On the Testability-Preserving Decomposition and Factorization of Boolean Expressions

Jagadeesh Vasudevamurthy
M Tech (Indian Institute of Technology, Kharagpur)

Department of Electrical Engineering
McGill University

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

May 1991
© Jagadeesh Vasudevamurthy
Abstract

This thesis presents a new concurrent method for the decomposition and factorization of Boolean expressions based on two simple objects: two-literal single-cube divisors and double-cube divisors along with their complements. It is proved that the presence of common multiple-cube algebraic divisors, from a set of Boolean expressions, can be found by analyzing the set of double-cube divisors. It is also shown that in order to find the duality relations that may exist between various objects, only a subset of two-literal single-cube and double-cube divisors needs to be analyzed. Since the number of these objects grows polynomially with the size of the network, the number of objects that are to be analyzed for finding common algebraic divisors, and for finding the duality relations between them, is much less than the set of all algebraic divisors. Also, since the duality relations between these objects are exploited along with DeMorgan's laws, these objects constitute a richer set of divisors than the strictly algebraic divisors.

It is also proved that the transformations based on these simple objects preserve testability. This result implies that if the input Boolean network before decomposition and factorization is 100% testable for single stuck-at faults by a test set \( T \), then the area optimized output network will also be 100% testable for single stuck-at faults, and can be tested by the same test set \( T \). These results are proved using the concepts of corresponding faults in the circuits and relations between complete test sets. Since the method assumes that the initial network is only single stuck-at fault testable, and because single stuck-at fault testability is maintained through the transformations, the method can be applied to a large class of irredundant two-level and multi-level circuits to synthesize fully testable circuits.

Experimental results are presented and compared with various logic synthesis systems to demonstrate the efficiency and effectiveness of the method.
Résumé

Cette thèse présente une nouvelle méthode d'analyse pour la décomposition et la factorisation des expressions Booléennes. Cette méthode se base sur deux types d'objets élémentaires : les diviseurs à simple cube et à deux littéraux, et les diviseurs à double cubes et leurs compléments. Il est prouvé que la présence de diviseurs algébriques communes à multiples cubes, à l'intérieur d'un ensemble d'expressions Booléennes peut être détectée en analysant l'ensemble des diviseurs à double cubes. Il est également montré que, pour trouver les relations de dualité pouvant exister entre différents objets, il est nécessaire d'analyser uniquement un sous-ensemble comprenant des diviseurs simple cube à deux littéraux et des diviseurs à double cubes. Comme le nombre de ces objets augmente de façon polynomiale avec la taille du réseau, le nombre d'objets à analyser pour trouver le diviseur algébrique commun et pour trouver les relations de dualité entre ceux-ci, demeure beaucoup plus faible que l'ensemble des diviseurs algébriques. Également, comme les relations de dualité entre ces objets sont exploitées conjointement avec les lois de DeMorgan, ces objets constituent un ensemble de diviseurs plus riche que les diviseurs strictement algébriques.

Il est aussi prouvé que les transformations basées sur de tels objets élémentaires préservent la testabilité. Ce résultat implique que si le réseau Booléen initial est 100% testable pour les fautes simples "collé-à", par un ensemble de test $T$ avant la décomposition, alors le réseau final avec surface optimisée sera également 100% testable pour les fautes simples "collé-à" et pourra aussi être testé avec le même ensemble de test $T$. Ces résultats sont prouvés en utilisant les concepts de fautes correspondantes dans le circuit et de relations entre ensembles de test complets. Puisque la méthode présuppose que le réseau initial est uniquement testable pour les fautes simples "collé-à" et a cause que la testabilité des fautes simples "collé-à" est conservée au travers des transformations, la méthode peut être appliquée à un grand nombre de circuits non-redondants à deux ou multiples niveaux pour la synthèse de circuits complètement testables.

Les résultats expérimentaux sont présentés et comparés avec ceux provenant de multiples systèmes de synthèse logique. Ceci permettant de démontrer l'efficacité et le bon rendement de la méthode.
I would like to express my deep sense of gratitude to my thesis supervisor, Professor Janusz Rajski, for his invaluable guidance, constant encouragement, and support during the course of this research. He taught me how to do research, how to present ideas, how to see things in more than one way, and above all how to survive and work in crisis. His feedback over the years has been instrumental behind my development as a researcher and as an individual.

I also gratefully acknowledge my Ph.D. committee members, Professor Nicholas Rumin and Professor Hossam ElGindy, for their help in organizing this thesis. Thanks are due to Professor Vinod Kumar Agarwal for introducing me to Professor Janusz Rajski, and to the VLSI Design Laboratory.

I am grateful to Trang Nguyen, Lynn-Marie Holland and Jacek Slaboszewicz for their co-operation and the help that they offered me. I am indebted to my colleague Mr. Robert Treuer for proof-reading the various drafts of this thesis, and to Mr. Charles Arsenault for writing the French version of the abstract.

This research is supported by the Canadian Commonwealth and Fellowship Plan of Canada, the Microelectronics Fund of NSERC, Canada, and the Canadian Microelectronics Corporation. I am extremely grateful to them.

On a personal level, I would like to thank my late father, V Vasudevamurthy, and my mother, Radhabai, who sacrificed their whole lives to provide me with the best possible education. Finally, I would like to thank my wife, Geetha, for her patience and love.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Résumé</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td>Contents</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>v</td>
</tr>
<tr>
<td>Claim of Originality</td>
<td></td>
</tr>
</tbody>
</table>

### Chapter 1  Introduction

1.1 Design methodology based on logic synthesis.  
1.2 Preliminaries  
1.3 Review of decomposition and factorization algorithms  
1.4 Research motivation and the approach  
1.5 Organization of the thesis  

### Chapter 2  Double cubes and two-literal single cubes in decomposition and factorization

2.1 Single-cube and double-cube divisors  
2.2 Duality relations between objects  
2.3 Common algebraic divisors between expressions  

### Chapter 3  Concurrent decomposition and factorization algorithm

3.1 Generation of double-cube divisors  
3.2 Generation of single-cube divisors  
3.3 Substitution of a cube divisor  
3.4 Recomputation of double-cube divisors  
3.5 Decomposition and factorization of multi-output circuits  

### Chapter 4  Evaluation of concurrent decomposition and factorization algorithm

4.1 Comparison with weak division approach  
4.2 Comparison with MIS2.1  

v
4.3 Output phase assignment in multi-level logic decomposition
4.4 Boolean decomposition in multi-level logic optimization
4.5 Comparison with BOLD
4.6 Technology mapping
4.7 The effect of limiting sizes of double-cube divisors

Chapter 5 Testability-preserving transformations
5.1 Testability preservation
5.2 Testability preservation of single-cube extraction
5.3 Testability preservation of double-cube extraction
5.4 Testability preservation of dual expression extraction
5.5 Testability preservation of multi-output networks
5.6 Testability related properties of synthesized circuits
5.7 Comparison of single-output and multi-output minimization

Chapter 6 Conclusions

Bibliography
List of Figures

1.1 Role of synthesis tools in VLSI design ........................................ 3
1.2 Two-level logic ............................................................................. 5
1.3 Multi-level logic ........................................................................... 6
1.4 A multi-level realization of an 8-bit parity checker ...................... 6
1.5 Implementation of the factored form as a complex gate ............... 10
1.6 Output fault detection .................................................................. 15
1.7 Kernel tree ................................................................................... 20
2.1 Existence of duality between expressions ..................................... 32
4.1 Script used to generate columns 3 and 5 of Table 4.2 ....................... 49
4.2 Script used to generate columns 4 and 5 of Table 4.4 ....................... 52
4.3 Script used to generate column 2 of Table 4.9 .............................. 59
5.1 Transformation of networks .......................................................... 63
5.2 Division by a single cube .............................................................. 65
5.3 Division by a double cube ............................................................ 67
5.4 Double-cube divisors $d$ and their complements $\bar{d}$ ..................... 69
5.5 Dual expression extraction ........................................................... 71
5.6 Resubstitution in a multi-output network ................................... 72
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Local transformations</td>
<td>21</td>
</tr>
<tr>
<td>2.1</td>
<td>Double-cube divisors and their bases</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Performance comparison between WDIVA and Pendulum</td>
<td>48</td>
</tr>
<tr>
<td>4.2</td>
<td>Performance comparison between Pendulum and MIS21</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>Area minimization of multi-level circuits</td>
<td>51</td>
</tr>
<tr>
<td>4.4</td>
<td>Performance comparison between MIS21, OPAM and Pendulum</td>
<td>53</td>
</tr>
<tr>
<td>4.5</td>
<td>Comparison of Pendulum with algebraic and Boolean script</td>
<td>54</td>
</tr>
<tr>
<td>4.6</td>
<td>Comparison of Pendulum with various logic synthesis systems</td>
<td>56</td>
</tr>
<tr>
<td>4.7</td>
<td>Technology description</td>
<td>57</td>
</tr>
<tr>
<td>4.8</td>
<td>Two-input equivalent gates required for mapping</td>
<td>58</td>
</tr>
<tr>
<td>4.9</td>
<td>MCNC89 circuits optimization results</td>
<td>59</td>
</tr>
<tr>
<td>4.10</td>
<td>Percentage improvement in performance between MIS21 and Pendulum</td>
<td>60</td>
</tr>
<tr>
<td>4.11</td>
<td>Effect of limiting sizes of divisor on single-output minimized apex4 benchmark</td>
<td>61</td>
</tr>
<tr>
<td>5.1</td>
<td>Results on PLAs that are prime and irredundant with respect to every output</td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>Results on multiple output minimized PLAs</td>
<td>76</td>
</tr>
</tbody>
</table>
Claim of Originality

This dissertation presents new concepts, algorithms and results in the area of synthesis of testable multi-level circuits, as follows.

Basic objects for decomposition and factorization. Two simple objects are identified in this thesis which can be used as candidate subexpressions in decomposition and factorization. These objects are two-literal single-cube divisors and double cube divisors along with their complements. These objects can be used to reason about common algebraic divisors and duality relations between expressions. It is demonstrated that reasoning about the presence of common multiple-cube algebraic expressions from a set of Boolean expressions can be performed by analyzing only the set of double-cube divisors. Since the number of double-cube divisors grows polynomially with the number of cubes of a function, the set that must be analyzed to check for common multiple-cube divisors is much smaller than the set of all algebraic divisors. It is also shown that in order to find the duality relations that may exist between various objects, only a subset of two literal single-cube and double-cube divisors needs to be analyzed. Also, since the duality relations between objects along with DeMorgan's laws are exploited, the objects proposed constitute a non-set of divisors than the strictly algebraic divisors as objects.

A new concurrent decomposition and factorization algorithm. Using the above two objects, a greedy decomposition and factorization procedure is presented. The method provides the ability to compute the weight (which indicates the number of literals that can be saved) associated with these objects accurately and dynamically. As a consequence, the method is entirely greedy and during each iteration it extracts the best two-literal single-cube divisor or double-cube divisor along with its complement.

Testability preservation: Various transformations based on single cubes and double cubes along with their complements, are identified and it is shown that single fault testability with respect to a test set is maintained by the decomposition and factorization procedures. This result implies that if an input Boolean network before decomposition and factorization is 100% testable for single stuck-at faults by a test set \( T \), then the area-optimized output network is also 100% testable for single stuck-at faults and can be tested by the same test \( T \). Since the method assumes that the initial network is only single
stuck at fault testable, and because single stuck-at fault testability is maintained through the transformations, the method can be applied to a large class of irredudant two-level or multi-level circuits to synthesize fully testable circuits. These results are proved using the concepts of corresponding faults in the circuits and relations between complete test sets.

Extensive experimental comparisons of the performance of the concurrent decomposition and factorization procedure with other well known techniques are also provided.
Chapter 1

Introduction

Logic synthesis is a formal approach in which a well-defined sequence of steps is executed to arrive at a hardware implementation of a given design specification [Newton86, NewSan87, Trevi87]. This is in contrast with the conventional procedure of iterative design improvements based on heuristics and the designer's experience. The goal of logic synthesis is to accept functional specifications for a hardware unit and generate automatically a detailed, technology-specific implementation comparable in quality to that of an experienced engineer. Some of the criteria that are employed to evaluate an automatic logic synthesis system are as follows [Trevi87]

Correctness: The prerequisite for any logic synthesis system is that it must reliably generate a functionally correct representation.

Area. The classical measure of effectiveness of any logic synthesis system is the final area of the synthesized logic. The objective is to make the logic comparable in area to that which can be achieved by the best manual design.

Speed: The final logic produced by any logic synthesis system must meet the requirements on the minimum and maximum lengths of paths through the logic. Reducing the chip area reduces cost, but reducing the critical path delay speeds up the logic.

Testability considerations As logic synthesizers produce large and complex designs, the issue of testing the resulting logic becomes increasingly critical. The huge amounts of logic that can be created in minutes pose enough of a testing challenge, but the designer's lack of familiarity with the synthesized logic makes it even more difficult to create appropriate test vectors. The only practical way to thoroughly test large, synthesized
Design methodology based on logic synthesis

The primary consideration in the entire logic synthesis process is the quality of the resulting circuit. This quality in logic synthesis is measured by how closely the circuit-to-be-synthesized meets the desired speed, chip area, and testability goals. In practice, however, all three of these constraints conflict. Although it might be desirable to build a given circuit that is both small and fast, the circuit's area typically trades off with its speed [deGeus89]. The design of the optimal circuit which meets all of these constraints is a difficult problem due to the tremendous number of potential solutions for even a small circuit.

1.1 Design methodology based on logic synthesis

Logic circuits are designed in several steps as shown in Figure 1.1 [BohGre88, McLeod88, deGeus89]. The designer begins by describing a design behavior. Hardware description languages, Boolean equations, and circuit diagrams capture the functionality of the design. These methods are intended to capture functionality rather than implementation. Once the functionality has been thoroughly verified through simulation, the process of design and implementation begins. The designer reformulates the design in terms of registers, arithmetic units, storage registers, and combinational logic expressed in terms of Boolean equations. Although a chunk of combinational logic typically constitutes only 20% of a chip's area, creating it can take 80% of the total time to design the chip [Sangio85]. Coming up with the right logic structure for a design can be difficult and time-consuming, but will have a significant effect on the overall quality of a design. The resulting description is called a register-transfer level (RTL), since the equations describe how the data is transferred from one register to another.

Strictly speaking, only the steps after the RTL description constitute logic synthesis. Unlike silicon compilers, which generate completely laid-out chip designs from a very high-level description, logic synthesizers take a design from Boolean expressions to a netlist. Only some parts of the RTL description can be synthesized, mainly the combinational logic. The nonsynthesized parts generally include such predesigned cells as memories, arithmetic logic units, and data path blocks, all typically obtained from libraries or generated by module compilers.
Figure 1.1  Role of synthesis tools in VLSI design

Logic synthesis requires several operations to transform an RTL description into a physically realizable logic design. These operations include minimization of Boolean equations, decomposition and factorization of Boolean equations, and technology mapping [BrHaSa90].

Logic minimization is used to detect and eliminate redundant portions of the logic [BrHaMS84, DaAgRu86, BaBrHJ88]. The benefits of logic minimization are potential reduction in chip area and decrease in signal propagation time. In addition, redundant logic is not testable and the presence of faulty redundant logic may invalidate the detection of faults elsewhere in a circuit. Hence, eliminating redundant logic can improve the testability
Decomposition and factorization can be performed on a design whenever it has subfunctions that appear in more than one place [BraMcM82, BraMcM84]. When a design is decomposed, common subfunctions are identified and implemented only once. The outputs of the common subfunctions are shared across the entire design. As a consequence, a number of identical components are consolidated into one component. Hence a complicated logic function can be implemented as a multistage circuit. This thesis is concerned with the decomposition and factorization algorithms, where the main emphasis in the algorithm is on area optimization.

During technology mapping, minimized equations are mapped into a set of gates. The nonsynthesized portions of the logic are also mapped into technology-specific implementations at this point.

1.2 Preliminaries

The definitions that are used in this thesis are provided in Section 1.2.1 and are consistent with definitions in [BraMcM82, BrRSW87a, BrHaSa90]. The need for multi-level logic in digital design is explained in Section 1.2.2. The representation of multi-level logic used in this thesis is given in Section 1.2.3. The problem of decomposition and factorization that generates multi-level logic is introduced in Section 1.2.4 and 1.2.5. The role of algebraic manipulation and Boolean manipulation in decomposition and factorization are explained in Section 1.2.6.

1.2.1 Basic definitions

A literal is a variable or its negation. For example, \( x \) and \( \bar{x} \). A cube is a set \( C \) of literals such that \( x \in C \) implies \( \bar{x} \not\in C \). A cube represents the conjunction of the literals. For example, \( \{a, b, \bar{c}\} \) is a cube and \( \{a, \bar{a}\} \) is not a cube. A sum-of-products (SOP) form, also called an expression, is a set of cubes. An expression represents the disjunction of its cubes. An expression is nonredundant if no cube in the expression properly contains another. A Boolean expression \( f \) is a nonredundant expression. The support of an expression, written as \( \text{sup}(f) \), is the set of variables \( x \) such that either \( x \in C \) for some \( C \in f \) or \( \bar{x} \in C \) for
some $C \in f$. Informally, it is the set of variables that $f$ is defined over. For example, $sup(ab + c) = \{a, b, c\}$.

### 1.2.2 Two-level versus multi-level logic

![Two-level logic](image)

Figure 1.2 Two-level logic

Logic represented as a sum-of-products or as a truth table is two-level logic. Figure 1.2 illustrates the two-level logic. The first level consists of AND gates which corresponds to product terms in the sum of products. The second level consists of OR gates, one for each output.

The minimum sum-of-products form of a Boolean expression is not always the most economical to implement. Consequently, common terms are often factored, with the result that a cheaper realization is obtained. Two-level circuits that realize factored forms are multi-level circuits. As the name implies, multi-level logic may have more than two-levels. Figure 1.3 shows a multi-level implementation of Figure 1.2. The first level of gates processes primary inputs to produce intermediate nodes. Successive levels of logic...
take in both primary inputs and previously generated intermediate nodes to produce new intermediates and primary outputs.

Multi-level logic, also called random logic, is often more economical to implement than two-level logic, and multi-level timing performance may be better [BrHaSa90]. For Boolean functions such as parity functions or comparator functions, multi-level representations can give a logarithmic reduction in the size of representation over sum-of-products representations. Additionally, logic gates have size and fan-in restrictions that prohibit two-level implementations of most functions.
Example 1: A striking example of the tradeoff between logical complexity and speed (number of levels) is the parity check function as follows:

\[ f = x_1 \oplus x_2 \oplus \ldots \oplus x_n. \]

This function has a value of 1 if an odd number of input variables are 1 and has a value of 0 if an even number of input variables are 1. A 2-level sum-of-products realization of an \( n \)-variable parity function has \( 2^{n-1} \) \( n \)-input AND gates and a single \( 2^{n-1} \)-input OR gate. This function can be realized by connecting \( (n-1) \) two-input exclusive-or circuits in the form of a tree with \( \lceil \log_2 n \rceil \) levels as shown in Figure 1.4. Such a circuit has \( 3 + \lfloor \log_2 n \rfloor \) levels, but has only \( 3(n-1) \) two-input gates and \( 2(n-1) \) inverters, and has fan-out and fan-in indices equal to two.

1.2.3 Representation of multi-level logic

Any combinational logic can be represented by a Boolean network [BrDeKM86, BrRSW87a, BrHaSa90]. Boolean network \( \eta \) is a directed acyclic graph (DAG) such that for each node \( n_i \), there is an associated Boolean expression \( f_i \), expressed in sum-of-products form and a Boolean variable \( y_i \) representing the output of \( f_i \). There is a directed edge from node \( n_i \) to \( n_j \) if \( f_j \) explicitly depends on \( y_i \) or \( \bar{y}_i \). Furthermore, some of the variables in \( \eta \) may be classified as primary inputs and primary outputs.

The Berkeley Intermediate Interchange Format, BLIF, is used to describe a Boolean network in textual form [BrDeKM86].

Example 2: The BLIF representation of the multi-level logic in Figure 1.3 is as follows:

```
.model Example
.inputs a b c
.output f
.names a b M
10 1
01 1
.names c M f
10 1
```
1.2.4 Decomposition

Decomposition of a network deals with translating a two-level or a very shallow multi-level network to a deeper multi-level network [BraMcM82]. The main objective of decomposition is to find repeated sub-expressions so that these sub-expressions are implemented once but used as often as needed. Generally, decomposition increases the number of levels of logic while decreasing the average fan-in per gate.

The operation of decomposition on a network is the process of creating some intermediate functions and variables, and then re-expressing the original functions as new functions of the original as well as the intermediate variables [BrHaSa90]. Let set \(\mathcal{U} = \{u_1, u_2, \ldots, u_k\}\) of variables, be a description of Boolean equations as a Boolean network \(\eta\) and a non-negative integer \(K\). The problem of decomposition is to find an equivalent Boolean network that contains \(K\) or fewer literals.

Example 3. Consider a two-level circuit having four primary inputs, \(1..4\), and four primary outputs, \(f_1, \ldots, f_4\), as follows. The process of translating

\[
\begin{align*}
  f_1 &= \overline{1}23 + 1234 + \overline{1}234 \\
  f_2 &= \overline{1}234 + 1234 + 234 \\
  f_3 &= 1234 + 1234 + 134 \\
  f_4 &= \overline{1}24 + 1234 + \overline{1}234
\end{align*}
\]

result in

\[
\begin{align*}
  f_1 &\equiv 37 \\
  f_2 &\equiv 29 \\
  f_3 &\equiv 110 \\
  f_4 &\equiv 48 \\
  5 &\equiv 1\overline{2} + 1\overline{2}
\end{align*}
\]
is decomposition.

The associated optimization problem is to find a set of intermediate functions such that the resulting network is minimal, measured in either area or delay [BrHaSa90].

An optimal solution to the decomposition problem is described in Section 13.1.

1.2.5 Factorization

Factorization is the process of deriving a factored form from a sum-of-products form of a function [Brayt87a, Brayt87b]. Let \( E \) be a Boolean expression having \( l \) literals. The problem of factorization is to find an equivalent Boolean expression that contains \( l \) or fewer literals.

**Example 4:** The sum-of-product expression

\[
f = a\bar{b} + a\bar{c} + b\bar{a} + b\bar{c} + c\bar{a} + c\bar{b}
\]

can be factored as

\[
f = (a + b + c)(\bar{a} + \bar{b} + \bar{c}).
\]

The associated optimization problem is to find a factored form with the minimum number of literals [BrHaSa90].

The operation of decomposition is similar to factorization except that each sub-expression is formed as a new intermediate variable and substituted into the functions
Figure 1.5 Implementation of the factored form as a complex gate being decomposed. Both decomposition and factorization are concerned with identifying common sub-expressions and rewriting logic functions in factored form.

Factored forms have many attractive properties [BrRSW87a, Brayt87a, Brayt87b]. 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 1.5. The example also illustrates that the factored form specifies not only the number of transistor pairs in the gate, but also the way in which they are connected. The number of literals in the factored form corresponds to the number of transistors needed to implement the function as a complex gate. Hence factored forms are useful in estimating the area and delay in a multi-level logic synthesis and optimization system.

A method to find an optimal factored form is given in Section 1.3.2.

1.2.6 Algebraic and Boolean Transformations

Algebraic transformations result in logic equations, that when multiplied are identical to the original form. Boolean transformations create logic functions that are equivalent, but may not necessarily have the same equations.

Let $T$ be a transformation such that given two expressions, $f$ and $p$, it generates $q$ and $r$ such that

$$f = pq + r.$$

Expressions $p$ and $q$ have disjoint support if $\text{sup}(p) \cap \text{sup}(q) = \emptyset$ and the product $pq$ is
1.3 Review of decomposition and factorization algorithms

An exact solution to the problem of decomposition is proposed in [RotKar62] if carried to completion. The algorithm will yield a minimal area circuit. A systematic approach to the determination of minimal (literal-cost) forms of a general Boolean function that has more than two levels is proposed in [Lawler64]. The above two techniques provide an optimum solution to the decomposition and factorization problems and are discussed in

Example 5 Consider the switching function

\[ f = ab + bc + a\bar{c}d \]

There are three ways of transforming \( f \). The first two ways are apparent and are as follows

1. \( f = a(b + \bar{c}d) + bc \)
2. \( f = b(a + c) + a\bar{c}d \)

Let us add a superfluous term \( c\bar{c}d \) to \( f \) as follows.

3. \( f = ab + bc + a\bar{c}d + c\bar{c}d = (a + c)(b + \bar{c}d) \)

Algebraic transformations are used in methods 1 and 2 above, whereas a Boolean transformation is used in method 3. Boolean transformations are desirable to perform, as they usually result in equations with a smaller literal count, but they are computationally expensive [Brayt87a, DeWaNS89]. In this case, the resulting \( f \) has the same truth table as the original \( f \) but by multiplying back we may not get the original equation. Hence Boolean transformations create logic functions that are equivalent, but may not necessarily have the same equations. As shown in Example 4, a Boolean transformation creates a don't care condition. Boolean transformations use the identity of Boolean algebra such as \( aa = 0 \), \( aa = a \) and \( a + \bar{a} = 1 \). Boolean procedures are typically slower than algebraic procedures and hence must be used discriminantly [Brayt87a, Brayt87b]

1.3 Review of decomposition and factorization algorithms

An exact solution to the problem of decomposition is proposed in [RotKar62] if carried to completion. The algorithm will yield a minimal area circuit. A systematic approach to the determination of minimal (literal-cost) forms of a general Boolean function that has more than two levels is proposed in [Lawler64]. The above two techniques provide an optimum solution to the decomposition and factorization problems and are discussed in
Sections 1.3.1 and 1.3.2 However, the optimum solution to the problem of decomposition and factorization is essentially enumerative and impractical to use for today’s VLSI circuits. It is shown in Section 1.3.3. that the decomposition and factorization problems, and the problem of verifying functional equivalence of two Boolean networks are difficult problems [GarJoh79].

Since exact solutions are impractical, the current research is focussed on developing heuristic algorithms. A general heuristic algorithm that can capture various decomposition and factorization algorithms [BraMcM82, DieSch65, VasRa90b] is presented in Section 1.3.4. It is evident from this algorithm that the solution to the problem of decomposition and factorization consists of finding and substituting common sub-expressions in a Boolean network. The area of the final logic and the time to synthesize the logic depends on candidate sub-expressions. An algorithm which employs single-cubes as candidate sub-expressions is reviewed in Section 1.3.5. An algorithm which employs cube-free multiple-cube divisors and single-cube divisors as candidate sub-expressions is reviewed in Section 1.3.6. The advantages and disadvantages of using different objects for decomposition and factorization are studied. A method based on local transformations is reviewed in Section 1.3.7.

The optimization criterion in all the above algorithms is to minimize the area and testability is only considered as a side effect of a less redundant implementation [BrRSW87a]. In Section 1.3.8, a method for synthesizing a Boolean network that is 100% testable, and which generates test vectors as a by-product of minimization, is reviewed. A method which employs testability as an optimization criterion, rather than area is reviewed in Section 1.3.9.

1.3.1 An exact method for deriving minimal area networks

The form of the minimization problem considered by the authors in [RotKar62] is as follows. Let \(\beta\) be a prescribed set of primitive Boolean functions. Associate with each a positive integer called its cost. Let the cost of a expression be the sum of the costs of the primitive functions describing it. Let \(f\) and \(d\) be Boolean functions. The problem is to devise an efficient algorithm for finding an expression \(F\), having a minimum cost, composed of primitives from \(\beta\) such that \(f \rightarrow F \rightarrow f \lor d\). One can recognize that the above problem describes what might be termed the general problem of logical design of circuits with one
output and no feedback, where the set \( \mathcal{J} \) corresponds to a set of primitive logical blocks each with a primitive cost and each with one output, and where the Boolean function \( f \) expresses don't care conditions of the circuit represented by \( f \). The problem then is to design an equivalent circuit of minimum cost using only primitive circuits.

In [RotKar62], decomposition theory [Ashenh59] is extended so that it becomes a general approach to the synthesis of gate-type combinational circuits. The algorithm, for any given partial or total switching function, finds a decomposition which satisfies arbitrary restrictions on the primitive functions used in the decomposition (equivalently, the primitive elements employed in the circuit) and, for any given assignment of costs to the primitive functions, has minimal cost. Primitive functions are considered as potential decomposition functions. The possible decompositions are ordered alphabetically and tried in turn at each step of the algorithm. A heuristic algorithm is also proposed which would direct decomposition at each step. This algorithm is essentially enumerative, but because it employs a number of useful heuristics and some efficient tests for decomposition, it finds minimal circuits for functions up to five or six variables.

### 1.3.2 An exact method for deriving minimal factored forms

A systematic approach to the determination of minimal (literal-cost) forms of a general Boolean function that have more than two levels, and also of their absolute minimal form(s) is proposed in [Lawler64]. This is the only known approach for determining a factored form with a minimum number of literals. Minimal expressions of either the sums of \( N \)-level forms, \( sN \), or products of \( N \)-level forms, \( pN \), can be obtained by the method developed in this paper. The logical approach to the minimization problem can be considered as a generalization of an algorithm for two-level AND-OR synthesis. Certain functions called the prime \( p(N - 1) \)-implicants (prime \( s(N - 1) \) implicants) are preselected. It is shown that the minimal \( sN \)-forms (\( pN \)-forms) are the sum (product) of prime \( p(N - 1) \) implicants (prime \( s(N - 1) \)-implicants). Thus it is clear that preselection of the prime \( p(N - 1) \) implicants is analogous to the determination of the set of prime implicants in two-level minimization. A minimal \( sN \)-form is then obtained by choosing a least-cost subset of the candidates which cover the function. The selection phase is analogous to the selection of the set of prime implicants in two-level minimization. Finally, it is shown that absolute minimal forms can be obtained by carrying out multi-level minimization to a sufficient number of levels, \( N \).
However, the computational complexity of deriving a minimal cover for a function increases tremendously with the number of factors. The complexity of this technique would appear to make it impractical to use for all but the smallest functions.

### 1.3.3 Complexity results of decomposition and factorization problems

An exact algorithm for the decomposition and factorization problem was given in the previous sections. It is evident that both of the methods are essentially enumerative and try all the possible combinations of candidate sub-expressions to find an exact answer. These algorithms are known to have an asymptotic computing time that is exponential in the size of the input problem. Hence these algorithms are computationally feasible only for very small problems. In this section, it is shown that both the decomposition and factorization problems are computationally related to problems such as tautology and satisfiability. Thus, the decomposition and factorization problems can be solved in polynomial time if and only if the above problems can also be solved in polynomial time. Hence it would appear unlikely that the decomposition and factorization problems have a polynomial algorithm. Certainly the search for an efficient, exact algorithm should not be attempted and it is more appropriate to concentrate on other, approximate answers. In this section, several problems commonly encountered in this thesis are investigated, and their complexities are given.

**MINIMUM BOOLEAN NETWORK**

**INSTANCE** Set $U = \{u_1, u_2, ..., u_k\}$ of variables, a description of a Boolean function as a Boolean network $\eta$ and a non-negative integer $K$.

**QUESTION** Is there an equivalent Boolean network that contains $K$ or fewer literals?

**Comment:** Transformation from TAUTOLOGY and remains CO-NP-hard.

**Reference:** [KeuRic89]

**MINIMUM EQUIVALENT EXPRESSION**
INSTANCE: A well-formed Boolean expression $E$ involving literals on a set $V$ of variables, the constants TRUE and FALSE, and the logical connectives $\lor$ (disjunction), $\land$ (conjunction), $\neg$ (not), and $\rightarrow$ (implies), and a non-negative integer $K$.

QUESTION: Is there a well-formed Boolean expression $E_1$ that contains $K$ or fewer occurrences of literals such that $E_1$ is equivalent to $E$, that is, such that for all truth assignments to $V$ the truth values of $E_1$ and $E$ agree?

Comment: Transformation from SATISFIABILITY and remains NP hard.

Reference: [GarJoh79]

Another problem of interest is to find the Boolean equivalence between two networks. The problem is of practical importance, since even after finding a minimal network it is necessary to verify the final network with the initial network for functional equivalence. In order to see the computational complexity of this problem, consider two Boolean networks $B_1$ and $B_2$ having $n$ inputs, $1..n$, and a single output $f$. One way of verifying that $B_1$ and $B_2$ are not functionally equivalent is to find a test vector in the circuit of Figure 1.6 that produces a '1' on the output $p$. This problem is studied in [IbaSah75] and is as follows.

OUTPUT FAULT DETECTION
1.3 Review of decomposition and factorization algorithms

INSTANCE: A combinational circuit C

QUESTION Is there a fault on the output line that can be detected by I/O experiments?

Comment: Transformation from NON-TAUTOLOGY and remains NP-hard

Reference: [IbaSah75].

Since exact decomposition and factorization are impractical, the research is focused on developing heuristic algorithms. Techniques based on global and local transformations are discussed in future sections.

1.3.4 Heuristic solutions to decomposition and factorization

A heuristic solution to the decomposition and factorization problem is given in Algorithm 1.

\[ \text{Algorithm 1} \ \text{Decomposition and Factorization} \]
\[
\text{Input} \ \text{Set of Boolean equations having a cost } c_1 \\
\text{Output} \ \text{Set of Boolean equations having a cost } c_2 \leq c_1 \\
\text{Method} \\
\text{begin} \\
\ \ \ \text{Generate candidate sub-expressions} \\
\ \ \ \text{repeat} \\
\ \ \ \ \ \text{Select the best sub-expression} \\
\ \ \ \ \ \text{Divide Boolean equations by the sub-expression} \\
\ \ \ \ \ \text{Update candidate sub-expressions} \\
\ \ \ \text{until} (\text{No sub-expressions of sufficient merit exists}) \\
\text{end}. \\
\]

The first step in the decomposition and factorization procedure is the generation of candidate sub-expressions to be used as possible divisors. A cost is also associated with each of the generated candidate sub-expressions which indicates the reduction in logic if a particular candidate is selected. Next, the selection step chooses the best sub-expression from those generated, and divides out of all functions in which it appears. After the division, the candidate sub-expressions generated in the first step may cease to be divisors of the resulting network. Also, the cost associated with the candidate sub-expressions may change. Hence it is necessary to update the sub-expressions generated in the first step.
The process of selecting and dividing continues until there are no candidates of sufficient merit to warrant division.

The area of the final logic produced and the processing time required by Algorithm 1 depends on the candidates that are generated during the first step and the time required to update the candidate sub-expressions. If the candidates are limited to only single-cube divisors, then Algorithm 1 corresponds to the technique proposed in [DieSu69]. This technique is also employed in [Breuer69, Sasa86]. A factoring algorithm for NOR logic design is dealt with in Section 1.3.5.

If the candidates are limited to cube-free multiple-cube divisors called kernels, then Algorithm 1 corresponds to the techniques proposed in [BraMcM82, BraMcM84]. These techniques are used as decomposition and factorization algorithms in various logic synthesis systems and are covered in detail in Section 1.3.6.

### 1.3.5 A factoring algorithm for NOR logic design

A factoring algorithm for NOR circuit synthesis is proposed in [DieSch65]. The method determines which variables can be grouped to form factors. The algorithm then goes through an evaluation routine to determine which factors to pick first. It solves the problem with these factors and ends up without requiring any additional factoring, or else with a subset of terms which the previous steps have to be applied to, in order to break the problem down into implementable factors. The algorithm is terminated when all factors meet the fan-in/fan-out criteria. The main disadvantage of this method is that, common factors which consist of more than one cube (multiple-cube divisors) are not considered.

### 1.3.6 Kernel-based decomposition and factorization algorithm

A bottom up technique for detecting common sub-expressions, based on kernels, is proposed in [BraMcM82]. Because of the importance of the technique to the implementation of synthesis systems like Yorktown silicon compiler [BrBrCD85], WDIVA/WDIVB [BarHac85], SOCRATES [GrBadH86], Multilevel logic interactive synthesis system MIS [BrDeKM86, BrRSW87a], CARLOS [Mathon85, MatBai88], DECAF [LiKeBr87], LOGOPT [Prabhu89] and SYNFUL [Ward90], algorithms based on kernels are covered in detail in this section.
The notion of kernels [BraMcM82] of a logic expression is introduced to provide a means of efficiently finding common sub-expressions from a set of expressions. Kernels are the bridges between algebraic expressions and factored forms. A kernel of an expression is defined by the following rules [Brayt87b]:

1. A kernel $K$ of an expression $f$ is the quotient of $f$ and a cube $C$. $K = f/C$

2. A kernel is cube-free, i.e. $K$ cannot be written as $K = dg$ where $d$ is a nontrivial cube and $g$ is an expression.

The cube $C$ used to obtain the kernel $K = f/C$ is called the co-kernel of $f$.

Since no single cube is cube-free, a kernel must have at least two cubes. The notation $K(f)$ refers to the set of kernels of $f$.

**Example 6.** Consider the expression

$$f = abde + abdg + eh$$

Note that $f = abd(e + g) + eh = abdg + e(abd + h)$. $f/abd = e + g$ with a remainder $eh$ and $f/e = abd + h$ with a remainder $abdg$. The kernels of $f$ are $e + g$ and $abd + h$, and the respective co-kernels are $abd$ and $e$.

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

**Theorem 1 [BraMcM82]:** Expressions $f$ and $g$ have common multiple-cube divisors if and only if there exist $k_f \in K(f)$ and $k_g = K(g)$ such that $|k_f \cap k_g| \geq 2$.

This theorem states that two functions have a common multiple-cube divisor if and only if the intersection of a kernel from $f$ and a kernel from $g$ has more than one cube.

**Example 7:** Let $f = ab + ac + bd + be$ and $g = mb + mc + pd + pr$. Then $K(f) = \{b + c, d + e\}$ and $K(g) = \{b + c, d + r\}$. $K(f) \cap K(g) = \{b + c\}$ and hence $(b + c)$ is the common multiple-cube divisor of $f$ and $g$.

Theorem 1 is used to determine if two or more expressions have any common
algebraic divisors other than single cubes. This can be done by computing the set of kernels for each logic expression and forming nontrivial (more than one term) intersections among kernels from different functions. If the intersection set is empty, then it is evident that common divisors between two or more expressions are only single-cubes and are easier to identify compared to multiple-cubes between expressions. Thus, one need not have to compute the set of all algebraic divisors for each expression to determine if there are common multiple-cube divisors. This leads to greater run time efficiency since the set of kernels is much smaller than the set of all algebraic divisors. Using the definitions of kernels and Theorem 1, the following decomposition algorithm is proposed in [BraMcM82].

<table>
<thead>
<tr>
<th><strong>Algorithm 2</strong></th>
<th>Decomposition algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>begin</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Distill</strong></td>
<td></td>
</tr>
<tr>
<td>1. Enumerate all kernels for each logic expression</td>
<td></td>
</tr>
<tr>
<td>2. Find a pair of kernels which intersect in two or more enumerated kernel sets</td>
<td></td>
</tr>
<tr>
<td>3. Substitute the new expression into the network</td>
<td></td>
</tr>
<tr>
<td>4. Repeat (1)-(4) while useful intersections are found in step (2)</td>
<td></td>
</tr>
<tr>
<td><strong>Condense</strong></td>
<td></td>
</tr>
<tr>
<td>1. Select two cubes which intersect in two or more literals</td>
<td></td>
</tr>
<tr>
<td>2. Substitute the new factor into the network</td>
<td></td>
</tr>
<tr>
<td>3. Repeat (1)-(3) while useful intersections are found in step (1)</td>
<td></td>
</tr>
<tr>
<td><strong>Collapse</strong></td>
<td></td>
</tr>
<tr>
<td>1. Back substitute the new expression if they are used only once</td>
<td></td>
</tr>
<tr>
<td><strong>end</strong></td>
<td></td>
</tr>
</tbody>
</table>

In the first phase, distillation, multiple-cube divisors are extracted from the network until no multiple-cube divisors remain. At this point, the only common divisors are single-cubes. In the second phase, condensation, single-cube divisors are extracted until no common divisors remain. At this point, the only common divisors in the networks are single literals. Some of the important issues in Algorithm 2 are discussed below.

**Number of kernels:** The number of kernels of an expression can grow exponentially in the number of literals in the support of the expression [BrRSW87a]. Recursively dividing each function by all literals produces a kernel tree as illustrated in Figure 1.7. It is only necessary to divide each function by those literals greater than or equal to the current...
literal as $f/xy$ is obtained by $(f/x)/y$ and does not need to be regenerated by $(f/y)/x$. It is evident from the Figure 1.7 that the maximum number of calls to a kernel generation routine is $2^s$, where $s$ is the size of the functions' support as it could be necessary to divide the function by all $2^s$ possible cubes.

**Separate kernel and single-cube extraction phase:** It is evident from Algorithm 2, that the method first extracts useful multiple-cube divisors from the functions. This terminates only when there are no more multiple-cube divisors for any pair of functions. Then, single-cube divisors are extracted until no further single-cube divisors exist. This distinct separation of kernel and single-cube extraction may have disadvantages, since the method may miss some of the best single-cube divisors.

**Separate complement extraction phase:** The method does not consider the complement of a kernel or of a single-cube divisor during the evaluation or the extraction of candidate sub-expressions. Hence, the method generates an expression and its complement expression as distinct common sub-expressions. In order to establish a complement relation between these two sub-expressions, the method calls for a simplification phase as a local optimization [BraMcM84, BrRSW87a]. The simplification routines are abstracted from the ESPRESSO PLA minimizer [BrHaMS84]. As a result, the complement relations between expressions are determined after kernel and single-cube extraction. This process is computationally expensive and it is profitable to find the complementary divisors while evaluating and extracting common sub-expressions.

**Figure 1.7** Kernel tree
Not entirely greedy method  The greedy method introduced in Algorithm 2 is to compute the set of kernels for each function of the network. Next the method identifies common kernels or common kernel intersections. There may be many common, intersected kernels. The computation to find the intersections of all kernels is not trivial, since the problem of detecting intersections in a set of kernels, and detecting common subcubes in a set of functions, are computationally equivalent to finding the kernels of an expression [BrRSW87b] With each of these intersected kernels a value is associated which indicates the number of literals that can be saved if the method has to extract this intersection. Among these, the one with the greatest value is selected and substituted. However, after one kernel is selected and substituted, it is possible that the values of the resulting kernels change. In fact, after one substitution, a kernel may cease to be a kernel of the resulting network. The recomputation of kernels after substitution is very expensive. Hence the method uses the set of kernels up to the point, where the values of kernels becomes inaccurate [BrRSW87a] This simplification does not guarantee that the best kernel is always selected and therefore the method is not entirely greedy.

1.3.7 Logic synthesis through local transformations

A technique for decomposition and factorization based on local transformations is proposed in [DaJoBT81, DaBrGJ84]. The emphasis of these techniques is to produce an acceptable, rather than optimal implementation in terms of design constraints such as area, delay etc in a short time.

<table>
<thead>
<tr>
<th>Refer Fig3[DaJoBT81]</th>
<th>Transformations</th>
<th>Name of the transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>nand1</td>
<td>( \overline{a} = a )</td>
<td>Involution</td>
</tr>
<tr>
<td>nand2</td>
<td>( ab\overline{c} = abc )</td>
<td>Multi nand to a single nand</td>
</tr>
<tr>
<td>nand3</td>
<td>( ab + ac = (b + c)a )</td>
<td>Factoring</td>
</tr>
<tr>
<td>nand4</td>
<td>( ab + \overline{b} = a + \overline{b} )</td>
<td>Absorption</td>
</tr>
<tr>
<td>nand5</td>
<td>( abc + ad = (bc + d)a )</td>
<td>Factoring</td>
</tr>
<tr>
<td>nand6</td>
<td>( d = abc, e = be \rightarrow d = ae, e = be )</td>
<td>Single-cube extraction</td>
</tr>
<tr>
<td>nand7a</td>
<td>( abc + ab = ab )</td>
<td>Single-cube containment</td>
</tr>
<tr>
<td>nand7b</td>
<td>( ab + \overline{ab} = a )</td>
<td>Consensus</td>
</tr>
<tr>
<td>nand7c</td>
<td>( a + \overline{ab} = a + b )</td>
<td>Absorption</td>
</tr>
</tbody>
</table>

Table 1.1 Local transformations
The local transformations used are shown in Figure 3 of [DaJoBT81]. and are described in detail in Table 11. Whenever the method recognizes one of the configurations in a circuit, it replaces that configuration with another. Hence each transformation encapsulates small pieces of design knowledge by describing how to change and potentially improve a design.

The quality of the final logic produced by this technique depends almost entirely on how well the method decides when and where to apply transformations to a design. Since the physical characteristics of components are available, actual circuit size and performance can be measured and thus improved. The primary disadvantage is the lack of global information when the decision is made to apply a particular transformation.

**1.3.8 Multi-level logic minimization using implicit don't cares**

A multi-level minimization algorithm [BaBrHJ88, BoHaJL87] that produces networks that are 100% testable, and generates test vectors as a by-product of the minimization, is reviewed in this section. The method extends the concept of two-level minimization to a multi-level context.

The connection between logic minimization and test generation rests on a simple observation that the absence of a test is associated with redundancy in the network. Removal of redundancies leads to a more cost-effective realization of a logic circuit, and it improves the testability as well.

In two-level logic, the sources of redundancy are well understood. A minimal two-level logic representation is both prime and irredundant. A sum-of-product logic representation is irredundant if and only if no terms can be deleted without changing the Boolean function. A product term of a sum-of-products representation is prime if and only if all literals of the term are necessary. These concepts are extended to multi-level context in [BaBrHJ88]. A Boolean network $\eta$ is said to be prime if all the cubes in each of the two-level representations $F_j$'s of $\eta$ are prime, and irredundant if all of these cubes are irredundant.

The don't care set for each node in a Boolean network arises from the following two phenomena 1) For a particular design the designer may decree that a primary input
vector, \( x \), will never occur. The vector \( x \) constitutes a don't care minterm and such minterms are don't care for all primary outputs. This set is called the fan-in don't care set of the node. 2) The set of node-input patterns, at which the node output does not contribute to any of the primary outputs of the circuit, is called the fan out don't care set of the node. The union of the two-sets is simply called the don't care set \( D_j \) of a node [BaBrHJ88].

In [BaBrHJ88] it is proved that a Boolean network is prime and irredundant if the following conditions are satisfied for every node:

1. If cube \( c \in F_j \), then \( c \not\in (F_j - \{c\}) \cup D_j \).

2. If a literal \( l \) (\( \bar{l} \)) appears in cube \( c \), then \( d \not\in F_j \cup D_j \), where \( d \) is the cube obtained by deleting \( l \) (\( \bar{l} \)) from cube \( c \).

The multi-level minimization procedure described in [BaBrHJ88] aims at satisfying both the conditions 1 and 2. The procedure is based on 1) Computing, for each intermediate node \( j \) in \( \eta \), a representation \( D_j \) of the don't care set. 2) Minimizing the representation of \( F_j \) with respect to \( D_j \) by calling the efficient two-level minimizer ESPRESSO [BrHaMS84].

The minimization of \( F_j \)'s are iterated until no change from what it had been in the previous pass. The network at this point is not only 100% testable, but the test vectors falls out as a by-product during the minimization of \( F_j \)'s.

The results of running the multi-level logic minimizer ESPRESSO-MLD is given in [BaBrHJ88] and ESPRESSO-MLT is given in [BoHaJL87]. The CPU time requirements may become exorbitant for large circuits since generation of all the complete don't care sets \( D_j \)'s can be very expensive [DeMaNS88].

1.3.9 Synthesis for testability

It was evident from Section 1.3.8 that the algorithm that produces prime, irredundant, and 100% testable circuits consumes an exorbitant amount of CPU time. Hence the current trend in logic synthesis has begun to focus on testability itself as an optimiza-
1.4 Research motivation and the approach

The motivation for this research has two key objectives. First, to develop a technique that generates an optimized network in terms of area. We should be able to apply this technique to any combinational circuit and optimize for area with reasonable computing resources. Second, if the network before optimization is 100% testable for single stuck-at faults by a complete test set $T$, the synthesized circuit must also be 100%...
testable and must be tested by the original test set $T$.

The approach presented in this thesis is to identify and extract useful double-cube divisors and single-cube divisors concurrently with their complements that together provide the greatest cost reduction in terms of the number of literals [VasRa90b]. In order to make the extraction process very simple, and to ensure that the operations are in the polynomial time domain, the method in every iteration looks at objects which are either multiple-cube divisors having exactly two cubes called double-cube divisors along with their complements, or single-cube divisors having exactly two literals. Using objects of size two, whose numbers are always in the polynomial domain, algebraic divisors of arbitrary size and their complements are obtained. The relationships between double-cube divisors, single-cube divisors, algebraic divisors and their complements are studied in this thesis. These properties are exploited in designing an efficient concurrent decomposition algorithm.

The discussion of testability preservation of algebraic transformations is presented in [HaJaKM89, RajVas89, VasRa90a, RajVas90]. It is shown in [HaJaKM89] that, for each multi-fault in an algebraically factored circuit, with each factor having only positive references in the circuit, there is an equivalent multi-fault in the original circuit. Hence it is shown that multi-fault testability is invariant and is maintained by algebraic factorization procedures. These results imply that, when algebraic factorization is applied to a minimized two-level circuit, all tests needed for complete multi-fault coverage of the synthesized circuit can be derived from the single-fault tests for the original circuit. In this work [VasRa90a, RajVas90], various transformations based on single-cubes and double-cubes along with their complements, are identified and it is shown that single-fault testability with respect to a test set is maintained by the decomposition and factorization procedures. This result implies that if an input Boolean network before decomposition and factorization is 100% testable for single stuck-at faults by a test set $T$, then the area optimized output network is also 100% testable for single stuck-at faults and can be tested by the same test $T$. Since the method assumes that the initial network is only single stuck-at fault testable, and because single stuck-at fault testability is maintained through the transformations, the method can be applied to a large class of irredundant two-level or multi-level circuits to synthesize fully testable circuits. These results are proved using the concepts of corresponding faults in the circuits and relations between complete test sets.
1.5 Organization of the thesis

Chapter 2 introduces the concept of two-literal single-cube divisors and double-cube divisors. Their properties are explored to demonstrate how they help to reason about common algebraic divisors and duality relations between expressions.

Chapter 3 discusses algorithms for generating cube divisors and for concurrent decomposition and factorization of multiple-output, multi-level circuits.

Chapter 4 provides experimental results that illustrate the effectiveness of the concurrent decomposition and factorization algorithm and compares the results with the well-known techniques.

Chapter 5 demonstrates that each transformation in the decomposition and factorization routine preserves testability, which implies that a complete test set developed for an input network also gives complete coverage of faults in the synthesized multi-level network. Experimental results of the testability-related properties are also provided. We conclude in Chapter 6.

Parts of this dissertation were presented at technical workshops and conferences, and is submitted for a journal. An algorithm for decomposition and factorization that uses two-literal single-cubes as decomposition objects during the initial phase and then uses double-cube divisors along with initial testability results, was presented at the Fourth Technical Workshop in New Directions for IC Testing [RajVas89]. Testability-related results for decomposition and factorization were presented at the 13th Annual IEEE Design for Testability Workshop [VasRa90a]. Formal proofs for each of the transformations in the concurrent decomposition and factorization procedure, using the concept of complete test sets, were presented at International Test Conference [RajVas90]. The concurrent decomposition and factorization procedure, along with the proofs for duality checking and reasoning about common algebraic divisors, were presented at International Conference on Computer-Aided Design [VasRa90b]. An article describing the concurrent decomposition and factorization procedure for multi-output networks, along with the testability-preserving results proved using the concepts of corresponding faults and relations between complete test sets, is submitted for publication in IEEE Transactions on Computer-Aided Design [RajVas91].
Chapter 2

Double cubes and two-literal single cubes in
decomposition and factorization

Decomposition and factorization translates a shallow multi-level network to a
deeper multi-level network by identifying common sub-expressions and rewriting the logic
in terms of those expressions. A heuristic solution to the problem is given in Section
1.3.4. Given a Boolean network, the method finds a candidate sub-expression and divides
the network by that sub-expression. Each of these candidate sub-expressions constitute
an object and division constitutes an operation in decomposition and factorization. The
selection of objects and the operation of division continues until there are no objects of
sufficient merit.

The basic problem in Algorithm 1 is to generate a rich set of possible objects
without generating an excessive number from among all possible sub-expressions. Using
these objects, it should be possible to reason about common algebraic divisors and duality
relations between expressions. A recursive top-down decomposition and factorization
algorithm is reviewed in Section 1.3.5 which employs only single-cubes as objects. The
disadvantage of this method is that common factors which consist of more than one cube
are not considered. Hence the method fails to reason about common algebraic multiple
cube divisors between expressions. A bottom-up decomposition and factorization technique
based on kernels is reviewed in Section 1.3.6. This technique uses both cube-free multiple-
cube divisors and single-cube divisors as possible objects. The problem with this method
is that it may generate an exponential number of objects.

Both of these techniques do not consider the complement relations that may
exists between objects. Hence, the methods generate an object and its complement as two
distinct objects. As the objects are of arbitrary size the duality relations may not be easy to establish.

In this chapter, we investigate objects that form a rich set of possible candidates. It will be shown that only two objects, two-literal single cubes, and double cubes along with their complement relations, are sufficient to reason about common algebraic divisors and duality relations between expressions. We will demonstrate that the number of these objects grow polynomially with the size of the input Boolean network. The duality relations between these objects are also investigated.

2.1 Single-cube and double-cube divisors

Given two Boolean expressions \( f \) and \( g \), \( g \) is called an algebraic divisor of \( f \) if \( f = gh + r \), where \( h \) and \( r \) are expressions and \( h \) is not null. Also \( sup(g) \cap sup(h) = \emptyset \). If \( g \) has exactly one cube, then \( g \) is called a single-cube divisor. It is evident that the total number of single-cube divisors having exactly two literals for a Boolean function with \( m \) literals is \( O(m^2) \). If \( g \) has more than one cube, then \( g \) is called a multiple-cube divisor. For example, if \( f = abc + abd + p \), \( g = ab \) is called a single-cube divisor and \( g = bc + bd \) is called a multiple-cube divisor. A Boolean expression \( f \) is cube-free if the only cube dividing \( f \) evenly is 1. Note that a cube-free expression must have more than one cube. For example, \( ab + c \) is cube-free but not \( ab + ac \) and \( abc \).

Double-cube divisors of a Boolean expression \( f \) are cube-free, multiple-cube divisors having exactly two cubes. For example, if \( f = ax + bx + cx \), then the double-cube divisors of \( f \) are \( \{a + b, a + c, b + c\} \). The set of all double-cube divisors of \( f \) is written as \( D(f) \), where

\[
D(f) = \{d \mid d = \{c_i \setminus (c_i \cap c_j), c_j \setminus (c_i \cap c_j)\}\}
\]

(1)

for \( i, j = 1 \ldots n, i \neq j \), where \( n \) is the number of cubes in \( f \).

It is evident that the total number of double-cube divisors for a Boolean function with \( n \) cubes is \( O(n^2) \). \((c_i \cap c_j)\) is called the base of cubes \( c_i \) and \( c_j \) and the double-cube divisor definition allows double-cube divisors with an empty base.

Example 8: Let \( f = ade + ag + bcde + bg \). The double-cube divisor and their bases are shown in Table 21.
2.2 Duality relations between objects

In this section, we study duality relations of double-cube divisors and single-cube divisors having exactly two literals. We demonstrate that these objects, despite their simplicity, provide a very good framework to reason about duality relations between expressions.

As the name indicates, double-cube divisors have exactly two cubes. A set of double-cube divisors is represented by $D$. A subset of double-cube divisors is represented by $D_{x,y,s}$, where $x$ is the number of literals in the first cube, $y$ is the number of literals in the second cube, and $s$ is the number of variables in the support of any double-cube divisor $d$. Note that $\max(x, y) \leq s \leq (x + y)$. Without loss of generality, we can assume $r = y$ for example, $xy + yzp \in D_{2,3,4}$.

Let $S$ denote the set of all single-cube divisors. A subset of single-cube divisors is denoted by $S_k$, where $k$ is the number of literals in the single-cube divisor. For example, $ab \in S_2$.

In the following, we discuss some useful relations between single-cube divisors, double-cube divisors, and their complements. We assume, in the discussion that follows, that double-cube divisors are extracted from functions which are prime and irredundant with respect to every output. The complements are obtained by only using DeMorgan's laws, while the Boolean reductions such as $a + a = a$, $a + \bar{a} = 1$, $a.a = a$, and $a + \bar{a} = 0$ are not employed.

Lemma 1: $D_{1,1,1}$ and $D_{1,2,2}$ are null sets.
Proof. Follows from the definition of double-cube divisors and the assumption that the Boolean network is prime and irredundant.

**Lemma 2:** For any \( d \in D_{1,1,2}, \overline{d} \in S_{2}. \)

**Proof.** For any \( d \in D_{1,1,2}, d \) has a support of 2. If the support is \{a, b\}, then \( d \) is either \( a + b \), or \( a + b \), or \( \overline{a} + b \), or \( \overline{a} + b \). By DeMorgan's laws, the complement of any \( d, d \in D_{1,1,2}. \) is a single-cube divisor having exactly two literals.

**Example 8:** The complement of \( a + b \) is \( \overline{a}b \in S_{2} \).

**Lemma 3:** For any \( d \in D_{1,2,3}, \overline{d} \notin D. \)

**Proof:** Since \( D_{1,2,3} \) has a support of 3, each variable in any double-cube divisor \( d \) must be different. Hence the complement of \( d, (d \in D_{1,2,3}), \) cannot be cube-free, and therefore it cannot be present in \( D \).

**Example 9:** \( a + \overline{b}c \in D_{1,2,3}. \) Its complement is \( \overline{a}b + \overline{a}c \). Since \( D \) is a set of cube-free divisors, \( \overline{a}b + \overline{a}c \) cannot be in \( D \).

**Lemma 4:** For any \( d \in D_{2,2,2}, d \) is either an exclusive-or or an exclusive-nor expression and \( \overline{d} \in D_{2,2,2} \).

**Proof:** Since each cube uses two variables and the support of the double-cube divisor is also two, the same variables are used in both cubes. Furthermore, since the double-cube divisor is cube-free, no literal can appear in both of the cubes. This implies that a variable must appear in the true form in one cube and in the complemented form in another cube. A two variable function having a two-cube, four-literal algebraic representation, is either exclusive-or or exclusive-nor and hence \( D_{2,2,2} = D_{\text{exor}} \cup D_{\text{exnor}} \) and \( D_{\text{exor}} \cap D_{\text{exnor}} = \phi \). For any \( d \in D_{\text{exor}}, \overline{d} \in D_{\text{exnor}}, \) and for any \( d \in D_{\text{exnor}}, \overline{d} \in D_{\text{exor}}, \) and since \( d \in D_{2,2,2}, D_{\text{exor}} \subseteq D_{2,2,2} \) and \( D_{\text{exnor}} \subseteq D_{2,2,2}, \overline{d} \in D_{2,2,2} \)

**Example 10:** \( \overline{a}b + a\overline{b} \in D_{2,2,2}. \) Its complement is \( a\overline{b} + \overline{a}b \in D_{2,2,2}. \)

**Lemma 5:** For any \( d \in D_{2,2,3}, \overline{d} \in D_{2,2,3}. \)
2.2 Duality relations between objects

Proof: Since each cube uses two variables and the support of the double cube divisor is three, one of the three variables must be present in both cubes. Since the double-cube divisors are cube-free, the common variable must be present in true form in one cube and in the complemented form in the other cube of the double cube divisor. By De Morgan's and absorption laws of Boolean algebra, \( \overline{ab + bc} = \overline{ab} + \overline{bc} \). Hence, the complements of elements in \( D_{2,2,3} \) are also present in the same set. Q.E.D.

Example 11: \( \overline{ab + ac} \in D_{2,2,3} \). Its complement is \( \overline{ab + ac} = \overline{ab} + \overline{ac} \in D_{2,2,3} \).

Lemma 6: For any \( d \in D_{2,2,4} \), \( \overline{d} \notin D \).

Proof: Similar to Lemma 3. Q.E.D.

Given two Boolean expressions \( f \) and \( g \), the important task during decomposition is to establish whether a function \( f \) has a complement cube divisor in \( g \). Theorem 2 establishes such relations between two expressions.

Theorem 2: Let \( f \) and \( g \) be two Boolean expressions. If

\[
\begin{align*}
&d_i \neq s_j \quad \text{for every } d_i \in D_{1,1,2}(f), s_j \in S_2(g), \text{ and} \\
&d_i \neq \overline{s}_j \quad \text{for every } d_i \in D_{ezor}(f), d_j \in D_{ezor}(g), \text{ and} \\
&d_i \neq \overline{d}_j \quad \text{for every } d_i \in D_{ezor}(f), d_j \in D_{ezor}(g), \text{ and} \\
&d_i \neq \overline{d}_j \quad \text{for every } d_i \in D_{2,2,3}(f), d_j \in D_{2,2,3}(g), \text{ and} \\
&d_i \neq s_j \quad \text{for every } d_i \in D_{1,1,2}(g), s_j \in S_2(f),
\end{align*}
\]

then \( f \) has neither a complement double-cube divisor nor a complement single cube divisor in \( g \).

Proof: If duality exists, as indicated by arrows in Figure 2.1, then \( f \) has a complement cube divisor in \( g \). This can be verified by using Lemma 1 to Lemma 6. For any other double-cube divisor \( d, d \in D(f) - \{ D_{1,1,2}, D_{2,2,2}, D_{2,2,3} \} \), \( d \) cannot have complement.
2.3 Common algebraic divisors between expressions

This theorem has a very important practical application in checking duality relations between Boolean expressions. Not only is the problem of duality checking reduced to a subset of double-cube divisors and single-cube divisors, but for each such subset only a subclass of objects, as shown in Figure 2.1, has to be checked.

2.3 Common algebraic divisors between expressions

In this section we answer the following important question pertaining to the extraction of common multiple-cube algebraic expressions from a multi-output network. Given expressions \( f \) and \( g \), if \( f \) and \( g \) have a common multiple-cube divisor, then is it possible to locate that divisor by examining the double-cube divisors of \( f \) and \( g \)? Theorem 3 demonstrates that reasoning about the presence of common multiple-cube divisors can be performed based on the set of double-cube divisors.
Theorem 3: Expressions $f$ and $g$ have a common multiple-cube divisor if and only if $D(f) \cap D(g) \neq 0$.

Proof: If $D(f) \cap D(g) \neq 0$, then there exists a $d \in D(f) \cap D(g)$ which is a double-cube divisor that divides both $f$ and $g$.

Only if. Now suppose that $f$ and $g$ have a common multiple-cube divisor $e$. $C|f, C|g$. ( $C|f$ denotes $C$ divides $f$.) Let $C = \{c_1, c_2, \ldots, c_m\}$. Take any $e = \{e_1, e_j\}$ such that $c_i, c_j \in C$. If $e$ is cube-free, then $e \in D(f) \cap D(g)$. If $e$ is not cube-free, then $e' = \{c_i \setminus (c_i \cap c_j), c_j \setminus (c_i \cap c_j)\}$ exists since $f$ and $g$ are non-redundant. Hence $e' \in D(f) \cap D(g)$.

Q.E.D.

This theorem states that two functions $f$ and $g$ have a common multiple-cube divisor if and only if an intersection of a double-cube divisor from $f$ and a double-cube divisor from $g$ is not empty. Thus, it provides a method for detecting whether two or more expressions have any common multiple-cube divisors other than only single cubes. The method is to: 1) compute double-cube divisors for each $f$ using equation 1, and 2) finding the intersection of all double-cube divisors. If the intersection set of all double-cube divisors is empty then it is guaranteed that the functions have no common multiple-cube divisors. This is a very important result, since one does not have to compute the set of all algebraic divisors for each expression to determine if there are common multiple-cube divisors. Since the number of double-cube divisors grows polynomially with the number of cubes of a function, this set must be much smaller than the set of all algebraic divisors. Hence Theorem 3 gives a very strong technique to reason about common algebraic divisors between various expressions.
Chapter 3 Concurrent decomposition and factorization algorithm

All heuristic solutions to the decomposition and factorization of Boolean expressions essentially have two basic operations. They are: generation of useful objects, and substitution of best objects in the expressions. Chapter 2 demonstrates that two-literal single cubes and double cubes constitute important objects of decomposition and factorization. The number of these objects grow polynomially with the size of a Boolean network and can reason about common algebraic divisors and duality relations between expressions. In this chapter, algorithms which can generate these objects efficiently are presented. A method for decomposition and factorization of Boolean expressions using these objects is presented in Algorithm 3.

---

**Algorithm 3 Concurrent Decomposition and Factorization**

**Input**: Set of Boolean equations as a sparse matrix $B$

**Output**: Set of multi-level equations

**begin**

1. Generate double-cube divisors with weights using Algorithm 4
2. **repeat**
   1. Select a double-cube divisor $d$ that has a maximum weight, $W_{d_{max}}$
   2. Check for a single-cube divisor $s$ having weight $W_{s_{max}}$ using Algorithm 5
   3. If $W_{d_{max}} > W_{s_{max}}$
      - then Select double-cube divisor $d$
      - else Select single-cube divisor $s$
   4. $W = \max(W_{d_{max}}, W_{s_{max}})$
   5. Substitute the new expression in $B$ using Algorithms 6, 7 and 8
   6. Recompute weights of affected double-cube divisors
3. **until** $(W < 0)$

**end**

---

In this algorithm, the multi-level logic is represented by a Boolean matrix $B$, where there is one row for each cube in the disjunctive form and one column for each different literal. The primary output or intermediate output is associated with each of the
3.1 Generation of double-cube divisors

The generation of all double-cube divisors is done, using equation (1), only once during the synthesis. For every pair of cubes connected to the same output, we generate a double-cube divisor \( d, d \in D_{x,y,s} \). If neither the double-cube divisor \( d \), nor its complement double-cube divisor exists in set \( D_{x,y,s} \), the new \( d \) is added to the set \( D_{r,v,s} \).

With each of the generated double-cube divisors, \( d \), a weight function \( W(d) \) is associated which indicates the number of literals that can be reduced, if this double cube divisor is selected. This information is used to select the most useful of all candidates generated. In order to compute the weight of a double-cube divisor, the following information is stored with each of the divisors. 1) the number of times the double-cube divisor is used, \( p \) and 2) the number of literals in the base of each double-cube divisor calculation, \( \| c_i \cap c_j \| \). Each of these bases are denoted by \( b_i \), for \( 1 \leq i \leq p \).

The complement of the double-cube divisor is also considered in the weight computation. For any double-cube divisor \( d, d \in D_{2,2,2} \) or \( d \in D_{2,2,3} \), its complement
double-cube divisor is also in \( D_{2,2,2} \) or \( D_{2,2,3} \) respectively, and hence the same information as explained above is stored. In the case of \( d \in D_{1,1,2} \), the complement is a single-cube divisor, \( s = ab \in S_2 \), where \( a \) and \( b \) are two distinct literals of a double-cube divisor. The number of cubes, \( C' \), which contain both \( a \) and \( b \), is stored with the corresponding double-cube divisor. This indicates the number of literals that can be reduced by using the complement of the double-cube divisor.

The weight of a double-cube divisor \( d, d \in D_{x,y,z} \), is given by the formula

\[
W(d) = (p - 1)(x + y) - p + \sum_{i=1}^{p} |b_i| + C 
\]  

(\( p - 1)(x + y) \) accounts for the number of literals that can be saved by implementing the double-cube divisor of size \( (x + y) \) once. The argument \(-p\) in the formula accounts for the number of literals needed to connect the new double-cube divisor in the \( p \) occurrences of the cube-divisor.

A method for generating double-cube divisors is given in Algorithm 4

```
Algorithm 4 Generation of double-cube divisors
Input A Boolean function \( f \) with \( n \) cubes \( c_1, c_2, \ldots, c_n \)
Output Set of double-cube divisors with weights
begin
  for \( i = 1 \) to \( n - 1 \) do
    for \( j = (i+1) \) to \( n \) do
      begin
        \( d = c_i \setminus (c_i \cap c_j), c_j \setminus (c_i \cap c_j) \)
        if \( d \) or \( \{d \in D_{2,2,2} \) or \( D_{2,2,3} \) \) is not present then
          begin
            Store \( d \) in appropriate \( D_{x,y,z} \)
            Compute implementation cost
            if \( d \in D_{1,1,2} \), then compute \( C' \)
          end
        Compute base and update weight using equation (2)
      end
end
```

Algorithm 4 generates double-cube divisors from a function \( f \), irrespective of the number of literals in each of its cubes. It is possible to generate only a subset of double-cube divisors by limiting the number of literals in them. As a consequence, it is possible to reduce the number of double-cube divisors to be handled by the decomposition and factorization algorithm. A series of experiments were performed to limit the number of
literals in each of the double-cube divisors. The quality of the final logic produced and the CPU time required for the decomposition and factorization are reported in detail in Section 4.7.

3.2 Generation of single-cube divisors

Single-cube divisor extraction is the process of finding cubes which are common to two or more cubes, and extracting the common cube to simplify Boolean expressions. Double-cube divisor extraction is done separately for each function, but single-cube divisors can be extracted from two or more cubes even though they may belong to different functions.

Double-cube divisors are extracted once using Algorithm 4, and updated as the synthesis proceeds. So at any step during the synthesis, the decomposition procedure has the best available double-cube divisor. The maximum number of literals that can be saved by using the best available double-cube divisor is denoted by $W_{dmax}$.

---

**Algorithm 5** Generation of a single-cube divisor

**Inputs** Matrix $B$ of size $\{n, p\}$, $n$ is the number of columns and $p$ is the number of cubes

**Output** Weight of the "best" available double-cube divisor, $W_{dmax}$

**begin**

$k = 0 \ , \ L = \{\}$

repeat

step 1 For each column compute height of the column

step 2 $k =$ maximum number of literals found in any column and not considered so far

if $(k - 2) \leq W_{dmax}$ then

No single-cube divisor having weight more than $W_{dmax}$ exists Exit

step 3 $T =$ {all columns having $k$ literals }

step 4 For all possible pairs of $T$ compute coincidence and store in maxheap

step 5 For all possible pairs of $T$ and $L$ repeat step 4

step 6 Threshold = (maxheap)

step 7 If Threshold $\geq k$ then

begin

If $(\text{Threshold} - 2) > W_{dmax}$ then

Top of the heap gives single-cube divisor of weight $W_{dmax}$ Exit

else

No single-cube divisor having weight more than $W_{dmax}$ exists Exit

end

$L \ := \ L \cup \{T\}$

until (Threshold $\leq 2$).

No single-cube divisor exists Exit

end.
Substitution of a cube divisor

The objective of the single-cube divisor generation procedure is to find a single-cube divisor that has a maximum weight, $W_{\text{smax}}$. If this weight is greater than $W_{\text{dmmax}}$, then the decomposition routine selects a single-cube divisor rather than the current double-cube divisor. To facilitate the computation of required single-cube divisors we introduce the following definitions.

*Height of a column*(t) is the number of elements in column(t). The *coincidence* $C'(t,j)$ of a single-cube divisor $s \in S_2 = tj$, where $t$ and $j$ are columns of $B$, is defined as the number of cubes which contain both $t$ and $j$. Note that $C'(t,j) = C'(j,i)$ and $C'(t,j) \leq \min(\text{height}(t),\text{height}(j))$. Weight of a single-cube divisor $s \in S_2 = tj$ is defined as $W(t,j) = C(t,j) - 2$.

$W(t,j)$ denotes the number of literals that can be saved in the network if $s$ is chosen as a single-cube divisor. The "-2" in the formula accounts for the implementation of single-cube divisors. The greedy, iterative method for single-cube divisor generation is given in Algorithm 5. During each iteration of the algorithm, the method looks at the height of each column, not considered so far, and selects column(s) that has(have) a maximum count of $k$ literals (step 1.2). If the value of $(k - 2)$ is less than or equal to $W_{\text{dmmax}}$, then the algorithm terminates as it is beneficial to extract the current double-cube divisor having a weight of $W_{\text{dmmax}}$. For all possible combinations (worst case $O(n^2)$, where $n$ is the total number of columns in matrix $B$) coincidences are computed and stored in a maxheap data structure (step 3.4 and 5). The coincidence of columns having the highest count computed so far is defined as *threshold*, $T$, and the top of the maxheap gives this information (step 6). If the threshold is greater than or equal to $k$, it is certain that no two columns having coincidences better than current $k$ are available. These two columns then correspond to a single-cube divisor, $s$, that provides the maximum literal saving. The weight of $s$ is $W_{\text{smax}} = T - 2$. If the current threshold is less than the current column count $k$, then the algorithm iterates until it finds two columns which have coincidence greater than or equal to $k$, or the threshold reduces to 2.

The problem of finding coincidence between two columns is similar to finding the intersection of two sets in a sorted list. This can be done in linear time. Also, as the coincidences are stored in the maxheap data structure, the top of the heap always gives the best available single-cube divisor.
3.3 Substitution of a cube divisor

In any iteration, Algorithm 3 finds a two-literal single-cube divisor or a double cube divisor along with its complement as a common sub expression. The Boolean network is then expressed in terms of this new object. More precisely, a new variable \( r \) that represents the selected object \( g \) is added to the network. Then each expression \( f \) in the Boolean network \( B \) is represented by a new expression \( f \) as

\[
f = v\left(\frac{f}{g}\right) + \bar{v}\left(\frac{f}{\bar{g}}\right) + r
\]

where \( r \) is the remainder resulting from the division. This process is referred to as algebraic substitution of \( v \) for \( g \) in \( f \) [BraMcM84, BaCodH86]

**Example 12** Consider a Boolean network having the following 3 equations

\[
\begin{align*}
f &= ab + ac \\
g &= b + c + d \\
h &= \bar{b}e
\end{align*}
\]

Let \( v = b + c \) be the selected sub-expression. After algebraic substitution, the resulting equations are as follows:

\[
\begin{align*}
f &= av \\
g &= d + v \\
h &= \bar{v}e
\end{align*}
\]

A method for substituting a single-cube divisor in a Boolean network is given in Algorithm 6.

---

**Algorithm 6** \( S(B, s) \)

**Inputs** Boolean equations in sparse matrix \( B \),

A single-cube divisor \( s \)

**Output** Boolean equations expressed in terms of \( s \)

**begin**

Add a new column \( v \) to matrix \( B \)

for each cube \( C \) in \( B \) do

begin

if \( s \subseteq C \) then \( C' = (C - s) \cup v \)

end

end
The method rewrites each cube that has a single-cube divisor \( s \) in terms of the added variable \( v \). A method for substituting a double-cube divisor (that has no complement) in a Boolean network is given in Algorithm 7.

\textit{Algorithm 7:} \( D(B, d) \)

\begin{itemize}
  \item \textbf{Inputs:} Boolean equations in sparse matrix \( B \)
  \item \textbf{A double-cube divisor} \( d \)
  \item \textbf{Output:} Boolean equations expressed in terms of \( d \)
\end{itemize}

\begin{itemize}
  \item \textbf{begin}
  \item Add a new column \( v \) to matrix \( B \)
  \item for each pair of cubes \( c_i \) and \( c_j \) used to compute \( d \) do
    \item \textbf{begin}
    \item Delete cubes \( c_i \) and \( c_j \)
    \item Add a new cube \( c_k = (c_i \cap c_j) \cup v \) in \( B \)
    \item \textbf{end}
  \item \textbf{end}
\end{itemize}

In this case, the Boolean network is rewritten in terms of the new double-cube divisor \( d \). A method for substituting a double-cube divisor along with its complement is given in Algorithm 8.

\textit{Algorithm 8:} \( DC(B, d) \)

\begin{itemize}
  \item \textbf{Inputs:} Boolean equations in sparse matrix \( B \)
  \item \textbf{A double-cube divisor} \( d \)
  \item \textbf{Output:} Boolean equations expressed in terms of \( d \)
\end{itemize}

\begin{itemize}
  \item \textbf{begin}
  \item \( D(B, d) \)
  \item if \( (d \in D_{2,2,2} \lor D_{2,2,3}) \land (\bar{d} \text{ exists}) \) then \( D(B, \bar{d}) \)
  \item else
    \item if \( (d \in D_{1,1,2}) \land (\bar{d} \text{ exists}) \) then \( S(B, \bar{d}) \)
  \item \textbf{end}
\end{itemize}

### 3.4 Recomputation of double-cube divisors

When a double-cube divisor or a single-cube divisor is selected and substituted, the weights of the remaining cube divisors may change and therefore they have to be recomputed. In fact, after one substitution, a double-cube divisor may cease to be a divisor of the resulting network. In order to recompute: (1) with each of the double-cube divisors in set \( D_{x,y,z} \), the cubes that are used to compute them are stored, and (2) with each cube of matrix \( B \), the double-cube divisors that are computed using them are stored. With the selection of a double-cube divisor, cubes that are affected can be known using information (1) From these cubes, using information (2) above, it is possible to know which double-
cube divisors are affected. Only the weights of these double-cube divisors are recomputed and updated.

Notice that the process of division by a single-cube divisor may affect double-cube divisors which are either in set $D_{2,2,2}$ or in $D_{2,2,3}$. Whenever such a substitution is performed the corresponding double-cube divisors preserve a copy of their original form, in addition to the new one, to enable the determination of the duality relation as stated in Theorem 2.

### 3.5 Decomposition and factorization of multi-output circuits

The concurrent decomposition and factorization procedure presented in Algorithm 3 can be applied to a multi-output circuit. The normal operation is to replace a multi-output network with a single-output network for each of its outputs. This makes each output of the multi-level network a separate single-output function. Note that the circuit obtained by this operation, for a two-level circuit, is different from the circuit obtained from the single-output option of a two-level minimizer like ESPRESSO [BrHaMS84], although their functions are identical.

The disadvantage of this approach is that the number of product terms to be handled may increase with the number of outputs. This is especially true if a product term is used in more than one output. For a circuit having $n$ inputs, $p$ product terms and $m$ outputs, the number of product terms to be handled may increase from $p$ to $p + m$. This implies that the number of double-cube divisors to be extracted increases from $p^2$ to $mp^2$. For a circuit having a large value of $m$, there may be considerable increases in the space to store double-cube divisors and also in the CPU time to process and update the cube-divisors.

In this section, an algorithm is presented to extract common sub-expressions from the outputs of a multi-output network which reduces considerably the number of product terms to be processed. The advantage of this approach is that the possible sharing of common sub-expressions is identified from the primary outputs before applying the concurrent decomposition procedure. Let $C$ be a multi-output network having $n$ primary inputs, $k$ cubes and $m$ outputs, $f_1, f_2, \ldots, f_m$. The method extract common sub-expressions from the outputs of the network by the following procedures:
3.5 Decomposition and factorization of multi-output circuits

Construction of cube/output matrix \( CO \): The method constructs a cube/output matrix \( CO \) of size \([k + m] \) with \( CO_{ij} = 1 \), \( 1 \leq i \leq k \), \( 1 \leq j \leq m \), if the product term \( i \) is connected to the output \( j \). This matrix consolidate cubes that are connected to the same outputs.

**Algorithm 9** Decomposition and factorization of multi-output network

**Input**: A Boolean network having \( n \) primary inputs, \( k \) cubes \( C_1, \ldots, C_k \), and \( m \) outputs \( f_1, \ldots, f_m \)

**Output**: An optimized Boolean network

begin
 /* Step 1 */ Construct cube/output matrix \( CO \) */
 for \( i = 1 \) to \( k \) do
   for \( j = 1 \) to \( m \) do
     if cube \( C_i \) is connected to output \( f_j \) then \( CO_{ij} = 1 \)
 /* Step 2 */ Extract single-cube divisors from matrix \( CO \) */
 \( r = 0 \)
 repeat
   increment \( i \).
   Extract two-literal single-cube divisor, \( f_{m+i} \) from \( CO \)
 until (coincidence < 2)
 \( r = i \)
 /* Step 3 */ Construct an output/single-cube matrix \( OS \) */
 for \( i = 1 \) to \( m \) do
   for \( j = 1 \) to \( r \) do
     if output \( f_i \) is used in single-cube divisor \( f_{m+j} \) then \( OS_{ij} = 1 \)
 /* Step 4 */ Extract single-cube divisors from matrix \( OS \) */
 \( s = 0 \)
 repeat
   increment \( i \).
   Extract two-literal single-cube divisor, \( f_{m+r+i} \) from \( OS \)
 until (coincidence < 2)
 /* Step 5 */ Call decomposition and factorization algorithm */
 Apply Algorithm 3 on \( k \) cubes and the matrix \( CO \)
end

**Extraction of single-cube divisors from matrix \( CO \)**: In Step 2, single-cube divisors are extracted from the matrix \( CO \) using Algorithm 5. The method is greedy and at any iteration extracts a two-literal single cube divisor, \( f = uv \), having maximum coincidence. These two columns are merged into a new column, as explained in Algorithm 6: the literals are removed from columns \( u \) and \( v \) and a new column is added to the matrix \( CO \). In the next iterations, the method treats primary columns and added intermediate columns in exactly the same way, and the extraction terminates when there are no single-cubes having
a coincidence more than 1. Let \( f_{m+1}, \ldots, f_{m+r} \) be \( r \) single-cube divisors of size 2 extracted from the matrix \( CO \). The size of the matrix \( CO \) at the end of step 2 is \([k + \lfloor m + r \rfloor]\), since \( r \) single-cube divisors are extracted. Each of the single-cube divisors, \( f = uv \), indicates that the outputs \( u \) and \( v \) use \( f \).

**Construction of output/single-cube matrix \( OS \)** An output/single-cube matrix \( OS \) of size \([m + r]\) is constructed, with \( OS_{ij} = 1 \), \( 1 \leq i \leq m \) \( 1 \leq j \leq r \), if an output \( f_i \) is used in the extracted single-cube \( f_j \).

**Sharing of possible output expressions**: Single-cube divisors are extracted from the matrix \( OS \) as in step 2. Each of the extracted single-cube divisors gives the possible sharing of expressions from \( f_{m+1}, \ldots, f_{m+r} \). The concurrent decomposition procedure is then applied on the \( k \) cubes of \( C \) and the matrix \( CO \).

**Example 13** As an example illustrating the foregoing procedure, consider the circuit with [BrHaMS84], which realizes the following multi-output circuit. The circuit has four inputs, \( 1 \) \( 4 \), and seven outputs, \( f_1, \ldots, f_7 \).

\[
\begin{align*}
  f_1 &= 234 + 234 + 234 + 1 + 234 + 34 + 23 \\
  f_2 &= 234 + 234 + 234 + 1 + 234 \\
  f_3 &= 234 + 34 + 23 + 23 \\
  f_4 &= 234 + 24 \\
  f_5 &= 234 + 234 + 234 + 1 + 34 + 23 \\
  f_6 &= 234 + 234 + 234 + 34 + 23 \\
  f_7 &= 234 + 234 + 1 + 234 + 23 
\end{align*}
\]

The matrix \( CO \) is shown in A. The resulting matrix after step 2 is shown in B. The single-cube divisors extracted are shown in C.
Step 2 has extracted 5 single-cube divisors, $f_8, \ldots, f_{12}$. It is evident from the extracted single-cube divisors that:

- $f_8$ is used by $f_1$ and $f_5$.
- $f_9$ is used by $f_2$ and $f_7$.
- Since column 10 is empty, $f_{10}$ cannot be generated. Hence the occurrence of $f_{10}$ in $f_{11}$ and $f_{12}$ should be replaced by $f_6$ and $f_8$.
- $f_{11}$ is used by $f_1$, $f_2$, $f_5$, $f_6$, and $f_7$.
- $f_{12}$ is used by $f_1$, $f_3$, $f_5$, and $f_6$.

Using the above information an OS matrix, as shown in D, is constructed. Row 1 in D indicates that the primary output $f_1$ is used in $f_8$, $f_{11}$ and $f_{12}$. The matrix D after step 4 is shown in E and the extracted single-cube divisors are shown in F:

The extracted single-cube divisors indicate that $f_{13} = f_{12} + f_{11}$, $f_{14} = f_9 + f_{11}$ and $f_{15} = f_{13} + f_8$.

According to Step 5, the concurrent decomposition procedure is applied on the $k$ product terms and matrix B, as follows:

The resultant multi-level equations are as follows:

PRIMARY OUTPUTS

$f_1 = 25 + f_{15}$
$f_2 = 25 + f_{14}$
$f_3 = 23 + 6 + f_{12}$
$f_4 = 47$
3.5 Decomposition and factorization of multi-output circuits

\[

e_5 = e_{15} \\
e_6 = e_3 + e_{13} \\
e_7 = e_6 + e_{14} \\
\text{Intermediates} \\
\begin{align*}
e_{11} &= 28 \\
e_{12} &= 34 + 25 \\
e_{13} &= e_{12} + e_{11} \\
e_{14} &= 25 + 1 + e_{11} \\
e_{15} &= e_{13} + 6 + 1 \\
e_7 &= 2 + 3 \\
e_8 &= 34 + 34 \\
e_5 &= 34 \\
e_6 &= 23 \\
\end{align*}
\]

TOTAL LITERAL in SOP 43
Decomposition and factorization of Boolean expressions is a known difficult problem. Hence the research is focused on developing heuristic solutions to this problem. Various heuristics that are employed for both the generation and substitution of objects in Algorithm 1 were presented in Chapters 1 and 2. For a certain class of input problems, the objects that are employed for decomposition and factorization may favor and produce excellent results, both in terms of the final area of the logic and the processing time required to synthesize them. The same objects may produce inferior quality results in other circuits. The only way of understanding any heuristic algorithm is to evaluate it on a large class of circuits and to compare it with various other approaches.

In order to compare various approaches of decomposition and factorization, a set of benchmark circuits are provided in Book PLAs [BrHaMS84], Design Automation Conference [deGeus86], and International Workshop on Logic Synthesis [Lisank89]. The main objective of this chapter is to evaluate the concurrent decomposition and factorization procedure presented in Algorithm 3 with other well-known techniques like YORKTOWN silicon compiler [BrBrCD85], WDIVA [BarHuc85], SOCRATES [GrBadH86], Multilevel logic interactive synthesis system-MIS [BrDeKM86, BrRSW87a], DECAF [LiKeBr87], and BOLD [BoHaJL87]. Algorithm 3 is implemented in a C program called Pendulum. The criteria employed for the evaluation are based on the guidelines described in [deGeus86] and are as follows: 1) The quality of the final logic is measured in terms of literal counts, the number of gates normalized to two-input equivalent gates and, the total gates required after technology mapping using a particular cell library. 2) The cost of synthesis is based on the CPU time required.
In our experiments, and in our comparisons of the results with other synthesis systems, three measures of circuit quality are used. The first measure is the number of literals and is defined as follows:

The literals in sum-of-products form cost function for a Boolean network is the sum over all nodes of the number of literals in the sum-of-products representation for the function. The literals in factored form cost function for a Boolean network is the number of literals in an optimal factored form for each expression in the network. The number of literals is an important measure since literals translate roughly into transistors or transistor pairs, therefore similar reductions in chip size may be expected [BoHaJL87].

The second measure is the number of gates normalized in terms of two input equivalent gates as suggested by de Geus in [deGeus86].

The third measure is the total chip area after technology mapping to a particular cell library.

In the results provided in this chapter we have run the program MIS2 1 from University of California, Berkeley, to compile the data provided in Tables 4.2, 4.3, and 4.9. As the final logic produced by MIS2 1 is dependent upon the script (set of operations) used [BrDeKM86], we provide the script used to run MIS2 1. Since we do not have access to other logic synthesis systems (and in many cases the script that was used to run MIS2 1 by authors are not reported), the results are taken directly from the literature and are compared with the concurrent decomposition and factorization algorithm.

4.1 Comparison with weak division approach

Algebraic decomposition or weak division is a method of recognizing common sub-expressions which are common to two or more different functions and factoring individual functions. A method that implements this technique was reported in [BarHac85]. The method is iterative and has the following phases: the generation of candidate sub-expressions using the techniques presented in [BraMcM82], selection of the best sub-expressions, and the substitution of these sub-expressions into the functions which they divide. The complement of the candidate sub-expressions is also considered for substitution. Table 4.1 shows the results of running weak division, WDIVA, on several Book PLAs.
Comparison with MIS2.1

Table 4.2 summarizes the results obtained using Pendulum and compares to those generated with MIS2.1 (both run on the same SUN 3/260) using the script given in Figure 4.1.
Note that only "gkx" and "gcx" commands are used to run MIS21. These scripts ensure that only level-0 kernels [BrRSW87a] are used and the more efficient "ping-pong" algorithm [Brayt87a] is used to find a good (but not necessarily the best) kernel intersection and single-cube divisor. These options, in MIS21, take less CPU time compared to the standard script which finds all the kernels and then chooses the best kernel. The effectiveness of the method can be summarized by the total literal count and the total time taken to synthesize the circuits. The numbers are 8.871 literals and 6.076 CPU seconds for Pendulum and 11.488 literals and 53.648 CPU seconds for MIS21. For these benchmarks, Pendulum synthesized circuits with, on the average, 20 percent fewer literals and using one ninth of the CPU time required by MIS21. The results for MIS21 were reported in the MCNC 1989 Workshop poster session [Lisank89] and are given in the last two columns of Table 4.2. *Single* is the *factored form* literal count using a single-execution of a standard script. *Best* is the best factored form literal count using several executions of a standard script. It can be seen that for large benchmark circuits like *apex1, apex3, apex4, apex5* and *seq*, Pendulum produces smaller networks than that of the *best* results obtained by MIS and for *9sym, duke2, e64* and *rd84*, Pendulum produces smaller networks than that of the single execution of MIS21.

The effectiveness of our approach is due to the fact that Pendulum work with single-cube divisors, double-cube divisors and their complements which grows polynomially with the size of a network. The duality relation between various intermediate nodes is also
### Table 4.2 Performance comparison between Pendulum and MIS21

<table>
<thead>
<tr>
<th>pla</th>
<th>Final literals in SOP</th>
<th>CPU seconds</th>
<th>Literal count in factored-form</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pendulum</td>
<td>MIS21*</td>
<td>Pendulum</td>
</tr>
<tr>
<td>5xp1</td>
<td>163</td>
<td>186</td>
<td>1.7</td>
</tr>
<tr>
<td>9sym</td>
<td>263</td>
<td>255</td>
<td>95.5</td>
</tr>
<tr>
<td>add6</td>
<td>85</td>
<td>104</td>
<td>220</td>
</tr>
<tr>
<td>apex1</td>
<td>1101</td>
<td>1268</td>
<td>75</td>
</tr>
<tr>
<td>apex2</td>
<td>452</td>
<td>540</td>
<td>4498</td>
</tr>
<tr>
<td>apex3</td>
<td>1381</td>
<td>1876</td>
<td>184.5</td>
</tr>
<tr>
<td>apex4</td>
<td>1976</td>
<td>2594</td>
<td>346.9</td>
</tr>
<tr>
<td>apex5</td>
<td>893</td>
<td>1112</td>
<td>183</td>
</tr>
<tr>
<td>chkn</td>
<td>469</td>
<td>452</td>
<td>184.2</td>
</tr>
<tr>
<td>duke2</td>
<td>435</td>
<td>493</td>
<td>13.0</td>
</tr>
<tr>
<td>e64</td>
<td>253</td>
<td>253</td>
<td>2.7</td>
</tr>
<tr>
<td>rd73</td>
<td>106</td>
<td>117</td>
<td>11.8</td>
</tr>
<tr>
<td>rd84</td>
<td>143</td>
<td>446</td>
<td>136.6</td>
</tr>
<tr>
<td>sao2</td>
<td>171</td>
<td>213</td>
<td>4.9</td>
</tr>
<tr>
<td>seq</td>
<td>948</td>
<td>1547</td>
<td>63.8</td>
</tr>
<tr>
<td>xor9</td>
<td>32</td>
<td>32</td>
<td>55</td>
</tr>
<tr>
<td>TOTAL</td>
<td>8,871</td>
<td>11,488</td>
<td>6,076.6</td>
</tr>
</tbody>
</table>

* Results obtained by the author using the script given in Figure 4.1

** Results reported in MCNC'89 workshop poster session

- Results not available

Determined immediately during the synthesis. These divisors are extracted only once and updated as the synthesis proceeds. This provides the ability to have the weights of potential divisors accurately and dynamically recomputed during decomposition and factorization. Either single cubes having two literals or double-cube divisors along with their complements are selected at each step, which provides an added advantage over performing the “gcx” and “gkx” commands separately as done in MIS.
4.3 Output phase assignment in multi-level logic decomposition

Table 4.3 summarizes the results obtained on multi-level circuits as inputs. The Script given in Figure 4.1 is used to run MIS21 and technology mapping is done using MIS21.

<table>
<thead>
<tr>
<th>Name</th>
<th>Gates</th>
<th>Area</th>
<th>CPU sec</th>
<th>Gates</th>
<th>Area</th>
<th>CPU sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>b9</td>
<td>79</td>
<td>184</td>
<td>1.0</td>
<td>87</td>
<td>197</td>
<td>13.5</td>
</tr>
<tr>
<td>des</td>
<td>1827</td>
<td>4965</td>
<td>2941</td>
<td>2354</td>
<td>6456</td>
<td>990</td>
</tr>
<tr>
<td>C17</td>
<td>5</td>
<td>12</td>
<td>1</td>
<td>5</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>C499</td>
<td>233</td>
<td>619</td>
<td>22</td>
<td>239</td>
<td>707</td>
<td>48.7</td>
</tr>
<tr>
<td>C880</td>
<td>241</td>
<td>591</td>
<td>53</td>
<td>234</td>
<td>618</td>
<td>44</td>
</tr>
<tr>
<td>C1908</td>
<td>249</td>
<td>715</td>
<td>34.7</td>
<td>336</td>
<td>855</td>
<td>8021</td>
</tr>
</tbody>
</table>

Table 4.3 Area minimization of multi-level circuits

4.3 Output phase assignment in multi-level logic decomposition

The output phase assignment problem deals with assigning a true or complement phase to the primary output in order to reduce the final area. It is well known that for a PLA, implementing $f$ or $\bar{f}$ can make a substantial difference in the final area. This has motivated the study of the output phase assignment for PLA minimization. A program, PLAYGROUND, that explores the tradeoffs in implementing logic functions with positive or negative phase is reported in [WeyCha88]. It is shown experimentally that the number of product terms can be further reduced by as much as 50% of the results obtained if only the true logic output minimization are allowed.

However, for multi-level logic, this should not be a factor, since either phase of an output can be obtained by inserting an inverter [BrHaSa90]. Hence, as mentioned in [BrHaSa90], output phase assignment should not drastically change the quality of the final logic if both the positive and negative phases are considered for each node of the multi-level logic. Another study that explores the tradeoffs in multi-level logic with and without phase assignment is reported in [WeChJo89]. When a multi-output function, $(z_1, z_2, \ldots, z_m)$, of multi-level logic is to be realized by complex gates, either $z_i$ or $\bar{z}_i$ can be realized for each output. Since there are $2^m$ different output phase assignments for $m$ output functions.
an efficient nonexhaustive phase assignment algorithm called *output phase assignment algorithm for multi-level logic optimization* (OPAM) is proposed in [WeChJo89]. After a proper phase assignment the regular multi-level logic synthesis using MIS2.1 [BrRSW87a] is invoked Table 4.4 shows the experimental results on the 41 Book PLAs [BrHaMS84]. The script used by the authors for running MIS2.1 is given in Figure 4.2. Columns 4 and 5 of Table 4.4 represent the literal count of the optimized networks with MIS2.1 and with OPAM.

```
collapse
sweep
decomp -g
gcx -ab
resub -a; sweep
gcx -b
resub -a, sweep
resub -a;
print_stats -f
```

**Figure 4.2** Script used to generate columns 4 and 5 of Table 4.4

For the 30 PLAs, out of 41 PLAs, OPAM has synthesized networks with smaller literal counts compared to that of MIS2.1. It can be seen that for the example misg the literal count is reduced by 28%. Pendulum was run on the same 41 PLAs and the results are reported in columns 7 and 8 of Table 4.4. For 27 out of 41 PLAs, Pendulum has produced networks with smaller literal counts compared to that of MIS2.1 and for 20 PLAs the results are superior to that of OPAM. It can be seen that for the example, x6dn, the literal count is reduced by 50% to that of MIS2.1.

For the 41 PLAs the total literal counts produced by MIS2.1, OPAM and Pendulum are 9103, 8591 and 7539 respectively. This indicates a literal count reduction of 5.62% for OPAM compared to MIS2.1 and 17.18% for Pendulum compared to MIS2.1. It should be pointed out here that Pendulum has no separate phase assignment phase but the method examines double-cube divisors and single-cube divisors along with their complements during its evaluation and extraction phases. These experimental results clearly indicate that a separate output phase assignment may not be necessary if the objects that are used for decomposition and factorization consider both the true and the complement phases in order to evaluate and extract them.
4.4 Boolean decomposition in multi-level logic optimization

It was evident from Algorithm 1 that the generation of candidate sub-expressions...
Boolean decomposition in multi-level logic optimization plays a key role in the quality of the final multi-level logic produced. These candidate sub-expressions may also be Boolean (strong) divisors rather than only algebraic (weak) divisors [Brayt87a]. A multiple-valued Boolean minimization technique for identifying and extracting Boolean factors which can be used as strong divisors is proposed in [DeWaNS89]. The results obtained from such a technique is given in Table 4.5.

<table>
<thead>
<tr>
<th>Name</th>
<th>Algebraic script</th>
<th>Boolean script</th>
<th>Pendulum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Literals</td>
<td>CPU Sec</td>
<td>Literals</td>
</tr>
<tr>
<td>5xp1</td>
<td>155</td>
<td>64.2</td>
<td>109</td>
</tr>
<tr>
<td>9sym</td>
<td>219</td>
<td>384.3</td>
<td>113</td>
</tr>
<tr>
<td>Z5xp1</td>
<td>177</td>
<td>58.9</td>
<td>109</td>
</tr>
<tr>
<td>add6</td>
<td>77</td>
<td>1201.8</td>
<td>75</td>
</tr>
<tr>
<td>alu3</td>
<td>97</td>
<td>29.7</td>
<td>97</td>
</tr>
<tr>
<td>clpl</td>
<td>19</td>
<td>1.7</td>
<td>19</td>
</tr>
<tr>
<td>radd</td>
<td>54</td>
<td>21.5</td>
<td>51</td>
</tr>
<tr>
<td>rd53</td>
<td>57</td>
<td>11.9</td>
<td>43</td>
</tr>
<tr>
<td>rd73</td>
<td>112</td>
<td>142.4</td>
<td>80</td>
</tr>
<tr>
<td>rd84</td>
<td>126</td>
<td>917.7</td>
<td>117</td>
</tr>
<tr>
<td>root</td>
<td>154</td>
<td>78.5</td>
<td>178</td>
</tr>
<tr>
<td>sao2</td>
<td>164</td>
<td>72.6</td>
<td>181</td>
</tr>
<tr>
<td>sym10</td>
<td>292</td>
<td>1101.5</td>
<td>183</td>
</tr>
<tr>
<td>vg2</td>
<td>92</td>
<td>71.6</td>
<td>112</td>
</tr>
<tr>
<td>x1dn</td>
<td>104</td>
<td>77.8</td>
<td>103</td>
</tr>
<tr>
<td>x2dn</td>
<td>221</td>
<td>293.9</td>
<td>220</td>
</tr>
<tr>
<td>x7dn</td>
<td>421</td>
<td>7192.8</td>
<td>393</td>
</tr>
<tr>
<td>z4ml</td>
<td>41</td>
<td>14.3</td>
<td>41</td>
</tr>
<tr>
<td>Total</td>
<td>2582</td>
<td>11737.1</td>
<td>2224</td>
</tr>
</tbody>
</table>

Table 4.5 Comparison of Pendulum with algebraic and Boolean script

For each example, MIS was made to execute an algebraic optimization script (with two-level Boolean minimization). The number of literals and CPU time on VAX 11/8800 using algebraic script is given in columns 2 and 3. Then selected Boolean factors were first used as strong divisors before executing the same algebraic script. The literal count and
CPU time obtained using Boolean division + algebraic scripts, called Boolean script, are given in columns 4 and 5 of Table 4. It was concluded in [DeWaNS89] that in almost all cases, better or comparable results were achieved in significantly faster time using the Boolean division with an algebraic script rather than algebraic script alone.

In our experiments, Pendulum was run on the same 18 PLAs and the results are reported in columns 6 and 7. For these 18 examples, Boolean division + algebraic scripts produced better results in 15 examples rather than using only algebraic script. For 11 out of 18 PLAs (10 out of 18 PLAs), Pendulum has produced networks with smaller literal counts compared to that of MIS using only algebraic script (Boolean division + algebraic script). For these 18 PLAs the total literal counts produced by MIS with algebraic script, MIS with Boolean division + algebraic script, and Pendulum are 2582, 2224 and 2059 respectively. This indicates the literal count reduction of 13.86% for MIS with Boolean division + algebraic script compared to MIS with algebraic script, and 20.25% for Pendulum compared to MIS with algebraic script. The total CPU time required by MIS to achieve these results is 131816 CPU sec on VAX 11/8800 compared to 3523 sec for Pendulum on SUN 3/260.

4.5 Comparison with BOLD

In this section we compare results obtained with various logic synthesis systems that are reported in [BoHaJL87]. The results are given in Table 4.6

The second, third and sixth columns of Table 4.6 show the results from 3 logic synthesis systems YSC [BrBrCD85], MIS [BrRSW87a], and BOLD [BoHaJL87]. The fourth column of data is obtained by first running the examples through MIS and then running the MIS output files through BOLD. Note that the MIS script used to generate the numbers of column 3 differs from the standard MIS script that was used to generate the MIS portion of column 4. The fifth column contains results for various arithmetic examples from behavioral descriptions in CHDL which were first extracted and then minimized by the BOLD system. The seventh column of data is obtained by running Pendulum

These results clearly indicate the strength of the logic synthesis system BOLD. The BOLD system alone or the MIS+BOLD combination gives the best results for many examples. BOLD features a method for multi-level logic minimization based on the ESPRESSO
### Table 4.6 Comparison of Pendulum with various logic synthesis systems

<table>
<thead>
<tr>
<th>Name</th>
<th>YSC</th>
<th>MIS</th>
<th>MIS+BOLD</th>
<th>CHDL+BOLD</th>
<th>BOLD</th>
<th>Pendulum (Fact)</th>
</tr>
</thead>
<tbody>
<tr>
<td>con1</td>
<td>19</td>
<td>19</td>
<td>22</td>
<td>-</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>f2</td>
<td>24</td>
<td>24</td>
<td>28</td>
<td>-</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>z4ml</td>
<td>36</td>
<td>38</td>
<td>37</td>
<td>36</td>
<td>37</td>
<td>44</td>
</tr>
<tr>
<td>rd53</td>
<td>39</td>
<td>40</td>
<td>37</td>
<td>30</td>
<td>36</td>
<td>48</td>
</tr>
<tr>
<td>misex1</td>
<td>65</td>
<td>57</td>
<td>54</td>
<td>-</td>
<td>53</td>
<td>71</td>
</tr>
<tr>
<td>vg2</td>
<td>87</td>
<td>88</td>
<td>86</td>
<td>-</td>
<td>90</td>
<td>96</td>
</tr>
<tr>
<td>misex2</td>
<td>111</td>
<td>109</td>
<td>111</td>
<td>-</td>
<td>106</td>
<td>108</td>
</tr>
<tr>
<td>alupla</td>
<td>125</td>
<td>146</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>5xp1</td>
<td>113</td>
<td>129</td>
<td>91</td>
<td>71</td>
<td>93</td>
<td>109</td>
</tr>
<tr>
<td>f51mm</td>
<td>124</td>
<td>130</td>
<td>85</td>
<td>67</td>
<td>97</td>
<td>*</td>
</tr>
<tr>
<td>rd73</td>
<td>115</td>
<td>143</td>
<td>86</td>
<td>48</td>
<td>81</td>
<td>98</td>
</tr>
<tr>
<td>sao2</td>
<td>158</td>
<td>150</td>
<td>131</td>
<td>-</td>
<td>154</td>
<td>149</td>
</tr>
<tr>
<td>bw</td>
<td>203</td>
<td>209</td>
<td>156</td>
<td>-</td>
<td>169</td>
<td>227</td>
</tr>
<tr>
<td>9sym</td>
<td>236</td>
<td>260</td>
<td>223</td>
<td>64</td>
<td>209</td>
<td>74</td>
</tr>
<tr>
<td>rd84</td>
<td>136</td>
<td>268</td>
<td>134</td>
<td>66</td>
<td>159</td>
<td>135</td>
</tr>
<tr>
<td>9symml</td>
<td>225</td>
<td>279</td>
<td>226</td>
<td>64</td>
<td>193</td>
<td>*</td>
</tr>
<tr>
<td>misex3</td>
<td>608</td>
<td>465</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1761</td>
</tr>
<tr>
<td>duke2</td>
<td>477</td>
<td>478</td>
<td>-</td>
<td>-</td>
<td>360</td>
<td>420</td>
</tr>
<tr>
<td>misex3c</td>
<td>638</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>577</td>
</tr>
</tbody>
</table>

- Results not available
* Benchmarks not available

[BrHaMS84] which has the following 3 distinct features.

1. ESPRESSO algorithm redesigned for the context of multi-level network.

2. Introduction of Boolean resubstitution procedure for multi-level logic minimization.


It is unlikely to get results that are obtained by BOLD with algebraic decomposi-
tion and factorization approaches followed by Boolean simplification techniques. However, these techniques for Boolean factoring and multi-level Boolean minimization require a very large amount of CPU time [DeWaNS89]

4.6 Technology mapping

In the previous sections, the results are compared in terms of the final literal counts. Since the number of literals translates roughly into transistor pairs, similar reductions in chip size can be expected. In this section, we compare the results in terms of the number of actual gates required after technology mapping.

4.6.1 Comparison with DECAF, SOCRATES and DAGON

In this section, we compare the results with various logic synthesis systems in terms of the number of gates after technology mapping.

<table>
<thead>
<tr>
<th>Gate</th>
<th>Function</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv1x</td>
<td>$\bar{a}$</td>
<td>0.5</td>
</tr>
<tr>
<td>inv2x</td>
<td>$\bar{a}$</td>
<td>1.0</td>
</tr>
<tr>
<td>Nand2</td>
<td>$a \land b$</td>
<td>1.0</td>
</tr>
<tr>
<td>Nand3</td>
<td>$a \land b \land c$</td>
<td>1.5</td>
</tr>
<tr>
<td>Nand4</td>
<td>$a \land b \land c \land d$</td>
<td>2.0</td>
</tr>
<tr>
<td>Nand5</td>
<td>$a \land b \land c \land d \land e$</td>
<td>2.5</td>
</tr>
<tr>
<td>Nor2</td>
<td>$\overline{a + b}$</td>
<td>1.0</td>
</tr>
<tr>
<td>Nor3</td>
<td>$\overline{a + b + c}$</td>
<td>1.5</td>
</tr>
<tr>
<td>Nor4</td>
<td>$\overline{a + b + c + d}$</td>
<td>2.0</td>
</tr>
<tr>
<td>aoI21</td>
<td>$(a1 \land a2) + b$</td>
<td>1.5</td>
</tr>
<tr>
<td>aoI22</td>
<td>$(a1 \land a2) + (b1 \land b2)$</td>
<td>2.0</td>
</tr>
<tr>
<td>oai22</td>
<td>$(a1 + a2) \land (b1 + b2)$</td>
<td>2.0</td>
</tr>
<tr>
<td>oai32</td>
<td>$(a1 \land a2 \land a3) + (b1 \land b2)$</td>
<td>2.5</td>
</tr>
<tr>
<td>oai222</td>
<td>$(a1 \land a2) + (b1 \land b2) + (c1 \land c2)$</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Table 4.7 Technology description
Table 4.8 Two-input equivalent gates required for mapping

<table>
<thead>
<tr>
<th>Name</th>
<th>DECAF</th>
<th>SOCRATES</th>
<th>DAGON</th>
<th>Pendulum</th>
</tr>
</thead>
<tbody>
<tr>
<td>5xp1</td>
<td>83.5</td>
<td>104.5</td>
<td>84.5</td>
<td>104.5</td>
</tr>
<tr>
<td>9sym</td>
<td>50.0</td>
<td>182.0</td>
<td>70.0</td>
<td>54.5</td>
</tr>
<tr>
<td>bw</td>
<td>139.0</td>
<td>130.0</td>
<td>140.0</td>
<td>117.5</td>
</tr>
<tr>
<td>con</td>
<td>15.0</td>
<td>15.0</td>
<td>15.5</td>
<td>15.5</td>
</tr>
<tr>
<td>duke2</td>
<td>298.0</td>
<td>318.0</td>
<td>315.0</td>
<td>285.5</td>
</tr>
<tr>
<td>f2</td>
<td>16.0</td>
<td>16.0</td>
<td>20.0</td>
<td>18.0</td>
</tr>
<tr>
<td>rd53</td>
<td>25.0</td>
<td>29.0</td>
<td>32.5</td>
<td>35.5</td>
</tr>
<tr>
<td>rd73</td>
<td>85.5</td>
<td>101.5</td>
<td>77.5</td>
<td>68.0</td>
</tr>
<tr>
<td>rd84</td>
<td>129.0</td>
<td>184.5</td>
<td>110.5</td>
<td>95.0</td>
</tr>
<tr>
<td>sao2</td>
<td>110.0</td>
<td>154.5</td>
<td>112.0</td>
<td>110.5</td>
</tr>
<tr>
<td>vg2</td>
<td>72.5</td>
<td>76.0</td>
<td>70.5</td>
<td>70.0</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1023.5</td>
<td>1311.0</td>
<td>1048.0</td>
<td>974.5</td>
</tr>
</tbody>
</table>

Table 4.8 shows the results of running DECAF [LiKeBr87], SOCRATES [GrBadH86], and DAGON [Keutze87] compiled in [LiKeBr87]. DAGON used MIS as the technology independent portion of the synthesis system, with a separate, manually designed script for each example [LiKeBr87]. For the same circuits, Pendulum is used for technology independent optimization and MIS2 1 is used for technology mapping using the technology description shown in Table 4.7. The number of two-input equivalent gates required is given in column 5 of Table 4.8. As can be seen, Pendulum synthesis networks are comparable or better than the other 3 systems.

4.6.2 MCNC89 circuits optimization

In this section, we compare the literal counts, area, gates and delay of the final logic produced for the MCNC 1989 benchmark circuits [Lisank89] obtained by using Pendulum and MIS 2 1. MIS2 1 is run using the script given in Figure 4.3 and the technology mapping is done using MIS2 1.
Figure 4.3 Script used to generate column 2 of Table 4.9

<table>
<thead>
<tr>
<th>Name</th>
<th>Literals</th>
<th>Area</th>
<th>Gates</th>
<th>Delay</th>
<th>Literals(SOP)</th>
<th>Area</th>
<th>Gates</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>5xp1</td>
<td>157</td>
<td>113215</td>
<td>63</td>
<td>8.99</td>
<td>163</td>
<td>130384</td>
<td>74</td>
<td>9.98</td>
</tr>
<tr>
<td>9sym</td>
<td>255</td>
<td>191632</td>
<td>100</td>
<td>16.55</td>
<td>265</td>
<td>189312</td>
<td>91</td>
<td>13.74</td>
</tr>
<tr>
<td>add6</td>
<td>104</td>
<td>83964</td>
<td>54</td>
<td>13.98</td>
<td>84</td>
<td>72384</td>
<td>45</td>
<td>10.77</td>
</tr>
<tr>
<td>apex1</td>
<td>2074</td>
<td>1652768</td>
<td>695</td>
<td>24.80</td>
<td>1101</td>
<td>925680</td>
<td>545</td>
<td>25.69</td>
</tr>
<tr>
<td>apex2</td>
<td>1924</td>
<td>1508000</td>
<td>780</td>
<td>25.10</td>
<td>1470</td>
<td>1231920</td>
<td>684</td>
<td>31.23</td>
</tr>
<tr>
<td>apex3</td>
<td>2870</td>
<td>2214672</td>
<td>1109</td>
<td>30.64</td>
<td>1976</td>
<td>1586416</td>
<td>815</td>
<td>33.20</td>
</tr>
<tr>
<td>apex4</td>
<td>970</td>
<td>764672</td>
<td>438</td>
<td>19.86</td>
<td>893</td>
<td>717808</td>
<td>410</td>
<td>17.65</td>
</tr>
<tr>
<td>chkn</td>
<td>447</td>
<td>339184</td>
<td>177</td>
<td>18.56</td>
<td>468</td>
<td>359600</td>
<td>192</td>
<td>17.10</td>
</tr>
<tr>
<td>duke2</td>
<td>515</td>
<td>404608</td>
<td>225</td>
<td>16.99</td>
<td>435</td>
<td>373520</td>
<td>230</td>
<td>15.43</td>
</tr>
<tr>
<td>e64</td>
<td>253</td>
<td>249632</td>
<td>190</td>
<td>65.71</td>
<td>254</td>
<td>259376</td>
<td>196</td>
<td>49.81</td>
</tr>
<tr>
<td>rd73</td>
<td>117</td>
<td>89088</td>
<td>50</td>
<td>11.37</td>
<td>105</td>
<td>82128</td>
<td>45</td>
<td>9.34</td>
</tr>
<tr>
<td>rd84</td>
<td>198</td>
<td>148944</td>
<td>81</td>
<td>14.73</td>
<td>143</td>
<td>115072</td>
<td>65</td>
<td>11.91</td>
</tr>
<tr>
<td>sa02</td>
<td>201</td>
<td>144768</td>
<td>75</td>
<td>14.16</td>
<td>171</td>
<td>129920</td>
<td>68</td>
<td>11.79</td>
</tr>
<tr>
<td>seq</td>
<td>1945</td>
<td>1531200</td>
<td>819</td>
<td>26.34</td>
<td>948</td>
<td>785088</td>
<td>467</td>
<td>23.64</td>
</tr>
<tr>
<td>xor9</td>
<td>32</td>
<td>23200</td>
<td>13</td>
<td>18.03</td>
<td>32</td>
<td>22272</td>
<td>12</td>
<td>7.28</td>
</tr>
</tbody>
</table>

Table 4.9 MCNC 'h9 circuits optimization results

As Pendulum performs only technology independent optimizations MIS2 is
### 4.7 The effect of limiting sizes of double-cube divisors

The literal count and the CPU time presented in previous sections are obtained by techniques that extract cube divisors from the network irrespective of the number of literals in them. Consequently, the synthesis procedure may generate expressions that have a larger number of literals in them. However, during technology mapping, these larger expressions may be further decomposed to map these expressions to a set of library primitives. A series of experiments are performed to limit the number of literals in each expression generated during the decomposition. This was performed by limiting the number of literals, \( n \), in the double-cube divisors. The final decomposed functions were mapped to a particular library. The literal count in SOP form, the number of gates, the final area, and the CPU time for one single-output minimized benchmarks, \( \text{apex4} \), is given in Table 4.11.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>MIS2.1</th>
<th>Pendulum</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literals</td>
<td>12062</td>
<td>8508</td>
<td>29.5%</td>
</tr>
<tr>
<td>Area</td>
<td>9459547</td>
<td>6980880</td>
<td>26.2%</td>
</tr>
<tr>
<td>Gates</td>
<td>4869</td>
<td>3939</td>
<td>19.1%</td>
</tr>
<tr>
<td>Delay</td>
<td>325.81</td>
<td>288.56</td>
<td>11.4%</td>
</tr>
</tbody>
</table>

Table 4.10 Percentage improvement in performance between MIS2.1 and Pendulum

used for technology mapping. Table 4.9 summarizes the results obtained by MIS2.1 and Pendulum. Pendulum extracts two-literal single-cube divisors and double-cube divisors along with their complements. But MIS2.1 extracts kernels and single-cube divisors of arbitrary sizes. The purpose of this experiment is to study and compare the final area, number of gates and delay parameters due to different decomposition objects employed in both systems. It can be seen from Table 4.10 that Pendulum produces smaller circuits in terms of literal count, area, and number of gates, and that the final logic is faster than that of MIS2.1. This result clearly indicates the superiority of results using simpler objects for decomposition and factorization, and concurrency involved during extraction rather than extracting arbitrary size kernels, since large expressions may be further decomposed by the technology mapper to map into a set of simpler library elements.
Table 4.11  Effect of limiting sizes of divisor on single-output minimized sop for benchmark

<table>
<thead>
<tr>
<th>n</th>
<th>final lit SOP</th>
<th>no of gates</th>
<th>Area</th>
<th>CPU Sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2657</td>
<td>1197</td>
<td>3363</td>
<td>446.8</td>
</tr>
<tr>
<td>3</td>
<td>2653</td>
<td>1205</td>
<td>3364</td>
<td>466.6</td>
</tr>
<tr>
<td>4</td>
<td>2605</td>
<td>1198</td>
<td>3339</td>
<td>451.9</td>
</tr>
<tr>
<td>5</td>
<td>2581</td>
<td>1226</td>
<td>3377</td>
<td>532.9</td>
</tr>
<tr>
<td>6</td>
<td>2549</td>
<td>1195</td>
<td>3334</td>
<td>676.1</td>
</tr>
<tr>
<td>7</td>
<td>2511</td>
<td>1188</td>
<td>3317</td>
<td>1233.8</td>
</tr>
<tr>
<td>8</td>
<td>2473</td>
<td>1177</td>
<td>3297</td>
<td>2423.0</td>
</tr>
<tr>
<td>9</td>
<td>2449</td>
<td>1183</td>
<td>3314</td>
<td>4741.6</td>
</tr>
<tr>
<td>10</td>
<td>2438</td>
<td>1181</td>
<td>3307</td>
<td>7906.3</td>
</tr>
<tr>
<td>Full</td>
<td>2427</td>
<td>1171</td>
<td>3287</td>
<td>19271.6</td>
</tr>
</tbody>
</table>

is increased by 9%, the number of gates and the final area is increased by 2% compared to full synthesis. Yet there is a striking reduction of CPU time by 98%. This is mainly because the total number of double-cube divisors processed during the partial synthesis is much smaller compared to the total number of double-cube divisors that would have been processed during the full synthesis. For the same benchmark, by limiting the number of literals in double-cube divisors to less than or equal to 10, we can obtain the same quality of results as that of the full synthesis but in 50% less CPU time. In general for large circuits by controlling the value of $n$, one may obtain the same quality of results as that of full synthesis within a reasonable amount of CPU time.
Chapter 5 Testability-preserving transformations

The decomposition and factorization algorithms presented in previous chapters can be used for the area optimization of multilevel combinational logic. The area reduction is accomplished by consolidating various identical components into one component. However, it is necessary to test the resulting circuit. The huge amount of logic that can be synthesized may create problems in testing. Also, the lack of familiarity with the synthesized logic may even make testing difficult.

It is shown in [BaBrJH88] that if a Boolean network is prime and irredundant then it is 100% testable for single stuck-at faults. For two-level logic synthesis, algorithms that can create logic which is prime and irredundant are given in [BrHaMS84, DaAgRu86]. One approach for area optimization of multi-level logic that can generate circuits which are prime and irredundant and hence 100% testable for single stuck-at faults is given in [BoHaJL87]. In reality, these results cannot be achieved with reasonable resources.

The current trend in logic synthesis is to focus on testability as an optimization criterion and to optimize the area of the logic [HaJaKM89]. Some of the results from [HaJaKM89] are given in Section 1.3.9. The testability preservation results are proved for networks which are multi-fault testable. It is shown that multi-fault testability is invariant and is maintained by algebraic factorization procedures. Hence these results can be applied primarily to minimized two-level circuits. Since the method has to be applied to both two-level and multi-level circuits, single stuck-at fault testability invariant is a better criterion rather than multi-fault testability.

In this chapter, we consider networks, two-level or multi-level, which are completely single-fault testable, and demonstrate the conditions under which the synthesized
circuit can be fully tested by the original test set [VasRa90a,Ra9Vas90]. The results are proved for all the transformations introduced: 1) division by a single cube divisor, 2) division by a double-cube divisor, and 3) division by a double-cube divisor and its complement. We demonstrate that the introduced transformations performed on single output sum of products sub-expressions preserve testability, which is defined in Section 5.1. Finally, we extend the results to multi-output networks. Since the method assumes that the initial network is only single stuck-at fault testable, and because single stuck-at fault testability is maintained through the transformations, the method can be applied to a large class of irredundant two-level or multi-level circuits. Experimental results of the testability related properties are provided at the end of this chapter.

5.1 Testability preservation

Let $N_1$ be a network and $F$ a transformation which transforms $N_1$ to a network $N_2$, denoted as $N_1 \xrightarrow{F} N_2$. Let $T$ be a complete test set for single stuck-at faults for $N_1$. We say that transformation $F$ preserves testability, if $T$ is a complete test set for single stuck-at faults in network $N_2$.

In order to prove that a transformation preserves testability, we consider decomposition and factorization to be a sequence of transformations, where each time a network $N_1$ is transformed to $N_2$. In the transformations used in this thesis it is assumed that $N_1$ is a single-output sum-of-products sub-expression as shown in Figure 5.1. Note that these single-output sum-of-products sub-expressions can have both primary inputs and intermediate outputs in them. In fact, only a part of the network $N_1$, namely $K_1$, is transformed to $K_2$, generating a new network $N_2$. Naturally $K_1$ and $K_2$ perform the same function.
5.2 Testability preservation of single-cube extraction

Lemma 7 A complete test set $T$ detects all single stuck-at faults in $\mathcal{N}_1$, outside of $K_1$, $(\mathcal{N}_1 \setminus K_1)$, if and only if $T$ detects all single stuck-at faults in $\mathcal{N}_2$, outside of $K_2$, $(\mathcal{N}_2 \setminus K_2)$.

Proof Since $K_1$ and $K_2$ perform identical functions, then any two corresponding nodes in $(\mathcal{N}_1 \setminus K_1)$ and $(\mathcal{N}_2 \setminus K_2)$ also perform the same function. This implies that a fault in $(\mathcal{N}_1 \setminus K_1)$ can be sensitized and propagated if and only if the corresponding fault in $(\mathcal{N}_2 \setminus K_2)$ can be sensitized and propagated. Q E D

Since the testability outside the region of modification is preserved, in order to prove the testability preservation of a transformation, it is sufficient to demonstrate that a complete test set for $K_1$ fully tests $K_2$.

We will show that some of the transformations, like single-cube and double-cube extraction, are such that each line in $K_2$ has a corresponding one or more lines in $K_1$ that perform the same function. Similarly, each fault in $K_2$ has a corresponding one or more faults in $K_1$. Corresponding faults occur on corresponding lines and have the same faulty values. The testability preservation of a transformation can be proved, if every fault in $K_2$ has at least one corresponding fault in $K_1$ such that the detection of the fault in $K_1$ implies the detection of the fault in $K_2$. For other transformations, like dual expression extraction, testability preservation is proved using the relations between complete test sets for the dual circuits, rather than the concept of corresponding faults.

5.2 Testability preservation of single-cube extraction

Single cube extraction is the process of extracting cubes which are common to two or more cubes. The common part is then created as a new intermediate node as shown in Figure 5.2. The transformation performed is as follows: from the expression $f = abA_1 + abA_2 + ... + abA_n$, a cube, $C = ab$, is extracted and substituted to obtain $f = cA_1 + cA_2 + ... + cA_n$. We show in Theorem 4 that if all faults in $K_1$ propagate to $f$, for any test set $T$, then all faults in $K_2$ also propagate to $f$ for $T$.

Theorem 4: Single-cube extraction in a single-output sum-of-products sub-expression preserves testability.
5.2 Testability preservation of single cube extraction

**Figure 5.2** Division by a single cube

*Proof.* Notice that only the faults in \( L_2 \) of \( K_2 \) have to be considered. The detectability of faults in \( K_2 \) outside \( L_2 \) remains unchanged compared to \( K_1 \).

Any fault in \( L_2 \) appears as a \( D \) or \( D \) symbol [Roth66] on the output of \( f \).

**Case 1:** \( C = D \)

Every fault in \( L_2 \) has \( n \) corresponding faults in \( L_1 \). Any complete test \( T \) for \( K_1 \) contains a vector \( t \) that detects a fault that appears as \( D \) on \( c_i \), \( (1 \cdot i \cdot n) \), on output \( f \). Such a test vector has the following implications.
5.3 Testability preservation of double-cube extraction

\{C_i = D \land f = D\}_{K_1}
\Rightarrow \{A_1 = 0, \ldots, A_i = 1, \ldots, A_n = 0\}_{K_1}
\Rightarrow \{B_1 = 0, \ldots, B_i = D, \ldots, B_n = 0\}_{K_1}
\Rightarrow \{B_1 = 0, \ldots, B_i = D, \ldots, B_n = 0\}_{K_2}
\Rightarrow \{f = D\}_{K_2}

Case 2: \(C = D\).

Any complete test \(T\) for \(K_1\) contains a vector \(t\) that detects a fault that appears as \(D\) on \(r_i\), \((1 \leq i \leq n)\), on output \(f\). Such a test vector has the following implications

\{C_i = D \land f = D\}_{K_1}
\Rightarrow \{A_1 = \{0, 1\}, \ldots, A_i = 1, \ldots, A_n = \{0, 1\}\}_{K_1}
\Rightarrow \{B_1 = 0, \ldots, B_i = D, \ldots, B_n = 0\}_{K_1}
\Rightarrow \{B_1 = \{0, D\}, \ldots, B_i = D, \ldots, B_n = \{0, D\}\}_{K_2}
\Rightarrow \{f = D\}_{K_2}

Hence, any complete test for \(K_1\) is also complete for \(K_2\). Notice that in the proof we use a 16-valued \(D\) alphabet as described in [RajCox90].

Q.E.D

5.3 Testability preservation of double-cube extraction

The double-cube extraction transformation consists of extracting a double cube from a single-output sum-of-products sub-expression. \(AC + CB \Rightarrow C(A + B)\), and is shown in Figure 5.3.

\textit{Theorem 5.} Double-cube extraction in a single-output sum-of-products sub-expression...
expression preserves testability.

**Proof:** Since both $K_1$ and $K_2$ in Figure 5.3 are internally fanout-free circuits, only the faults on inputs and fanout branches have to be considered.

**Case 1:** Stuck-at faults on $g_1$

Every fault on $g_1$ of $K_2$ has a corresponding fault on $g_1$ of $K_1$. If the fault is represented by a $D$ symbol that appears on $g_1$ of $K_1$, then any complete test set for $K_1$ contains a vector that detects the corresponding fault in $K_2$ due to the following implications:

$$\{g_1 = D \land f = D\} \rightarrow \{g_2 = 1, g_3 = 1, g_4 = 0, g_6 = 0, g_5 = D\} \rightarrow \{g_4 = 0, g_{23} = 1, g_6 = D\} \rightarrow \{f = D\}$$

Similarly, we can show that it is also true for faults represented by $D$.

**Case 2:** Stuck-at faults on $g_4$
The argument is analogous to case 1.

Case 3 Stuck-at faults on inputs C

Every fault on \( g_{23} \) of \( K_2 \) has a corresponding fault on \( g_{2}, g_{3} \) and inputs C of \( K_1 \). If the fault in \( K_1 \) is represented by a D symbol that appears on \( g_{2} \) of \( K_1 \), then any complete test set for \( K_1 \) contains a vector that detects the corresponding fault in \( K_2 \) due to the following implications:

\[
\{ g_{2} = D \land f = D \} \quad \text{in} \quad K_1
\]

\[
\Rightarrow \{ g_{1} = 1, g_{3} = D, g_{4} = 0, g_{6} = 0, g_{5} = D \} \quad \text{in} \quad K_1
\]

\[
\Rightarrow \{ g_{1} = 1, g_{4} = 0, g_{6} = 1, g_{23} = D \} \quad \text{in} \quad K_2
\]

\[
\Rightarrow \{ f = D \} \quad \text{in} \quad K_2
\]

Similarly we can show that it is also true for faults represented by \( \bar{D} \).

Notice that similar conditions are created by corresponding faults on \( g_{3} \) of \( K_1 \).

Hence, any complete test set obtained for circuit \( K_1 \) is sufficient to detect all single stuck-at faults in \( K_2 \). Q.E.D

5.4 Testability preservation of dual expression extraction

The dual expression extractions considered in the concurrent decomposition and factorization procedure are as follows:

1. Dual expression extraction in \( D\{1,1,2\} \) transforms a sum-of-products subexpression, \( f = aA + bA + \bar{a}\bar{b}B \), to \( M = a + b \) and \( f = MA + \bar{M}B \).

2. Dual expression extraction in \( D\{2,2,2\} \) transforms a sum-of-products subexpression, \( f = \bar{a}\bar{b}A + \bar{a}bA + \bar{a}bB + abB \), to \( M = \bar{a} + \bar{b} \) and \( f = MA + \bar{M}B \).
3 Dual expression extraction in $D_{2,2,3}$ transforms a sum-of-products subexpression, $f = abA + ar{b}cA + ar{a}bB + ar{b}cB$, to $M = ab + ar{b}c$ and $f = M A + ar{M} B$

In order to prove the testability preservation of dual expression extraction of an object we use Lemma 8

![Double-cube divisors and their complements](image)

**Figure 5.4** Double-cube divisors $d$ and their complements $\bar{d}$

**Lemma 8**: Let $T_1$ be a complete test for all faults that are represented by a $D$ symbol on the output of a double-cube divisor $d$. Let $T_2$ be a complete test for all faults that are represented by a $\bar{D}$ symbol on the output of a complement cube divisor $d$. Then $T_1 \cup T_2$ is a complete test set for both $d$ and $\bar{d}$ for single stuck-at faults

**Proof**:

**Case 1**: For any $d \in D_{1,1,2}$ and $\bar{d} \in S_2$

It is easy to verify for circuit A of Figure 5.4 that

$T_1 \supseteq \{ab = 11\}$

$T_2 \supseteq \{ab = 01, 10\}$
Notice that $T_1 \cup T_2 \supseteq \{ab = 01, 10, 11\}$ is a complete test for $d \in D_{1,1,2}$ and $\bar{d} \in S_2$ for both s-a-0 and s-a-1 faults.

**Case 2**: For any $d \in D_{2,2,2}$ and $\bar{d} \in D_{2,2,2}$

It is easy to verify for circuit B of Figure 5.4 that

\[
T_1 \supseteq \{ab = 00, 11\} \\
T_2 \supseteq \{ab = 01, 10\}
\]

Also, it is known that all four tests are required to test a Boolean network implementing a two input exclusive-or (exclusive-nor) function. Hence $T_1 \cup T_2$ is a complete test for both $d \in D_{2,2,2}$ and $\bar{d} \in D_{2,2,2}$.

**Case 3**: For any $d \in D_{2,2,3}$ and $\bar{d} \in D_{2,2,3}$.

It is easy to verify for circuit C of Figure 5.4 that

\[
T_1 \supseteq \{abc = 011, 01x, 100, x00\} \\
T_2 \supseteq \{abc = 11x, 110, x01, 001\}
\]

Notice that $T_1 \cup T_2 \supseteq \{abc = 001, 011, 100, 110\}$ is a complete test for $d \in D_{2,2,3}$ and $\bar{d} \in D_{2,2,3}$ for both s-a-0 and s-a-1 faults.

Q.E.D

The dual expression extraction transforms circuit $K_1$ to $K_2$ as shown in Figure 5.5

**Theorem 6**: Dual expression extraction from a single-output sum-of-products sub-expression preserves testability.

**Proof**:

**Case 1**: $M_1 = \bar{D}$

Any complete test set for $K_1$ contains a subset $T_1$ such that every fault in $E$
that appears as $D$ on $M1$ is tested. Every vector $t, t \in T_1$, has the following implications

$$\{M1 = D \land f = \overline{D}\}_{K1}\) $$

$$\Rightarrow \{M2 = 1\}_{K1}\) $$

$$\Rightarrow \{A = 1 \land B = 0\}_{K1}\) $$

These conditions, $(A = 1 \land B = 0)$, guarantee the propagation of faults in $E$ to $f$ for each vector of $T_1$ in $K2$.

**Case 2: $M2 = \overline{D}$**

Any complete test set for $K1$ contains a subset $T_2$ such that every fault in $E$ that appears as $D$ on $M2$ is tested. Every vector $t, t \in T_2$, has the following implications

$$\{M2 = \overline{D} \land f = \overline{D}\}_{K1}\) $$
5.5 Testability preservation of multi-output networks

\[ \exists \{ M1 = 1 \}_{K1} \]

\[ \exists \{ A = 0 \land B = 1 \}_{K1} \]

These conditions, \( (A = 0 \land B = 1) \), guarantee the propagation of faults in \( E \) to \( f \) for each vector of \( T_2 \) in \( K2 \).

Since any complete test set \( T \) for \( K1 \) has both \( T_1 \) and \( T_2 \), and the propagation conditions for any \( t, t \in T_1 \cup T_2 \) in \( K2 \) are guaranteed \( K2 \) is completely tested by \( T \).

Q E D

So far we have developed a set of results that are sufficient to prove single-fault testability after extraction of common expressions from single-output sum-of-products sub-expressions. Now, we will extend the results to multi-output networks.

### 5.5 Testability preservation of multi-output networks

In a multi-output network, we may have a number of nodes \( y_1, \ldots, y_k \), represented by the same expression (they have the same cover and represent the same function). Resubstitution is a transformation that replaces each copy of \( y_1, \ldots, y_k \) with a single node as in Figure 5.6. The conditions under which this transformation preserves testability are given in Theorem 7.

![Figure 5.6 Resubstitution in a multi-output network](image)

**Theorem 7** Resubstitution of common sub-expressions, in a multi-output func-
preserves testability if no two sub-expressions control the same output

Proof If there is any fault \( f \) that can be tested in \( K_1 \) by a test \( t \) and propagated to node \( y_i \), then the corresponding fault in \( K_2 \) also propagates to \( y_i \) in \( K_2 \), as there is no modification of the network \( E \) in \( K_2 \). Propagation conditions on the inputs of some \( C_1 \) in \( K_2 \), are not affected by \( y_i \) as there is no reconvergence from other branches controlled by \( y_i \). This implies that a test which detects a fault in \( K_1 \) on the primary output \( P \), also detects the corresponding fault on the primary output \( P' \) in \( K_2 \) (Q.E.D)

Corollary 1. The transformation opposite to the single cube extraction called substitution, does not preserve testability.

Example 14 Consider a multi-output function having \( n \) outputs, with single cube divisors \( M \) and \( N \) as follows

\[
M = \overline{ab} \\
N = ab \\
f_1 = M \\
f_2 = N \\
f_3 = f_1 + f_2
\]

Assume that \( M \) and \( N \) are used in other functions, say \( f_4 \) \( \ldots \) \( f_n \), so that their extraction as intermediate variables is justified. Notice that although \( f_1 \), \( f_2 \) and \( f_3 \) can be fully tested for single stuck-at faults, some faults will not propagate to \( f_3 \). If we substitute \( f_1 \) and \( f_2 \) in \( f_3 \), \( f_3 = \overline{ab} + ab \), hence \( f_3 \) contains redundancy.

We demonstrated that the algebraic transformations along with DeMorgan's laws presented in this report preserve testability in the sense that if a network, two-level or multi-level, is completely tested for single stuck-at faults by a test set \( T \), then the network obtained by any of the above transformations can also be tested for single stuck-at faults by the same test \( T \).
5.6 Testability related properties of synthesized circuits

In this section we study the testability related properties of the synthesized networks. A deterministic ATPG program, PLANET [RobRaj88], is used to generate test vectors for two-level networks and TULIP [MaaRaj90] is used to analyze the fault coverage of the synthesized multi-level network. In our results, we give the number of test vectors required to test the PLAs (reported by PLANET), the actual number of test vectors required to test the synthesized multi-level circuit for all single stuck-at faults (reported by TULIP), and the fault coverage.

The results in Table 5.1 are obtained using PLAs which are prime and irredundant for every output. These PLAs are first minimized using the single-output option of ESPRESSO [BrHaMS84] and then decomposed. Hence the input PLA comprised of a set of prime and irredundant single-output, two-level functions without product term sharing and hence no substitution is required. It can be seen from Table 5.1 that in each case the complete test set for a two-level network covers 100% of single stuck-at faults in the synthesized multi-level circuit. This experiment not only confirms the testability preservation of the transformations, but also shows the effect of the test set compression, which is due to the mapping of many fault sites into one fault as the decomposition and factorization proceed. The experimental results show that the test set for a multi-level network can be up to 10 times smaller than the test set for the corresponding two-level network. In the case of single-output minimized PLAs, the total number of test patterns required to test all the two-level structures is 12988, but 3430 test patterns are required to test all the multi-level structures, which indicates a total reduction of 1 to 4.

A second series of experiments, summarized in Table 5.2, are performed to study the effect of multiple output minimization, which does not guarantee that the circuit is prime and irredundant for every output. Therefore, unlike the previous experiment, substitution is performed on the minimized PLAs, and the product terms were shared. After decomposition it shows that for 8 out of 20 PLAs the fault coverage is less than 100%. Clearly testability with respect to a given test set is not preserved. Also, for some circuits an experiment was performed, by an ATPG program [RajCox90], where it was found that all the faults that are not covered by the test set were proven untestable. The Boolean overlap of terms from multiple outputs may create redundancy when each output is viewed separately.
5.7 Comparison of single-output and multi-output minimization

Tables 5.1 and 5.2 demonstrate the effect of single-output minimization versus multiple-output minimization in terms of the literal count and the CPU time required for synthesis. For example, for PLAs apex1 and seq, multiple-output minimization generates the smallest circuit compared to the one obtained through single-output minimization. Seq is twice as small. When multiple-output minimization is used, and is minimized approximately 50 times faster. For PLA apex2, single-output minimization is better than multiple output.
Comparison of single-output and multi-output minimization

<table>
<thead>
<tr>
<th>PLA</th>
<th>mit hits</th>
<th>final lit</th>
<th>CPU sec</th>
<th>no of pattern</th>
<th>fault coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SOP</td>
<td>two level</td>
<td>multi level</td>
<td>coverage</td>
<td></td>
</tr>
<tr>
<td>5mx1</td>
<td>353</td>
<td>163</td>
<td>1 7</td>
<td>90 43</td>
<td>97 76%</td>
</tr>
<tr>
<td>9sym</td>
<td>595</td>
<td>263</td>
<td>95 5</td>
<td>248 137</td>
<td>100%</td>
</tr>
<tr>
<td>add6</td>
<td>2551</td>
<td>85</td>
<td>220</td>
<td>384 25</td>
<td>100%</td>
</tr>
<tr>
<td>apex1</td>
<td>2842</td>
<td>1101</td>
<td>75</td>
<td>826 257</td>
<td>99 75%</td>
</tr>
<tr>
<td>apex2</td>
<td>15528</td>
<td>452</td>
<td>4498</td>
<td>4480 245</td>
<td>100%</td>
</tr>
<tr>
<td>apex3</td>
<td>3314</td>
<td>1381</td>
<td>184 5</td>
<td>848 321</td>
<td>98 66%</td>
</tr>
<tr>
<td>apex4</td>
<td>5432</td>
<td>1976</td>
<td>346 9</td>
<td>508 393</td>
<td>99 88%</td>
</tr>
<tr>
<td>chkn</td>
<td>1740</td>
<td>469</td>
<td>184 2</td>
<td>785 248</td>
<td>100%</td>
</tr>
<tr>
<td>duke2</td>
<td>990</td>
<td>435</td>
<td>13 0</td>
<td>398 115</td>
<td>99 51%</td>
</tr>
<tr>
<td>e64</td>
<td>2210</td>
<td>254</td>
<td>2 7</td>
<td>1984 75</td>
<td>100%</td>
</tr>
<tr>
<td>o64</td>
<td>195</td>
<td>130</td>
<td>21 6</td>
<td>197 196</td>
<td>100%</td>
</tr>
<tr>
<td>rd53</td>
<td>175</td>
<td>52</td>
<td>0 6</td>
<td>32 20</td>
<td>99 09%</td>
</tr>
<tr>
<td>rd73</td>
<td>903</td>
<td>106</td>
<td>11 8</td>
<td>128 41</td>
<td>100%</td>
</tr>
<tr>
<td>rd84</td>
<td>2070</td>
<td>143</td>
<td>136 6</td>
<td>256 45</td>
<td>99 96%</td>
</tr>
<tr>
<td>sao2</td>
<td>499</td>
<td>171</td>
<td>4 9</td>
<td>159 66</td>
<td>99 37%</td>
</tr>
<tr>
<td>seq</td>
<td>6225</td>
<td>948</td>
<td>63 8</td>
<td>2856 309</td>
<td>100%</td>
</tr>
<tr>
<td>vg2</td>
<td>914</td>
<td>102</td>
<td>9 3</td>
<td>320 67</td>
<td>100%</td>
</tr>
<tr>
<td>xor8</td>
<td>1152</td>
<td>28</td>
<td>53 0</td>
<td>32 8</td>
<td>100%</td>
</tr>
<tr>
<td>xor9</td>
<td>2560</td>
<td>32</td>
<td>43 7</td>
<td>32 9</td>
<td>100%</td>
</tr>
<tr>
<td>xor10</td>
<td>5632</td>
<td>36</td>
<td>984</td>
<td>32 10</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Table 5.2** Results on multiple output minimized PLAs

minimization (literal count is 422 compared to 452) For 2 out of 20 PLAs, single-output minimized PLAs resulted in smaller literal counts, when compared to multiple-output minimized PLAs. For 9 PLAs, multiple-output minimized PLAs resulted in smaller numbers of literals. The nine remaining PLAs were identical for single and multiple-output minimization. Overall, multiple-output minimized PLAs resulted in smaller literal counts in less CPU time. The total numbers are 8327 literals and 7341.86 CPU seconds for multiple-output minimized PLAs, and 10667 literals and 34851.90 CPU seconds for single-output minimized PLAs. In general, a multiple-output minimized PLAs require fewer product terms and results in smaller area and requires less CPU time.
Chapter 6

Conclusions

A heuristic solution to the decomposition and factorization of Boolean expressions has two important components: the generation and the substitution of candidate sub-expressions. The area of the final logic and the resources required to synthesize the logic depends heavily on these objects. The basic problem is to generate a rich set of possible objects without generating an excessive number from among all possible sub-expressions. Using these objects, it should be possible to reason about common algebraic divisors and duality relations between expressions. Algorithms were presented that employs only single-cube divisors as candidate sub-expressions. The disadvantage of this approach is that common factors which consist of more than one cube are not considered. Hence the method fails to reason about common algebraic multiple-cube divisors between expressions. Algorithms were also presented that employs single cubes and multiple cubes as candidate sub-expressions. The problem with this approach is that the method may generate an exponential number of objects. Both of these techniques do not consider the complement relations that may exists between objects. Hence, the methods generate an object and its complement as two distinct objects. As the objects are of arbitrary size the duality relations may not be easy to establish. Also it is not easy to prove the testability preserving properties with these complex objects.

It is shown in this thesis that two-literal single-cubes, and double-cube divisors along with their complements, form important objects for decomposition and factorization. These simple objects can be used to reason about common algebraic divisors and duality relations between expressions. It is demonstrated that reasoning about the presence of common multiple-cube algebraic expressions from a set of Boolean expressions can be performed by analyzing only the set of double-cube divisors. Since the number of double-cube divisors grows polynomially with the number of cubes of a function, the set that
must be analyzed to check for common multiple-cube divisors is much smaller than the set of all algebraic divisors. It is also shown that in order to find the duality relations that may exist between various objects, only a subset of two-literal single-cube and double-cube divisors needs to be analyzed. Also, since the duality relations between objects along with DeMorgan’s laws are exploited, the objects proposed constitute a richer set of divisors than the strictly algebraic divisors as objects.

Using this framework, a method for decomposition and factorization of Boolean expressions is provided in this thesis. The method proposed is concurrent and uses only two-literal single-cube divisors and double-cube divisors considered concurrently with their complements. The method provides the ability to compute the weight for these objects accurately and dynamically. Hence the method is entirely greedy and in each iteration it extracts the best two-literal single-cube divisor or a double-cube divisor along with its complement. This provides an added advantage over extracting multiple-cube divisors and single-cube divisors separately as done in the other techniques. As the method extracts duality relations during the evaluation and extraction, no simplification by other techniques is required.

It is also evident from the experimental results that the framework provided in this thesis has led to a superior technique for decomposition and factorization of Boolean expressions, compared to all the existing techniques for algebraic decomposition and factorization. The experimental results obtained by this method match the best known literal counts for most benchmark circuits, and in many cases the method generates Boolean networks with much smaller numbers of literals.

The simplicity of these objects also helped in proving the testability related results, assuming the fact that the initial circuit is completely single-fault testable. We demonstrated, both theoretically and experimentally, that the decomposition and factorization transformations introduced in this thesis preserve testability, which implies that a complete test set developed for an input network also gives complete coverage of faults in the synthesized multi-level network. These results are proved using the concepts of corresponding faults in the circuits and the relations between complete test sets. Since the method assumes that the initial network is only single stuck-at fault testable, and because single stuck-at fault testability is maintained through the transformations, the method can be applied to a large class of irredundant two-level or multi-level circuits.
Bibliography


[Mathon85] H J Mathony.. The algebraic minimization of multiple-output Boolean func-
trons. Technical Report, Institute for Information Processing Techniques, University of Karlsruhe. 1985


function generators."
Proceedings of the 23rd Design Automation Conference 1986
pp 86-93

[Trevil87] L Trevillyan. "An overview of logic synthesis system."
Proceedings of the 24th Design Automation Conference, 1987, pp 166-172

[VasRa90a] J Vasudevanurthy, and J Rajski. "Test generation and synthesis of fully
testable multi-level networks." 13th annual IEEE design for testability workshop, Vail
Colorado, April 1990

and factorization of Boolean expressions." Proceedings of the International Conference

CCVLSI, 1990, pp 631-638

[WeyCha88] C L Wey, and T Chang. "PLAYGROUND Minimization of PLAs with mixed
pp. 421-426

[WeChJo89] C L Wey, S N Chang, and J Y Jou. "An efficient output phase assignment
for multi-level logic minimization." Proceedings of the second international workshop
on logic synthesis, Research triangle park, North Carolina, USA, 1989