ is a directed graph (V, E) with two special vertices, s, the source, and t, the sink, and a function $C : E \rightarrow Z^+$ which assigns capacities to the arcs.*

The Max-Flow Min-Cut Theorem [34] states that the maximum value of an (s, t)-flow is equal to the minimum capacity of an (s, t)-cut-set, and furthermore, the minimum capacity cut-set can be found in polynomial time.

Given a Boolean network and its ϵ-critical section, the following procedure is used to construct a flow network from the Boolean network.

FLOW_NETWORK(η, ϵ)

1. For each $x \in CR(\eta, \epsilon)$, create two vertices v_x^- and v_x^+ and an arc (v_x^-, v_x^+) with capacity W_x.
2. For each $x, y \in CR(\eta, \epsilon)$ such that $x \in FI(y)$, create an arc (v_x^+, v_y^-) with capacity ∞.
3. Create a source vertex s and a sink vertex t.
4. For each $x \in CR(\eta, \epsilon) \cap PO(\eta)$, create an arc (v_x^+, t) with capacity ∞.
5. For each $x \in CR(\eta, \epsilon) \cap PI(\eta)$, create an arc (s, v_x^-) with capacity ∞.

Notice that the flow network so constructed has exactly one source vertex s and one sink vertex t because every internal critical node has at least one critical fanin and one critical fanout, according to Proposition 6.4.1.

LEMMA 6.4.4 *Node cut-sets of the Boolean network are in one-to-one correspondence with finite capacity edge cut-sets of the flow network.*

Proof. The proof follows from the one-to-one correspondence between a critical path in the Boolean network and an $s - t$ path in the flow network, i.e.

$$(x_1, x_2, \ldots, x_n) \leftrightarrow (s, v_{x_1}^-, v_{x_1}^+, v_{x_2}^-, v_{x_2}^+, \ldots, v_{x_n}^-, v_{x_n}^+, t)$$

■

THEOREM 6.4.5 *The minimum weighted node cut-set of a Boolean network is given by the minimum capacity cut-set of a flow network constructed by $FLOW_NETWORK$.*

Proof. For any Boolean network η, there is at least one node cut-set. That cut-set is $CR(\eta, \epsilon)$, the set of all the ϵ-critical nodes. Therefore, there is at least one finite capacity edge cut-set of the corresponding flow network, because all the weights (capacities) associated with the node s, (v_s^-, v_s^+), are finite. Thus, the minimum capacity cut-set of the flow network contains only finite capacity arcs and its corresponding node cut-set has the minimum total weight among all the ϵ-cut-sets of η (by Lemma 6.4.4). ■

Using Theorem 6.4.5 and procedure $FLOW_NETWORK$, the algorithm that finds the minimum weighted ϵ-cut-set of a Boolean network η can now be defined.

$MWCS(\eta, \epsilon)$
  $FN = FLOW_NETWORK(\eta, \epsilon)$
  $C = MINIMUM_CAPACITY_CUTSET(FN)$
  $CS = \{x | (v_x^-, v_x^+) \in C\}$
  return CS

6.4.4 Re-synthesis

Once the minimum weighted node cut-set is found, each node on the cut-set is then re-synthesized. The re-synthesis of a node x involves collapsing all the ϵ-critical fanins of x, which are at most distance d from x, into x, and then decomposing x back to 2-input NAND gates and inverters such that the critical path is minimized. The objective of the decomposition is to minimize A_x, the arrival time of x. Timing-driven decomposition algorithms will be presented and analyzed in the next section. Here, we simply use $COLLAPSE(x, d)$ and $TIMING_DECOMP(x)$ to denote the collapsing and re-decomposition operations in the re-synthesis of node x.

6.4.5 Re-structuring algorithm

With all the basic steps previous described, the global re-structuring algorithm can now be stated precisely as follows.

```
SPEED_UP(η, d, ε)
  repeat {
    C = CR(η, ε)
    for each x ∈ C, compute W_x(d)
    CS = MWCS(η, ε)
    for each x ∈ CS {
      COLLAPSE(x, d)
      TIMING_DECOMP(x)
    }
  } until requirement is satisfied
  return η
```

6.5 Timing-Driven Decomposition into Trees

Timing-driven decomposition in this section is specifically referred to as decomposing a single-output logic function into a tree of 2-input NAND gates and inverters such that the output arrival time is minimized, given the arrival times of all the inputs and a delay model. Existing methods include rule-based approaches [20] [5] and algorithmic tree-balancing techniques [32]. All these techniques work on an existing decomposition and

modify incrementally the decomposition to reduce the output arrival time. In this section, a direct constructive algorithm is described. Given a function and arrival times of all the inputs, the algorithm decomposes optimally the function into 2-input NAND gates and inverters with minimum output arrival time. Exact conditions will be given under which the algorithm produces optimum results.

6.5.1 Theorems About Optimum Decomposition

The following notation and definitions are needed.

NOTATION:

ND_x - n-input NAND gate.

X - set of variables.

$ND_x(X)$ - n-input NAND gate with inputs $X = \{x_1, x_1, \ldots, x_n\}$.

D_f - tree decomposition of f using 2-input NAND gates and inverters.

$ROOT(D_f)$ - root node of D_f, which could be a NAND gate or an inverter.

A_{D_f} - arrival time of $ROOT(D_f)$.

D_f^o - optimum decomposition of f, i.e. $A_{D_f^o}$ is minimum.

DEFINITION 6.5.1 *Let x and y be two signals, x is said to be* earlier *than y if $A_x \leq A_y$. Let $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$ be two sets of signals. X is said to be* earlier *than Y if there is a permutation π such that*

$$A_{x_i} \leq A_{y_{\pi_i}} \qquad i = 1, 2, \ldots, n$$

A circuit has the *monotone speedup property* [1] if speeding-up certain inputs (making them available sooner) can only reduce the arrival time of the output. For many commonly encountered circuits and commonly used delay models, the monotone speedup property holds. This property will be used later on in the section to prove theorems and to design algorithms. Here, several specific forms of the monotone speedup property are stated formally as lemmas.

[1] Named by Patrick McGeer

LEMMA 6.5.2 (Monotone Speedup Property - first form) *Let T be a tree decomposition of an n-input logic function where each node in T is either a 2-input NAND gate or an inverter. Let the inputs of T be $X = \{x_1, x_2, \ldots, x_n\}$. Let T' be T with one of the input x_i being replaced by y such that $A_y \leq A_{x_i}$. Then,*

$$A_{T'(x_1,x_2,\ldots,x_{i-1},y,x_{i+1},\ldots,x_n)} \leq A_{T(x_1,x_2,x_{i-1},x_i,x_{i+1},\ldots,x_n)}$$

Proof. Since T is a tree decomposition, signals whose arrival times are affected must be on the path from x_i to the root r, i.e. $x_i - s_1 - s_2 - \ldots - s_m$ where s_m is the root. Let $y - s'_1 - s'_2 - \ldots - s'_m$ be the corresponding path in T'. Using our linear delay model, we have $A_{s'_1} \leq A_{s_1}$ because $A_y \leq A_{x_i}$, $A_{s'_2} \leq A_{s_2}$ because $A_{s'_1} \leq A_{s_1}$, and in general $A_{s'_j} \leq A_{s_j}$ because $A_{s'_{j-1}} \leq A_{s_{j-1}}$. So, $A_{s'_m} \leq A_{s_m}$. Thus, we have

$$A_{T'(x_1,x_2,\ldots,x_{i-1},y,x_{i+1},\ldots,x_n)} \leq A_{T(x_1,x_2,x_{i-1},x_i,x_{i+1},\ldots,x_n)}$$

■

Lemma 6.5.2 can be generalized to allow any subset of inputs to be replaced by their corresponding faster signals. Here, the Lemma is stated for NAND gates, but can be easily generalized for any n-input functions which have tree-decompositions.

LEMMA 6.5.3 (Monotone Speedup Property - second form) *Let f be an n-input function, $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \ldots, y_m\}$ be two sets of signals such that $A_{x_i} \leq A_{y_i}$ for $i = 1, 2, \ldots, n$. Then,*

$$A_{D^o_{f(X)}} \leq A_{D^o_{f(Y)}}$$

Proof. Let T be $D^o_{f(Y)}$, the optimum decomposition of $f(Y)$. Then,

$$\begin{aligned} A_{D^o_{f(Y)}} &= A_{T(y_1,y_2,\ldots,y_n)} \\ &\geq A_{T(x_1,y_2,\ldots,y_n)} \\ &\geq A_{T(x_1,x_2,\ldots,y_n)} \\ &\geq \cdots \\ &\geq A_{T(x_1,x_2,\ldots,x_n)} \\ &\geq A_{D^o_{f(X)}} \end{aligned}$$

The last inequality is true by the definition of $D^o_{f(X)}$. All the other inequalities are true by Lemma 6.5.2. So, we have

$$A_{D^o_{f(X)}} \leq A_{D^o_{f(Y)}}$$

■

There are many implications of Lemma 6.5.2 and Lemma 6.5.3. In particular, they reveal certain structures of optimum timing decomposition of a class of logic functions, and therefore provide the bases for designing optimum timing decomposition algorithms.

THEOREM 6.5.4 *Let f be an $(n+2)$-input NAND gate with u and v being the earliest arriving signals, i.e, $f = ND_{n+2}(X, u, v)$ and $A_u \leq A_v \leq A_{x_i}$ for $i = 1, 2, \ldots, n$. Let $w = uv$ and $g = ND_{n+1}(X, w)$. Then*

$$A_{D^o_f} = A_{D^o_g},$$

i.e., there is an optimum decomposition of f with u and v being the inputs to a NAND gate.

Proof. By definition of D^o_f, we have $A_{D^o_f} \leq A_{D^o_g}$. So, we only need to prove $A_{D^o_f} \geq A_{D^o_g}$. Let u and s feed into a NAND gate p, and v and t feed into a NAND gate q, where s and t could be inputs or internal signals.

case 1: $p \notin TFI(q)$ and $q \notin TFI(p)$. This situation is illustrated in part (a) Figure 6.6. It shows the overall structure of the decomposition where T_s is the sub-tree

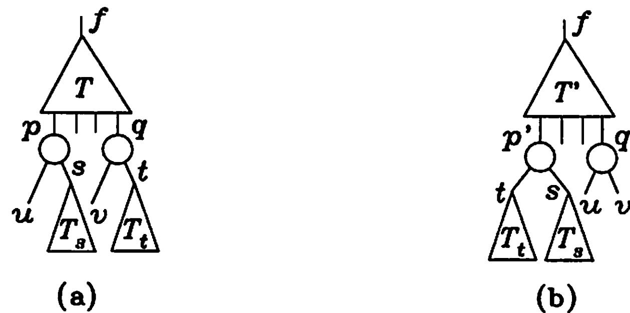


Figure 6.6: case 1: p and q are independent

rooted at s, T_t is the sub-tree rooted at t, and T is the remaining part of the decomposition. Now, if we switch u and t, as indicated in part (b) of Figure 6.6, we have the following:

$$A_{\bar{p}} = max(A_p, A_q)$$

$$A_{\tilde{q}} \leq min(A_p, A_q)$$

because $A_u \leq A_s$ and $A_v \leq A_t$. Therefore, the inputs of $\tilde{T}$ are earlier than the inputs of T, and we have

$$A_{D_g^o} \leq A_{\tilde{T}} \leq A_T = A_{D_f^o}.$$

The first "$\leq$" holds because $(\tilde{T}, \tilde{p}, T_s, T_t)$ is a decomposition of g. The second "$\leq$" holds by Lemma 6.5.3.

case 2: $p \in TFI(q)$ or $q \in TFI(p)$, as illustrated in part (a) of Figure 6.7. Without loss of generality, we assume q is in the transitive fanin of p. In this situation, we

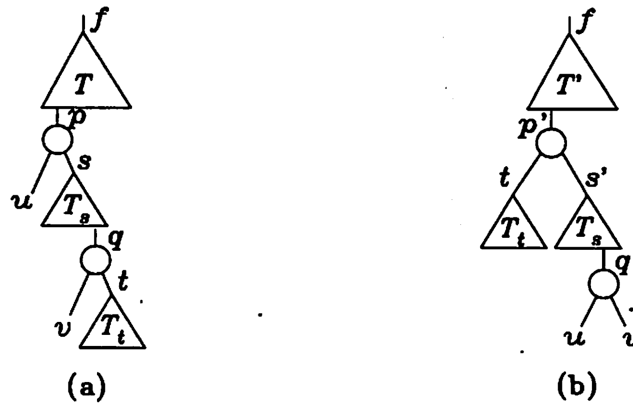


Figure 6.7: case 2: p depends on q

again switch u and t. The changes are reflected in part (b) of Figure 6.7. Since $A_u \leq A_t$, we have $A_{\tilde{s}} \leq A_s$ and $A_t \leq A_s$, which implies $A_{\tilde{p}} \leq A_p$. Therefore, we have

$$A_{D_g^o} \leq A_{\tilde{T}} \leq A_T = A_{D_f^o}.$$

Again, the first "$\leq$" holds because $(\tilde{T}, \tilde{p}, T_s, T_t)$ is a decomposition of g. The second "$\leq$" holds by Lemma 6.5.2 (or Lemma 6.5.3).

So, we have proved that $A_{D_f^o} = A_{D_g^o}$. ■

The theorem makes a simple statement that for a n-input NAND gate, there is an optimum timing decomposition in which there is a 2-input NAND gate with the two earliest arriving signals as the inputs. This theorem will be used later on to derive an algorithm which guarantees optimum decomposition of NAND gates.

Next, we look at a more complicated situation in which the functions to decomposed are not restricted to NAND gates. When a function is a sum of orthogonal cubes

(cubes with disjoint support), its decomposition has a very special structure. In fact, for any decomposition of such a function, one can draw a boundary between the decomposition of each cube and that of the OR function.

LEMMA 6.5.5 (AND-OR boundary lemma) *Let $\{c_1, c_2, \ldots, c_n\}$ be a set of cubes such that $c_i \perp c_j$ for all $i, j \leq n$. Let $f = c_1 + c_2 + \ldots + c_n$, i.e. f is a sum of orthogonal cubes. Let D_f be any tree decomposition of f using 2-input NAND gates and inverters. For each c_i, there is an internal signal (node) s_i in D_f such that the sub-tree rooted at s_i, T_{s_i}, is $D_{\bar{c}_i}$.*

Proof. First, we prove that there is a NAND gate x in D_f with inputs u and v such that the support of T_u is disjoint from the support of c_i and the support of T_v is exactly the support of c_i. Figure 6.8 shows the circuit structure. It is clear that the structure of

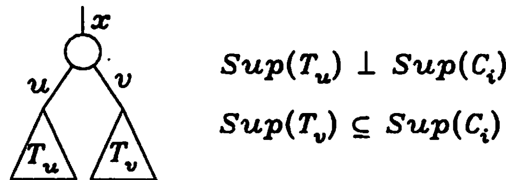


Figure 6.8: A structure always exists in D_f

Figure 6.8 always exists with $sup(T_v) \subseteq sup(c_i)$. We want to show that $sup(T_v) = sup(c_i)$. Suppose $sup(T_v) \subset sup(c_i)$. Setting all inputs of T_v to 0 and propagating the constants will result in a new circuit which is the decomposition of $f_{\bar{c}_i} = f - c_i$. The constant propagation must stop at x with input v having value 1, because T_u must remain in the circuit. If $sup(T_v) \subset sup(c_i)$, then some inputs of c_i must remain in the new circuit after the constant propagation. Notice that if a circuit is a tree of NAND gates and inverters, i.e., no multiple fanout even at the leaves, then no redundancy exists in the circuit. Since D_f is a tree, the new circuit containing variables in c_i can not possibly be $f - c_i$, and we have a controdiction. Therefore, we must have $sup(T_v) = sup(c_i)$.

Next, we show that v is s_i. This is quite easy. Setting all inputs of T_v (c_i) to 1 and propagating the contants will result in a constant circuit 1. The propagation can not stop at x with $v = 1$. So, v must be equal to 0 in this case. Setting any non-empty subset of inputs of T_v (c_i) and propagating the constant will result in a decomposition of $f - c_i$ not containing any variables in T_v (c_i). The propogation must stop at x with $v = 1$. Therefore,

we must have $v = \bar{c_i}$. So, $s_i = v$ exists. ■

A SOP form can also be viewed as a NAND-NAND form. Lemma 6.5.5 says that if f is the sum of orthogonal cubes, then *any* tree decomposition of f is made of tree decompositions of first level NAND gates feeding into the tree decomposition of the second level NAND gate, as indicated in Figure 6.9. This is quite useful because it gives us an

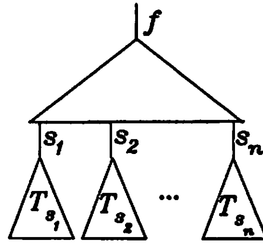


Figure 6.9: Structure of the tree decomposition of sum of orthogonal cubes

algorithm for the optimal decomposition of any function f which is a sum of orthogonal cubes. The following theorem formalizes the idea.

THEOREM 6.5.6 *Let f be the sum of orthogonal cubes, i.e. $f = c_1 + c_2 + \ldots + c_n$ and $c_i \perp c_j$ for all $i \neq j$. Let $s_i = \bar{c_i}$ for all i and $g = \overline{c_1 c_2 \ldots c_n}$. Then, $(D^o_g, D^o_{s_1}, D^o_{s_2}, \ldots, D^o_{s_n})$ is an optimum decomposition of f, i.e.*

$$A_{D^o_f} = A_{D^o_g}.$$

Proof. By definition, we have $A_{D^o_f} \leq A_{D^o_g}$. To prove $A_{D^o_f} \geq A_{D^o_g}$, take an optimum decomposition D^o_f of f. By Lemma 6.5.5, there exists in D^o_f a set of signals $s_1, s_2, \ldots, s_n$ such that each s_i is equal to $\bar{c_i}$. For each s_i, replace its fanin sub-tree T_{s_i} with $D^o_{\bar{c_i}}$ and call the new signal $\tilde{s_i}$. Clearly, s_i and $\tilde{s_i}$ are logically equivalent and

$$A_{\tilde{s_i}} \leq A_{s_i} \qquad i = 1, 0, \ldots, n$$

Now, $\tilde{S} = \{\tilde{s_1}, \tilde{s_2}, \ldots, \tilde{s_n}\}$ is earlier than $S = \{s_1, s_2, \ldots, s_n\}$, by Lemma 6.5.3, we have $A_{D^o_g} \leq A_{D^o_f}$. Thus, we have proved $A_{D^o_f} = A_{D^o_g}$. ■

6.5.2 Timing Decomposition Algorithms

The theorems developed in the previous section suggest certain algorithms for optimum timing decomposition of logic functions, and guarantee the optimality of the results when the functions have some special properties.

A general paradigm used in all the algorithms developed in this section is based on a bottom-up approach. The decomposition is a process of breaking a large function into a set of smaller sub-functions and recursively decomposing the sub-functions until all functions are either 2-input NAND gates or inverters. The bottom-up approach is to decompose a sub-function only if the arrival times of all the inputs are available. Once the decomposition of a sub-function is done, its output arrival time is computed so that it can be used to decompose the function that this sub-function feeds into.

The optimum timing decomposition algorithm of NAND functions is the direct result of Theorem 6.5.4. For a NAND function, the theorem states that there is an optimum timing decomposition in which the two earliest arriving signals are inputs of the same NAND gate. This suggests the following recursive algorithm for decomposing optimally a NAND function.

NAND_DECOMP$(f, model)$
  if $|FI(f)| \leq 2$
    return
  Let u and v be the two earliest arriving signals.
  $w = uv = u^{=}v$
  $g = SUBSTITUTE(f, w)$
  $A_w = ATIME(w, model)$
  NAND_DECOMP(g, model)
  return

The algorithm first selects the two earliest arriving inputs, u and v, then form a 2-input NAND gate with inputs u and v and feeding into an inverter. The output of the inverter is w. uv in f is then replaced by w by routine $SUBSTITUTE(f, w)$. The next step is to compute the arrival time of w using the delay model which is passed in as a parameter. The last step is to decompose recursively the new function g which is also a NAND function and has one less input than f. The recursion terminates when function f

is either a 2-input NAND gate or an inverter.

THEOREM 6.5.7 *Algorithm NAND_DECOMP yields the optimum timing decomposition of NAND functions.*

Proof. By induction on the number of inputs, n, of the n-input NAND gate ND_n. The theorem is obviously true for ND_2. Suppose it is true for ND_{n-1}. Let f be ND_n. By Theorem 6.5.4, there is an optimum timing decomposition of f with an internal signal $w = uv$ where u and v are the two earliest arriving inputs. The remaining decomposition of f with input w and the rest of the inputs must also be optimum. This remaining part is a decomposition of ND_{n-1}. By induction hypothesis, *NAND_DECOMP* will give optimum result for this ND_{n-1}. ∎

Another feature of the algorithm *NAND_DECOMP* is its independence of the delay model and the actual values of the input arrival times. The model is used to compute the arrival time of w that affects the choice of inputs at the next level of recursion. So, the algorithm automatically adapts to different delay models. Since there is no assumption about the actual values of the input arrival times, the result can be viewed as weight-balanced. The weight of an input is the combination of its arrival time and its distance to the root of the decomposition tree, i.e. the critical signals (inputs with large arrival times) are closer to the root.

NAND functions are very special. In global circuit re-structuring, functions encountered are usually more complicated. However, algorithms for decomposing more complicated functions often use *NAND_DECOMP* as a subroutine. In fact, if a function is kernel-free, its optimal timing decomposition is no more difficult to obtain than that of NAND functions. Theorem 6.5.6 suggests how to decompose a kernel-free function, is just a sum of orthogonal cubes. In the following algorithm, f is a set of cubes $(c_1, c_2, \ldots, c_n)$.

AND_OR_DECOMP(f, *model*)
  for each $c_i \in f$ {
    $s_i = \bar{c}_i$
    NAND_DECOMP(s_i, *model*)
    $A_{s_i} = ATIME(s_i, model)$
  }
  $f = \overline{s_1, s_2, \ldots, s_n}$
  NAND_DECOMP(f, *model*)

$A_f = ATIME(f, model)$
return

The algorithm treats the function in NAND-NAND form. The first level of NAND functions are decomposed first because all of their input arrival times are known. Once they are decomposed, their output arrival times are computed. Those arrival times are then used to decompose the second level NAND gate. The NAND gates are decomposed using routine *NAND_DECOMP*. The optimality of the result is guaranteed by Theorem 6.5.6.

AND_OR_DECOMP has wider applications than *NAND_DECOMP* does. Simple functions such as AND, OR, and NOR can all be treated as special case of kernel-free functions. The dual of kernel-free functions are product of orthogonal sums. They too can be optimally decomposed using *AND_OR_DECOMP*.

6.6 Incremental Delay Trace

Global timing optimization depends heavily on the delay information, such as slacks or arrival times, given by a chosen delay model. The procedures presented in Section 6.4.1 perform complete delay trace over an entire network. However, the re-structuring algorithm at each iteration modifies only certain sections of the network. It is not necessary, in fact it is quite wasteful, to re-compute completely all the timing information. A simple example is in Figure 6.10. The numbers at the nodes are the current arrival times. The

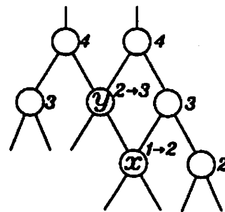


Figure 6.10: When A_x changes from 1 to 2, only A_y is effected

underlying delay model is *unit-delay*. Suppose that, during the re-synthesis, the arrival time of x changes from 1 to 2, the only node whose arrival time is affected is y. This is the direct consequence of the underlying delay model and the way arrival times are computed. Recall

that

$$A_x = max_{y \in FI(x)}\{A_y + B_y^x + D^x * NFO(x)\}.$$

As long as the maximum value of the right hand side stays the same, the arrival time of x remains un-affected. The primary objective of incremental delay trace is re-computing, after a Boolean network is modified, only the timing information that is actually changed. The secondary objective is to perform the computations in an efficient manner.

Let's use arrival time computation as an example to visualize the process of incrementally updating the arrival times after the arrival time of some signal changes. Typically, a change in an arrival time is called an *event*. If an event occurs at a node, it triggers a set of events at the fanout nodes, which in turn trigger more events. These events eventually die either because the change of arrival time at an input of a node did not affect the arrival time of the node, or because the events have propagated to the primary outputs. Obviously, there is a restriction on the order by which the events are handled, i.e. do not update the arrival time of a node unless all its input arrival times are updated. Thus, correct and efficient incremental delay trace requires a certain combination of data structure techniques and theoretical understanding of how the nodes should be visited.

In the following section, Boolean networks are abstracted as directed acyclic graphs (DAGs) and several graph theoretical results are developed concerning the properties of DAGs and the ordering of the nodes in DAGs. The goal is to gain some theoretical understanding of the subject, to make the task of algorithm design easier, and to provide foundations for proving correctness of the algorithms. Once the theoretical results are available, the incremental delay trace algorithms can be designed in a fairly straight-forward fashion.

6.6.1 Graph Theoretical Results

The basic object to be studied in this section is a DAG. We are mainly interested in finding and incrementally maintaining certain ordering of nodes in the DAG. Useful orderings in delay tracing should have one of the following properties.

- visiting a node after visiting all its **fanins**.
- visiting a node after visiting all its **fanouts**.

The formal definitions of these orderings are as follows:

DEFINITION 6.6.1 *An* **ordering** *of a DAG, $G = (V, E)$, is a function f that assigns each node in G a real number, i.e, $f : V \rightarrow \mathbf{R}$. f is also called an* **ordering function.** *A* **forward visit** *of nodes of G in ordering f is to visit the nodes, v's, in the increasing order of their values, $f(v)$'s. A* **backward visit** *of nodes of G in ordering f is to visit the nodes, v's, in the decreasing order of their values, $f(v)$'s.*

By definition, any function $f : V \rightarrow \mathbf{R}$ is an ordering function. The following definition defines a subset of ordering functions which are useful to us in delay tracing.

DEFINITION 6.6.2 *A* **topological ordering** *of a DAG, $G = (V, E)$, is an ordering function O with the property:*

$$O(v) < O(w) \quad \forall (v, w) \in E$$

Notice that such an ordering always exists, because G is acyclic.

To compute arrival times in a Boolean network, a node must be visited after all its fanins. To compute required times, a node must be visited after all its fanouts. A topological ordering gives us both of these visiting orders. In fact, we have

PROPOSITION 6.6.3 *Let O be a topological ordering of a DAG, $G = (V, E)$. The forward visit of nodes of G in ordering O visits a node after all its fanins. The backward visit of nodes of G in ordering O visits a node after all its fanouts.*

Proof. All we need to show is that the value of a node $v \in V$ given by $O(v)$ is strictly larger than all the values of its fanins and is strictly smaller than all the values of its fanouts, i.e.

$$O(u) < O(v) \quad \forall u \in V, (u, v) \in E$$

$$\textit{and}$$

$$O(v) < O(w) \quad \forall w \in V, (v, w) \in E.$$

This is guaranteed by the definition of O. ∎

To see an example of topological ordering, we need the following definition.

DEFINITION 6.6.4 *Given a DAG, $G = (V, E)$, a* **distance** *function D is defined recursively as*

- *$D(v) = 0$ if v is a source node.*
- *$D(v) = max_{w \in FI(v)}\{D(w)\} + 1$ otherwise.*

It is clear that the distance function D is a topological ordering. It gives us an ordering by which to compute both the arrival times and required times of signals in a Boolean network.

As stated earlier, our main interest is to update incrementally a topological ordering function as the graph changes. There are many ways a graph can change. However, all the changes can be made as a sequence of some primitive changes. In particular, the primitives are: deleting an edge and adding an edge. More complex operations can be performed by particular sequences of the primitive operations. For example, deleting a node requires deleting all the fanout edges and fan-in edges of the node. Another example is adding a node between a pair of connecting nodes. As illustrated in Figure 6.11, the operation can

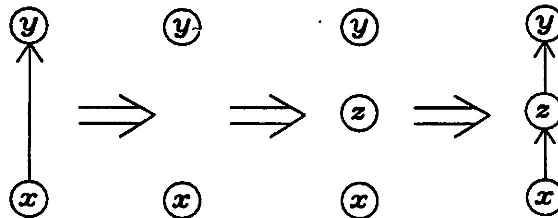


Figure 6.11: Inserting a node between a pair of connecting nodes

be accomplished by first deleting the connecting edge and then adding two new edges. Since the topological ordering is defined for DAGs only, it is implicitly assumed that the primitive operations on a graph do not create cycles. The next set of lemmas gives us a foundation for updating incrementally topological ordering functions.

LEMMA 6.6.5 *Let $G = (V, E)$ be a DAG, O be a topological ordering function of G, and e be an edge of G. Then, O is also a topological ordering function of $G' = (V, E - \{e\})$. In other word, deleting an edge from a DAG does not change any topological ordering of the nodes.*

Proof. $O(v) < O(w) \forall (v, w) \in E$ implies $O(v) < O(w) \forall (v, w) \in E - \{e\}$. So, O is still a topological ordering function for $G' = (V, E - \{e\})$. ∎

LEMMA 6.6.6 *Let $G = (V, E)$ be a DAG, O be a topological ordering function of G, and $e = (x, y)$ be an edge to be added into G. If $O(x) < O(y)$, then O is also a topological ordering function of $G' = (V, E \cup \{e\})$. In other word, adding an edge pointing from a node with smaller value to a node with larger value does not change any topological ordering of the nodes.*

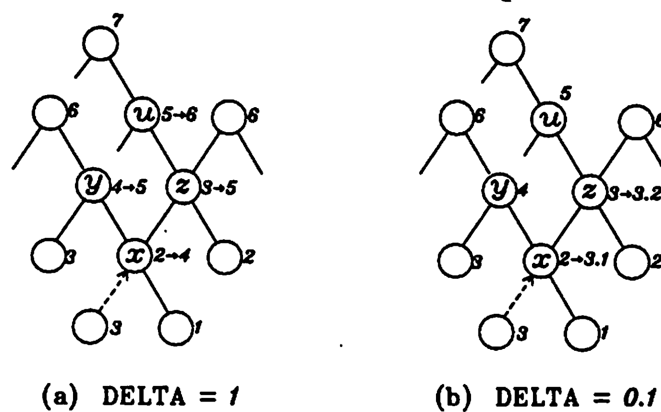
Proof. Because $O(v) < O(w) \forall (v, w) \in E$ and $O(x) < O(y)$, we immediately have $O(v) < O(w) \forall (v, w) \in E \cup \{e\}$. So, O is still a topological ordering function for $G' = (V, E \cup \{e\})$. ∎

The remaining case of primitive operations is adding an edge $e = (x, y)$ to a DAG such that $O(x) \geq O(y)$. It is obvious that this operation changes the topological ordering of the nodes in the graph. In this case, we would like to modify (incrementally update) O so it becomes a topological ordering function of the new DAG. The modifications involve assigning new values to some nodes in the DAG. For the moment, let's assume that the values are allowed only to increase. We first present a naive algorithm and then see how we can improve the efficiency of the algorithm by using certain data structure techniques.

```
REORDER_1(G, O, (x, y))
   let S = {y}
   while S ≠ φ {
      let x ∈ S such that FANIN(x) ∩ S = φ
      o = max_{y∈FANIN(x)}{O(y)} + DELTA
      if O(x) < o then {
         O(x) = o
         S = S ∪ FANOUT(x)
      }      S = S − {x}
   }
   return O
```

The algorithm *REORDER_1* takes a DAG, G, its current topological ordering O, and an edge (x, y) which is to be added into G. S is initialized to contain only node y and is used to keep all the nodes whose ordering value may possibly change. At each iteration, the algorithm picks a node from S such that all the fanins of the node are not in S, i.e. they have valid ordering values. Such a node always exists in S because the graph G is

acyclic. The algorithm then proceeds to compute a new ordering value of the node. If the new ordering value is greater than the old one, the ordering value is updated and all the fanout of the node are put into S because their ordering value may subsequently change. The algorithm stops when S becomes empty. In the computation of new ordering values, $DELTA$ is used to ensure the topological ordering condition, i.e. $O(x) < O(y)$ for every edge (x, y). Clearly, keeping $DELTA$ small can reduce the number of nodes to be updated. However, there is a minimum set of nodes which have to be updated regardless of the value of $DELTA$. For example, Figure 6.12 shows part of a DAG where the dashed line is the new edge. The numbers indicate current ordering values. Using $DELTA = 1$, nodes x, y,



(a) DELTA = 1 (b) DELTA = 0.1

Figure 6.12: Effect of $DELTA$ on the number of nodes updated

z, u have to be updated, as indicated in part (a). But, using $DELTA = 0.1$, only x and z have to be updated, as indicated in part (b). It is obvious that x and y need to be updated no matter how small the $DELTA$ is.

A problem with $REORDER_1$ is that the selection process at the first line of the while loop could be very slow. In the worst case, it has to examine all the nodes in S to find the one that satisfies the condition. So, instead of keeping the nodes in an un-ordered set S, it is desirable to keep some partial orderings among the nodes. Fortunately, the old ordering values of the nodes give us the partial ordering we need. S is modified to be an ordered set. Each node in S is given a value, its old ordering value. The minimum valued node in S has the property that none of its inputs is in S. So, selecting a node from S in $REORDER_1$ becomes quite easy. Then, the operations on S are: inserting a node into S with a value, and fetching a node in S with the minimum value. Thus, S is a *priority queue* where a smaller value means higher priority. The following notation is used to express the

priority queue operations.

DEFINITION 6.6.7 *Let Q be a priority queue, n be a node, and p be a number indicating a priority. $SCHEDULE(Q, n, p)$ denotes the operation of inserting n into Q with priority p. $FETCH(Q)$ returns a node in Q with highest priority.*

With the help of priority queue, $REORDER_1$ can now be modified to be more efficient. The new algorithm is called $REORDER$ which has the basic structure of algorithm $REORDER_1$. The set S is replaced by a priority queue Q. Each time a node is scheduled, the negative value of its old ordering is used so that smaller value implies higher priority.

```
REORDER(G,O,(x,y))
   let Q be a priority queue
   SCHEDULE(Q,y,-O(y))
   while Q ≠ φ {
      x = FETCH(Q)
      o = max_{y∈FANIN(x)}{O(y)} + DELTA
      if O(x) < o then {
         O(x) = o
         for each z ∈ FANOUT(x) {
            SCHEDULE(Q,z,-O(z))
         }
      }
   }
   return O
```

$REORDER$ schedules the nodes by their old ordering values and always updates the node with the smallest old ordering value. It is not at all clear that $REORDER$ is correct. In fact, it takes the following Theorem to establish the correctness of $REORDER$.

THEOREM 6.6.8 *The algorithm $REORDER$ correctly updates the ordering function.*

Proof. In order to show that $REORDER$ is correct, we need to first establish the fact that every node whose ordering function changes is put into the queue at some point. Then, we need to prove that the nodes in the queue are updated at the appropriate time.

Let z be any node whose ordering value needs to be updated. It is obvious that there must be a directed path from y to z. We show that z will eventually be put into the queue by induction on the distance from y to z. The statement is trivially correct when the distance is 0. Suppose the distance is n. By the definition of the ordering function, the change in ordering value of z must be caused by the change in ordering value of one of its input, u. u must be distance $n-1$ or less from y. By the inductive hypothesis, u must be put into the queue at some point and be updated later on. Since the ordering value of u changes, the algorithm will put, at the point of its evaluation, z into the queue.

To show that every node in the queue is updated at the correct time, let z be the highest priority node in the queue, i.e. the value of z is the smallest. We claim that all the inputs of z have correct ordering values at this point. Suppose one of its input u does not have correct ordering value. Since u is the fan-in of z, its old ordering value must be smaller than that of z. The fact that z is the highest priority node in the queue implies that u is not in the queue at this point. Because u is not in the queue and u has the incorrect ordering value, there must be a path from y to u such that every node on the path is not in the queue and has incorrect ordering value at this point. But this contradicts the fact that y is the first node put into the queue and, once it is evaluated, all of it fanouts are put into the queue. So the original assumption that u has incorrect ordering value is wrong. Therefore, all of the fanins of z have the correct ordering value and z is updated at the right time. ■

Until now, we have restricted ourselves to increasing ordering values of nodes in updating the ordering function. But nowhere in the definition of ordering function is this restriction forced. In fact, when adding an edge (x, y), it is just as valid to decrease the ordering value of x and its fanins as it is to increase the ordering value of y and its fanouts. For example in Figure 6.13, when adding the edge (x, y) in part (a), one can either increase values of y and its fanout shown in part (b), or decrease values of x and its fan-in shown in part (c). Depending on the current ordering function, one should choose one method or another to minimize the number of nodes to be updated. In incremental delay trace, there is no problem of determining how the ordering function is updated because of the particular choice of the ordering function.

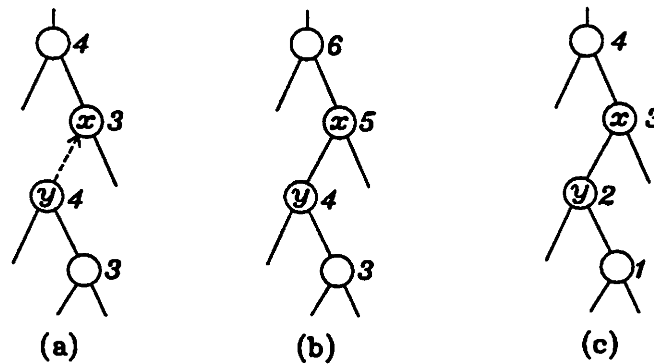


Figure 6.13: Two ways of updating an ordering function

6.6.2 Application in Incremental Delay Trace

The last section studied the properties of topological ordering of nodes in a DAG and how to update incrementally topological ordering functions. These results will be used as the basis for constructing an incremental delay trace system. As we'll see later on, the basic problem of incremental delay trace is to update incrementally some special topological ordering functions. So, the first task is to decide on what topological function to use. Once the topological function is chosen, several basic routines will be designed to serve as primitives in any incremental delay trace environment. Finally, several applications of incremental delay trace will be given to show the use of the basic primitive routines.

By definition, there are infinitely many topological ordering functions. In order to choose one that is most convenient to use in incremental delay trace, let's first look at some properties of arrival times and required times in a Boolean network. The arrival time of a node x is defined by

$$A_x = max_{y \in FI(x)}\{A_y + B_y + D * NFO(x)\}.$$

For any realistic delay model, B_y and $D * NFO(x)$ are positive. Thus, $A_y < A_x$ for all edges (y, x) in the Boolean network. So, the arrival times define a topological ordering function. Similarly, the required time of a node x is defined by

$$R_x = min_{y \in FO(x)}\{R_y - B_x^y - D^y * NFO(y)\}.$$

We have $R_x > R_y$, and $-R_x < -R_y$, for all edges (x, y) in the Boolean network. So, the negative of required times defines also a topological ordering function. Incremental

delay trace is nothing more than incrementally updating two special topological ordering functions.

Incremental delay trace can be used in a variety of ways depending on its particular applications. For example, in transistor sizing, incremental delay trace may be invoked after every single transistor change. In a rule-based system, it may be invoked after every application of rules. In our timing optimization system, it may be invoked after every *collapsing-decomposition* step. So, it is important to identify a few primitive routines which are, without lost of efficiency, general enough to be used in all applications.

In the previous section, we presented a routine *REORDER* which updates incrementally a given topological ordering function after inserting an edge into a DAG. Looking at it carefully, we find that *REORDER* really consists of two parts. The first part simply puts the node, whose ordering value has changed, onto a priority queue. The second part repeatedly fetches a node from the priority queue, computes its new ordering value, and puts more nodes, if any, onto the priority queue. These concepts are captured in the following two routines.

```
SCHEDULE_ARRIVAL(Q, x)
   SCHEDULE(Q, x, A_x)
```

```
UPDATE_ARRIVAL(Q)
   while Q ≠ φ {
      x = FETCH(Q)
      a = max_{z∈FI(x)}{A_y + B_y + D * NFO(x)}
      if A_x ≠ a {
         A_x = a
         for each y ∈ FO(x) {
            SCHEDULE_ARRIVAL(Q, y)
         }
      }
   }
```

SCHEDULE_ARRIVAL simply puts x onto the priority queue Q with the old arrival time as its priority. *UPDATE_ARRIVAL* fetches a node, evaluates its new arrival

time, and schedules more nodes until the queue becomes empty. The advantage of breaking the routine *REORDER* into these two routines is that now it is no longer necessary to update the ordering function after every single insertion of an edge into a DAG. In fact, one can schedule several nodes before calling *UPDATE_ARRIVAL* in order to be more efficient. For example in Figure 6.14, if the arrival times of x and y are both changed,

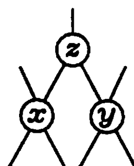


Figure 6.14: If x and y are scheduled before updating the arrival time, z is evaluated only once. Otherwise, z might be evaluated twice.

it would be more efficient to schedule both x and y before calling *UPDATE_ARRIVAL*, in which case the arrival time of z is computed only once. If *UPDATE_ARRIVAL* were called after scheduling x and called again after scheduling y, the arrival time z would have been computed twice. Just how many nodes to schedule before updating the arrival times depends on the particular applications.

Required times can be updated in a similar way by the following two routines.

SCHEDULE_REQUIRED(Q, x)
  SCHEDULE(Q, x, R_x)

UPDATE_REQUIRED(Q)
  while $Q \neq \phi$ {
    $x = FETCH(Q)$
    $r = min_{z \in FO(x)}\{R_z - B_x^z - D^z * NFO(z)\}$
    if $R_x \neq r$ {
      $R_x = r$
      for each $y \in FI(x)$ {
        SCHEDULE_REQUIRED(Q, y)
      }
    }
  }

The general paradigm of using incremental delay trace is to initialize the arrival times and required times of a Boolean network using the recursive routines *ARRIVAL* and *REQUIRED*. Then, as circuits are modified, affected nodes are scheduled in either required time queue or arrival time queue. At the point where new timing information is needed, *UPDATE_ARRIVAL* and *UPDATE_REQIRED* are invoked.

As an example, we'll show how incremental delay trace can be used in the timing optimization algorithm *SPEED_UP* presented in the earlier sections of this chapter.

```
SPEED_UP'(η, d, ε)
   DELAY_TRACE(η)
   initialize AQ as arrival time queue
   initialize RQ as required time queue
   repeat {
      C = {x ∈ NODES(η)|S_x < ε}
      for each x ∈ C, compute W_x(d)
      CS = MWCS(η, ε)
      for each x ∈ CS {
         COLLAPSE(x, d)
         SCHEDULE_REQUIRED(RQ, x)
         for each y ∈ FI(x)
            SCHEDULE_ARRIVAL(AQ, y)
         TIMING_DECOMP(x)
      }
      UPDATE_ARRIVAL(AQ)
      UPDATE_REQUIRED(RQ)
   } until requirement is satisfied
   return η
```

Algorithm *SPEED_UP'* still has the basic structure of *SPEED_UP* with the addition of incremental delay trace. It initializes all the arrival times and required times in η using routine *DELAY_TRACE*. Then, in each *collapsing-decomposition* step, it schedules the node x in the required time queue and all inputs of collapsed node x in the arrival time queue. After all the nodes on the critical cutset are re-synthesized, the arrival times and required times are incrementally updated.

6.7 Future Work

Several open problems remain in our approach of technology independent timing optimization. In this chapter, the concept of ϵ-critical network is used to identify the critical region in the network. Then, the re-synthesis is done on the critical region alone without looking at the non-critical region. However, in real circuits, the slacks may vary continuously over a wide range. Thus, it is difficult, or sometimes infeasible, to separate critical nodes from non-critical nodes in a clear way. This greatly affects the way the weight of a node is computed. So, a new way of computing weights is needed which tries to identify the ease of re-synthesis at a node by looking at the distribution of arrival times of its transitive fanins.

In this chapter, we have presented an algorithm for decomposing optimally a level-0 kernel function into NAND gates and inverters. Since all the tree decompositions of a level-0 kernel function have the same area, the decomposition which gives the minimum output arrival time is the best. Some times, the function to be decomposed is not kernel-free. In this case, the objective of decomposition is to meet a certain output arrival time requirement while minimizing the area. Such an optimal decomposition routine can improve the quality of the results in our timing optimization system.

Incremental delay trace is essential to the performance of circuit re-structuring. However, a well-designed incremental delay trace system can be used in many other applications. The routines presented in this chapter form the core of the incremental delay trace system. To make it easy to use, several layers of abstraction are needed on top of the core. The top most layer interacts directly with naive users who simply turn on or off a switch. When the switch is on, the delay information is automatically kept up to date after every single change to the network. It is obvious that this is not always necessary. So, for more intelligent users who understand the principle of incremental delay trace, the second layers of routines should be provided which allows the users to do a serious modification to a network before updating the delay information to improve the efficiency. Below the second layer is a manager layer which directly interfaces to the core, maintaining various queues, catching changes to the network, and doing the appropriate scheduling.

Chapter 7

Summary and Conclusion

This thesis has addressed four individual problems in multi-level logic optimization: factoring logic functions, simplifying logic functions, phase assignment, and technology-independent timing optimization.

7.1 Factoring Logic Functions

The research on factoring logic functions consists of three basic parts. Direct manipulation of logic functions in their factored form, deriving factored forms from sum-of-products forms, and properties of optimum factored forms. The manipulation of logic functions consists of 1) basic logic operations, 2) operations pertaining to generating useful divisors of logic functions (similar to kernels of algebraic sum-of-products forms) on the factored forms directly, and 3) higher level operations. Algorithms have been given for the basic operations. Emphasis has been placed on generating useful divisors from factored forms directly. It has been shown that the kerneling algorithm can be extended to operate on factored forms. The divisors generated can be associated with kernels and possibly some redundant cubes. The implication has two sides. On one side, the redundant cubes can be used to find better (Boolean) common factors between functions. On the other side, not knowing which cubes are redundant may increase the network size unnecessarily. Nevertheless, this is an interesting result and deserves further investigation. Higher level operations have been shown to be reducible to the problem of simplying a factored form using a don't-care set also in factored form. This simplification problem is still open, but its solution may have a significant impact on function manipulations and on common factor

(Boolean) identification.

The second part focused on deriving factored forms from the sum-of-products forms of logic functions. The problem has been known to be difficult and exact algorithms, even with latest implementation techniques, are not practical. This thesis provided a spectrum of heuristic factoring algorithms which have different quality-versus-time tradeoffs. Certain minimality of factored forms have been defined and some of the algorithms have been shown to guarantee this minimality. The algorithms have been implemented and tested in the multi-level logic optimization system, MIS. Because the algorithms are so efficient, they are constantly used to estimate the size of the networks during optimization. More powerful algorithms (Boolean factoring) have been proposed and implemented. Results showed that fast heuristic algorithms perform just as well on most of the examples (almost all the functions from real circuits), except a few contrived examples. All the factoring algorithms presented in this thesis work on completely specified functions. Further investigation is needed to factor incompletely specified functions.

In the last part, properties of optimum factored forms were examined. In particular, a theorem has been given to state conditions under which optimum factoring of a completely specified function can be obtained by optimally factoring the "sub-functions". In factoring incompletely specified functions, this thesis studied conditions for a variable or a literal to be essential or redundant in the optimum factored forms. Criterias have been given to identify some of the essential or redundant cubes, variables or literals. These results can be used as part of an optimum factoring algorithm, or can be used to derive better lower bounds on the size of optimum factored forms which can be used to judge the quality of the results generated by heuristic algorithms.

7.2 Simplifying Logic Functions

In this part of the thesis, a simplification procedure called simultaneous Boolean substitution, was introduced. This technique uses existing two-level minimization programs in the context of multi-level logic optimization. With an example, it has been shown that simultaneous Boolean substitution not only reduces the number of minimizations to be performed, but also is able to obtain the result which can not otherwise be obtained by any ordering of single-function substitution.

The simplification procedure has been shown through experiments to be very time-

consuming, due to the large amount of don't-care information presented to the two-level minimizer. The emphasis of the work was to make the simplification procedure more efficient by trimming down the don't-care set. An exact filter has been derived which reduces the size of a don't-care set by removing the un-necessary cubes. No optimality is lost when the don't-care set is trimmed using this exact filter. Further studies have performed to control the size of the don't-care sets during their generation in the first place based on circuit topologies implied by the exact filter. The concepts of minimal and sufficient region of interest were introduced to identify a sub-network in which a function can get as much simplification as in the entire network.

The size of the don't-care set can also be reduced by choosing appropriate representations. In particular, it has been shown and experimentally verified that the fan-in don't-care set can be reduced by introducing new variables, and the fan-out don't-care can be reduced by representing it in its complemented form. Preliminary studies showed that it is possible to extend existing two-level minimizers to handle the don't-care set in this special representation without having to complement the don't-care set. The possible drawback is the increase of run time. Further investigation in this area is highly desirable.

Last, a deficiency of the simplification procedure has been identified which is due to the fact that existing two-level minimizers treat all the literals in an expression equally. A modification to the simplification procedure has been proposed to solve the problem without having to modify the existing two-level minimizers.

7.3 Phase Assignment

The global phase-assignment problem, also known as inverter minimization, is unique to multi-level logic. There were three components in this part of the thesis. First, phase-assignment problem has been formulated as an integer programming problem with linear cost function and non-linear constraints. The problem has been shown to be NP-complete. Only when the network is in a very specific form can the phase-assignment problem be solved optimally in polynomial time. A dynamic programming algorithm has been given to minimize the total number of inverters in a tree network.

To offer practical solutions to the phase-assignment problem, a spectrum of algorithms have been designed and implemented in MIS. Experimental results have shown that the algorithms are efficient and effective in reducing the size of the network. The algorithm

can be extended to reduce other cost function defined on the network, such as maximum delay.

When used in a cell-based design style, e.g., standard-cells or gate-array technology, the phase-assignment problem can be extended to allow more general modifications to functions than just complementations. The modification has been formally defined and the heuristic algorithm has been generalized to solve this generalized phase-assignment problem. Experiments are needed to justify the new formulation.

7.4 Technology-Independent Timing Optimization

Timing optimization studied in this thesis is restricted to technology-independent circuit re-structuring to reduce the "longest" path in the network. This approach uses global timing information in the network to identify critical sections of the network to re-structure while keeping the area increase at minimal. Re-structuring is done by collapsing the network selectively along the critical path and then re-decomposing the collapsed function to avoid creating longer critical path in the network.

The critical step of timing optimization is timing-driven decomposition. Several theorems have been stated to give conditions and algorithms for optimum timing decomposition.

Global timing optimization depends heavily on delay information, such as slacks or arrival times of signals given by a chosen delay model. These timing information has to be updated constantly throughout the optimization process. Incremental delay trace has been proposed which updates the timing information as the circuit changes, providing correct timing information in an efficient way. Preliminary implementation has shown that the payoff of incremental delay trace is significant. Formal implementation in MIS is desirable and may affect the performance of other optimization algorithms such as transistor-sizing, buffering, and delay-driven phase assignment.

Bibliography

[1] G. Adams, S. Devadas, C. Kring, F. Obermeier, P. Tzeng, A. R. Newton, A. Sangiovanni-Vincentelli, and C. Sequin. Module generation systems. In *ICCAD86*, Nov. 1986.

[2] Robert L. Ashenhurst. The decomposition of switching functions. In *Proc. Int. Symp. Theory of Switching Functions*, April 1959.

[3] K. Bartlett, R. Brayton, G. Hachtel, R. Jacobi, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Algorithms for multi-level logic minimization using implicit don't-cares. In *ICCD86*, Oct. 1986.

[4] K. Bartlett, R. Brayton, G. Hachtel, R. Jacobi, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Algorithms for multi-level logic minimization using implicit don't-cares. In *IEEE Trans. on ICCAD*, June 1988.

[5] K. Bartlett, W. Cohen, A. de Geus, and G. Hachtel. Synthesis and optimization of multilevel logic under timing constraints. *IEEE Trans. on CAD of IC*, Oct. 1986.

[6] K. A. Bartlett and G. D. Hachtel. Library specific optimization of multi-level combinational logic. In *ICCAD85*, Oct. 1985.

[7] D. Bostick, G. Hachtel, R. Jacoby, M. Lightner, P. Moceyunas, C. Morrison, and D. Ravenscroft. The boulder optimal logic design system. In *Int. Conf. on Computer Aided Design*, Nov. 1987.

[8] D. Brand. Hill climbing with reduced search space. In *Proc. IEEE Int. Conf. on Computer Aided Design*, Nov. 1988.

[9] R. Brayton. Factoring logic functions. *IBM Journal of Research and Development*, Mar. 1987.

[10] R. Brayton, N. Brenner, C. Cohen, G. Hachtel, C. McMullen, and M. Otten. The yorktown silicon compiler. In *ISCAS Proceedings*, June 1985.

[11] R. Brayton, R. Camposano, G. De Micheli, R. Otten, and J. van Eijndhoven. The yorktown silicon compiler. In D. Gajski, editor, *Silicon Complication*, Addison-Wesley, 1988.

[12] R. Brayton, J. Cohen, G. Hachtel, B. Trager, and D. Yun. Fast recursive boolean function manipulation. In *ISCAS Proceedings*, April 1982.

[13] R. Brayton, E. Detjens, S. Krishna, P. McGeer T. Ma, L. Pei, N. Phillips, R. Rudell, R. Segal, A. Wang, R. Yung, and A. Sangiovanni-Vincentelli. Multiple-level logic optimization system. In *Proc. IEEE Int. Conf. on CAD (ICCAD)*, Santa Clara, California, Nov. 1986.

[14] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Mis: a multiple-level logic optimization system. *IEEE Trans. on CAD of IC*, Nov. 1987.

[15] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic optimization and the rectangular covering problem. In *Proc. IEEE Int. Conf. on CAD (ICCAD)*, Nov. 1987.

[16] R. K. Brayton, G. D. Hachtel, Curt McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic publishers, 1984.

[17] R. K. Brayton and Curt McMullen. The decomposition and factorization of boolean expressions. In *Proc. Int. Symp. Circ. Syst. (ISCAS-82)*, Rome, May 1982.

[18] Robert K. Brayton and Curt McMullen. Synthesis and optimization of multistage logic. In *IEEE Int. Conf. on Computer Design (ICCD)*, 84.

[19] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, Aug. 1986.

[20] J. Darringer, D. Brand, J. Gerbi, W. Joyner Jr., and L. Trevillyan. Lss: a system for production logic synthesis. *Journal of the Association for Computing Machinery*, (5), Sept. 1984.

[21] Edward S. Davidson. An algorithm for nand decomposition under network constraints. *IEEE Transactions on Computers*, Dec. 1969.

[22] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Technology mapping in mis. In *Proc. IEEE Int. Conf. on CAD (ICCAD)*, Nov. 1987.

[23] J. Dussault, C. Liaw, and M. Tong. A high level synthesis tool for mos chip design. In *Proc. 22nd Design Automation Conf.*, June 1984.

[24] D. Gale and L. S. Shaply. College admissions and the stability of marriage. *Amer. Math. Monthly*, 1962.

[25] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1987.

[26] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel. Socrates: a system for automatically synthesizing and optimizing combinational logic. In *Design Automation Conference*, June 1986.

[27] M. E. Hofmann. Automated synthesis of multi-level combinational logic in cmos technology. In *Ph.D. thesis, UC Berkeley*, 1985.

[28] S. J. Hong, R. G. Cain, and D. L. Ostapko. Mini: a heuristic approach for logic minimization. *IBM Journal of Research and Development*, Sept. 1974.

[29] T. Hoshino, M. Endo, and O. Karatsu. An automatic logic synthesizer for integrated vlsi design system. In *Proc. Cust. Int. Circ. Conf. (CICC)*, May 1984.

[30] R. M. Karp. *Reducibility among combinatorial problems.* Plenum Press, New York, 1972.

[31] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, Feb. 1970.

[32] K. Keutzer and M. Vancura. Timing optimization in a logic synthesis system. In *Proc. International Workshop on Logic and Architectural Synthesis*, May 1987.

[33] Hung Chi Lai and Saburo Muroga. Automated logic design of mos networks. *Advances in Information Systems Science*, 1970.

[34] Eugene Lawler. *Combinatorial Optimization.* Holt Rinehart Winston, 1976.

[35] Eugene L. Lawler. An approach to multilevel boolean minimization. *Journal of the Association for Computing Machinery*, July 1964.

[36] Abdul A. Malik, Robert K. Brayton, A. Richard Newton, and Alberto L. Sangiovanni-Vincentelli. A modified approach to two-level logic minimization. In *Proc. IEEE Int. Conf. on CAD (ICCAD)*, Nov. 1988.

[37] E. J. McCluskey. Minimization of boolean functions. *Bell Lab. Technical Journal*, Nov. 1956.

[38] P. McGeer and R. Brayton. Efficient, stable algebraic operations on logic expressions. In *Proceedings of the International Conference on Very Large Scale Integration*, Aug. 1987.

[39] M.G.Karpovsky. Multilevel logic networks. *IEEE Transaction on Computers*, Feb. 1987.

[40] A. R. Newton, A. Sangiovanni-Vincentelli, and C. Sequin. The berkeley synthesis project. In *ICCAD86*, Nov. 1986.

[41] W. V. Quine. A way to simplify truth functions. *Am. Math. monthly*, Nov. 1955.

[42] P. J. Roth and R. M. Karp. Minimization over boolean graphs. *IBM Journal of Research and Development*, April 1962.

[43] R. Rudell. *BLIF Reference Manual.* May 1986.

[44] R. Rudell and A. Sangiovanni-Vincentelli. Exact minimization of multiple-valued functions for pla optimization. In *Proc. IEEE Int. Conf. on CAD (ICCAD)*, 1986.

[45] R. Rudell and R. Segal. *BDSYN Users Manual.* April 1986.

[46] Alex Saldanha, Albert Wang, Robert Brayton, and Alberto Sangiovanni-Vincentelli. Multi-level logic simplification using don't cares and filters. In *to appear in Proc. IEEE Design Automation Conference*, June 1989.

[47] T. Sasao. Macdas: multi-level and-or circuit synthesis using two-variable function generators. In *Design Automation Conference*, June 1986.

[48] Claude. E. Shannon. The synthesis of two-terminal switching circuits. *Bell Lab. Technical Journal.*

[49] L. Trevillyan, W. Joyner, and L. Berman. Global flow analysis in automatic logic design. *IEEE Transactions on Computers*, Jan. 1986.

[50] G. Whitcomb. Exact factoring of incompletely specified functions. In *ee290ls class project report, UC. Berkeley*, May 1988.