University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations

1995

Synthesis of multilevel pass transistor logic networks.

Arunita. Jaekel
University of Windsor

Follow this and additional works at: http://scholar.uwindsor.ca/etd

Recommended Citation
Jaekel, Arunita., "Synthesis of multilevel pass transistor logic networks." (1995). *Electronic Theses and Dissertations.* Paper 3699.

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Synthesis of Multilevel Pass Transistor Logic Networks

by

Arunita Jaekel

A Dissertation

Submitted to the Faculty of Graduate Studies through the

Department of Electrical Engineering in Partial Fulfilment

of the Requirements for the Degree of

Doctor of Philosophy at the

University of Windsor

Windsor, Ontario

May, 1995

National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-10938-0

Canada

Arunita Jaekel 1995

© All Rights Reserved

ABSTRACT

Traditional design of logic circuits involves implementing a function in terms of standard logic gates. However, this type of design does not fully exploit the unique switching properties of MOSFETs, which can lead to more efficient realizations.

Over the past decade there has been considerable interest in Pass Transistor Logic (PTL) circuits. PTL circuits implement a logic function as a network of NMOS transistors. They show enhanced performance over conventional logic in terms of both speed and area optimization as well as reduced power dissipation, particularly for certain classes of circuits.

Existing synthesis techniques for PTL are limited to two-level synthesis, similar to that used for conventional logic. In conventional logic multilevel logic implementations have been shown to provide significant improvements over two-level representations. So it is of considerable interest to develop formal multilevel design methodologies for PTL in order to exploit potential efficiencies in that circuit family. Such formal design methodologies are also necessary to avoid incorrect implementations which can result from ad hoc design of PTL networks.

This thesis deals with the development of methodologies for the systematic design of multi-level PTL networks. In this thesis, we have investigated two approaches to multi-level logic synthesis techniques for PTL networks based on the concepts of (i) factorization and (ii) decision diagrams. Both approaches have shown significant savings over known synthesis techniques for PTL networks.

To my husband, Martin

ACKNOWLEDGEMENTS

I would like to acknowledge the guidance and support provided by my supervisors Dr. S. Bandyopadhyay and Dr. G.A. Jullien. I am grateful for the time and effort they have contributed in guiding me through this work. This thesis could not have been completed without their patient and active supervision. I would also like to thank my committee members Dr. M. Ahmadi, Dr. W.C. Miller and Dr. N. Wigley for their comments and suggestions regarding the thesis. Special thanks must go to Ms. Barbara Szydlowska and Mr. Sandeep Kamat for their invaluable help in implementing the various algorithms in this thesis and to John De Ryckere and Todd Bealor for their help with Cadence and FrameMaker. I would like to express my sincere thanks to our system administrators Mr. Walid Mnaymneh and Mr. Stephen Karamatos for all their help throughout the course of this work and the members of the Graduate Lab for a friendly and enlightening working atmosphere.

I would also like to thank my parents, Tushar and Manju Sarkar, and my sister Mitun for their constant support and words of encouragement. Finally, I must thank my husband Martin, for all his help and support. Without his patience and understanding, I could not have finished this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

1.1 Introduction

The main considerations for developing high performance VLSI circuits involve reducing the power consumption, area and delay of the circuit as much as possible. MOSFETs are ideal candidates for VLSI circuit realization due to their high packing density, good noise immunity and low power dissipation. Both NMOS and CMOS technologies have been widely used in VLSI design. Traditional design of logic circuits involves implementing a function in terms of standard logic gates. However, this type of design does not fully exploit the unique switching properties of MOSFETs. Implementing a function as a switching network (where the switching elements are MOS transistors) can lead to more efficient realizations.

Over the past decade there has been considerable interest in Pass Transistor Logic (PTL) circuits. PTL circuits implement a logic function as a network of MOS transistors, where input signals may be applied to the source/drain of the transistors as well as to the gates. They show enhanced performance over conventional logic in terms of both speed and area optimization as well as reduced power dissipation, particularly for certain classes of circuits such as

multiplexers, arithmetic circuits and bus drivers [WHI83]. Several papers [MIT92], [PAS89], [PAS91B], [WHI92], [SUZ93], [YAN90] have reported on the performance of PTL but without providing general design methodologies. An extensive literature survey shows that the published synthesis techniques for PTL are limited to two-level synthesis, [PED88], [RAD85] similar to that used for conventional logic.

1.2 Motivation

With the ongoing advances in technology, it is becoming possible to fit an increasing number of active devices on a single VLSI chip. However, even with such increases, there is always a demand for more devices and more functionality on a single chip. Thus there is always a need to find more efficient realizations of circuits for any logic family. Multilevel logic implementations [BRA90], [BRA82], [BRA84], [CAR91], [MAL89], [HSU92] have been shown to provide significant improvements over two-level representations for conventional logic. There are a number of papers showing multilevel implementations of specific PTL circuits. However, they do not provide any general synthesis procedure for automatically designing such circuits. So it is of considerable interest to develop formal multilevel design methodologies for PTL in order to exploit potential efficiencies in that circuit family. Such formal design methodologies are also necessary to avoid incorrect implementations which can result from *ad hoc* design of PTL networks.

1.3 Solution Outline

This thesis proposes two new design techniques for synthesizing multilevel PTL networks. The first creates a series-parallel PTL network using the concept of algebraic factorization. The second method produces, in general, a nonseries-parallel circuit based on decision diagrams. Both of these techniques result in significant improvements over conventional PTL design methodologies.

An accepted criterion for multilevel logic design is to minimize the area occupied by a circuit, while satisfying the timing constraints placed on the longest path [RUD89].

Therefore we do not allow arbitrarily large PTL networks, in order to minimize the quadratic delay dependence on the number of series-connected transistors. A large Boolean function is first broken into a number of smaller functions and each is then implemented as a multilevel PTL network. Individual PTL networks may be cascaded, but buffering is required between two stages, in order to restore signal levels and reduce delays.

Other important design issues which must be considered in PTL implementations are [PAS91]:

(i) possible creation of direct supply-to-ground paths resulting in short circuits and high power dissipation.
(ii) existence of sneak-paths created due to the bidirectional nature of MOSFETs.

Our multilevel synthesis techniques ensure an error free design where the above situations can never occur.

1.4 Thesis Organization

Chapter 2 discusses the relevant background material required for the remainder of the thesis. It gives a a brief overview of PTL circuits and the existing synthesis procedures for PTL design. It also describes multilevel logic synthesis techniques for conventional AND/OR logic as well as circuit implementations based on binary decision diagrams (BDDs).

Chapter 3 investigates the issues involved in designing series-parallel, multilevel PTL networks using algebraic factorization techniques. We first examine why existing techniques cannot be applied directly to PTL circuits and then propose a modified factorization algorithm for synthesis of PTL circuits.

Chapter 4 describes a methodology for implementing nonseries-parallel PTL networks based on the concept of decision diagrams. We introduce a graph model, the 123-Decision

Diagram (similar to a BDD), to represent a PTL network. The synthesis procedure consists of applying various transformations to manipulate the graph model and optimize the network.

Chapter 5 discusses the experimental results for our multilevel synthesis techniques. We compare our results with those obtained from existing two-level PTL design techniques for a number of benchmark circuits. We also show simulation results for our design.

Chapter 6 discusses the conclusions of this thesis and directions for future work.

Chapter2

Review of PTL and Multilevel Logic Design

2.1 Introduction

In this chapter we review some of the relevant topics and background material required for the remainder of this thesis. In the first section we will describe some of the important issues involved in Pass Transistor Logic (PTL) design. We will analyze some of the existing techniques for synthesis of PTL circuits and discuss some of the limitations associated with these approaches. Next, we will look at some of the widely used techniques for multilevel logic synthesis, such as factorization and decomposition. These techniques can be used to transform a two-level implementation of a given circuit into a multi-level realization, usually resulting in significant savings. In later chapters, we will discuss how such techniques can be adapted for PTL Design. Finally, we will outline the binary decision diagram (BDD) approach to logic synthesis and take a brief look at some of the interesting properties of binary decision diagrams.

2.2 Pass Transistor Logic Design:

2.2.1 What is a PTL Network

A PTL network is an interconnection of a set of transistors to achieve a particular switching function. Compared to conventional logic, networks of pass transistors can lead to significant improvements in speed and area optimization as well as substantially reduce power dissipation [WHI83] particularly for functions of up to eight variables. They can also naturally implement disabled or high impedance states, making them ideal for bus drivers and multiplexer realizations [WHI83]. Pass transistor circuits also lead to significant improvements over conventional logic design when implementing iterative logic arrays. Such circuits show enhanced performance when implementing arithmetic functions such as adders, comparators etc [SUZ93], [YAN90].

A NMOS pass transistor network, shown in Figure 2.1, consists of an interconnection of NMOS transistors. Instead of simply passing a value of 0 of 1, these networks are capable of passing any value from the set [0, 1, X_i, $\overline{X}_i$, Z] where X_i is any input variable and Z is the high impedance state. The signals which drive the gates of the transistors in the network are called *control signals* and the signals which are passed through the transistors are called *pass signals*. The associated variables are called *control variables* and *pass variables* respectively.

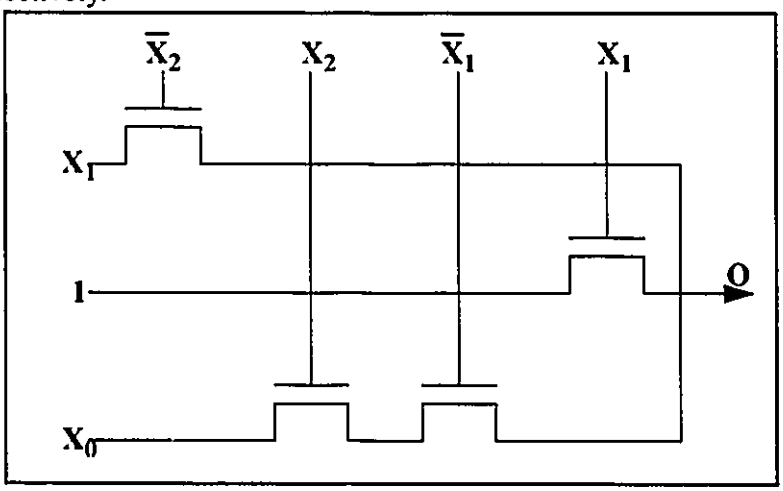

Figure 2.1 : PTL Network for the function $O = \overline{X}_2(X_1) + X_1(1) + X_2\overline{X}_1(X_0)$.

Each row, in the network, defines a product term P_i, which when enabled, allows the signal Y_i to reach the output. P_i is enabled if all the transistors in the i^{th} row are enabled i.e. all the control signals associated with that row are at logic level 1. If there is a transistor with input X_i ($\overline{X}_i$) at its gate, then X_i ($\overline{X}_i$) is a literal in the product term P_i. The output of one pass transistor feeds the input of the next one in a given row. The final output, **O,** is a wired OR of the outputs from each enabled row. If two or more rows are enabled simultaneously, the passed variables must all be at the same logic level. This ensures that there are no V_{DD}-to-ground conflicts. The output of a pass transistor network can thus be described by the equation

$\mathbf{O} = P_1 (Y_1) + P_2 (Y_2) + ... + P_n (Y_n)$ where $P_i (Y_i)$ means if P_i is enabled then Y_i is passed to the output [PAS91].

Conventional PTL networks may have speed degradation and other problems for larger functions. For example, with too many transistors in series, the speed of a pass-transistor network may become less than that of a circuit designed using conventional logic gates. Another disadvantage of a NMOS pass-transistor network is that it has poor low-to-high transition characteristics. The differential pass transistor logic (DPTL) techniques suggested in [PAS85] and [PAS93] and use of restoring buffers at regular intervals [ALA91] as well as other techniques [CHA92] have been proposed for such problems.

2.2.2 PTL synthesis Techniques:

Even though a number of applications of pass-transistor logic networks are available [GHO94], [KAN94], [MIT92], [PAS89], [PAS91b], [PAS91c], [SAL93], [SUZ93], [YAN90], there are only a few generalized synthesis techniques for pass-transistor logic design [PED88], [RAD85]. Radhakrishnan et al. [RAD85] introduced a two-level minimization technique, for pass-transistor logic, similar to two-level minimization of Boolean functions using Karnaugh maps and a tabular method which is a modification of the Quine-McCluskey approach [QUI55]. Pedron [PED88] described a synthesis

procedure for implementing a Boolean function as a disjunctive net. PTL can also be used for synthesizing sequential circuits [WHI92].

In a traditional K-map only the states which have an output of logic 1(0) are taken. The Karnaugh map method for PTL is similar to the conventional method except that a variable must be passed for every state for which an output has been defined. More than one variable may be passed through the network as long as it does not result in a V_{DD}-to-ground conflict. Finally, if there are don't care states, it must be ensured that it is not taken as 0 in one group and 1 in another.

In the modified Quine-McCluskey method, both 1s and 0s are considered when the prime implicants are formed, as opposed to only 1s (or 0s) in the traditional approach. Each implicant in the table is represented by three fields[Rad85]. The first two, the *base field* and the *difference field* are similar to the corresponding fields in the traditional approach. The third, *pass field*, represents the pass variable that is to be passed by the implicant. It may be 0, 1, X_i or $\overline{X}_i$. Two implicants can combine if

(a) The base fields differ in only one binary bit

(b) The difference fields agree and

(c) The pass fields are both constant or are identical.

The actual synthesis procedure is carried out in a manner identical to the Quine-McCluskey method, where new implicants are formed by combining old ones until no more implicants may be combined. The remaining uncombined entries are the prime pass implicants of the function from which a suitable cover must be chosen. As in the case with K-maps, care should be taken to ensure that don't care terms are not included in the final cover as both 1 and 0.

For example, consider the function f(a,b,c) = m(2,3,5,6,7). The initial eight pass implicants of this function are shown below in ascending order of their index.

Table 2.1. Table of Minterms

Truth Table			Output	Base(Difference)Pass	
X_2	X_1	X_0	f	List 1	List 2
0	0	0	0	0(1)0	$0(1,2)X_1$*
				$0(2)X_1$	$0(2,4)X_1$*
				0(4)0	
0	0	1	0	$1(2)X_1$	2(1,4)1*
				$1(4)X_2$*	
0	1	0	1	2(1)1	
				2(4)1	
1	0	0	0	$4(1)X_0$*	
				$4(2)X_1$	
0	1	1	1	3(4)1	
1	0	1	1	5(2)1*	
1	1	0	1	6(1)1	
1	1	1	1		

Combining 000 with 001 gives 00_ with pass variable 0. This is represented in terms of the three fields base(difference)pass as 0(1)0. Similarly, combining 000 with 010 gives $0(2)X_1$. Proceeding with this process, generates the prime pass implicants (marked by *) as shown in Table 2.1. [RAD85] gives a more detailed explanation of this process. The prime implicant table is shown in Table 2.2. and is used to pick a suitable cover. A suitable cover for f is $\{0(1,2)X_2, 2(1,4)1, 4(1)X_0\}$ which results in $f = \overline{X}_2(X_1) + X_1(1) + X_2\overline{X}_1(X_0)$.

Table 2.2. Prime Implicant Table

Prime Implicants	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7
$0(1,2)X_1$	*	*	*	*				
$0(2,4)X_1$	*		*		*		*	

Prime Implicants	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7
2(1,4)1			*	*			*	*
$1(4)X_2$		*				*		
$4(1)X_0$					*	*		
5(2)1						*		*

Another interesting method for analyzing and synthesizing a wide range of CMOS pass transistor circuits is given in [PED88]. The synthesis procedure, Pass Variable Optimal Synthesizer (PAVOS), implements a Boolean function as *disjunctive net* i.e. a circuit where the primary branches have no transistors in common. Similar to [RAD85], it also generates the pass prime implicants (i.e. prime implicants with associated pass variables), of the given function and then tries to extract a minimal cover. It does not take into account all types of *don't care states*, but only a subset called *don't happen states*.

2.2.3 Heuristic for Finding Minimal Cover

There are a number of well-known methods for selecting a minimum cover from the set of all prime implicants. In this section we will describe one such approach, which we also use in our algorithms. This is a heuristic for solving the set covering problem and is taken from [HOR78]. Even though this heuristic does not guarantee an optimal solution, we have used it for its simplicity.

We are given a family S of m sets S_i, $1 \leq i \leq m$. We denote by |A| the size of set A. Our problem is to find a subset $T = \{T_1, T_2, \ldots, T_k\}$ of S such that

- for each i, $1 \leq i \leq k$, $T_i = S_r$, for some r, $1 \leq r \leq m$
- $\bigcup T_i = \bigcup S_r$
- |T| is as small as possible

The heuristic selects set $T_1, T_2, \ldots T_p, \ldots T_k$. When selecting Tp, a set $\Theta = T_1 \cup T_2 \cup \ldots \cup T_{p-1}$ is available. The heuristic for selecting T_p is given below.

1. If $\Theta = \bigcup S_r$ no further selection is needed

2. If there is any set S_j containing an element which no other set contains, select S_j and replace Θ by $\Theta \cup S_j$

3. Otherwise, select a set S_j that contains the largest number of elements not already in Θ and replace Θ by $\Theta \cup S_j$

For 2-level PTL synthesis problem, the i^{th} prime pass implicant represents a set of minterms and defines set S_i.

2.3 Multilevel Logic Synthesis

2.3.1 What is Multilevel Logic Synthesis

The decomposition of Boolean functions and synthesis of logic networks for realizing these functions have been of interest for several decades. [ELL65] introduced a combinational logic synthesis procedure with upto three logic levels. The method proposed by Ashenhurst {ASH59] for decomposing logic functions using decomposition charts has been used extensively for obtaining multistage realizations of Boolean functions [SHE70]. Other earlier approaches to multilevel logic design include those based on transforms [LEC70], [MEN69], variational approach [HAC67] and universal logic modules (ULMs) [YAU70]. Synthesis of multi-output combinational logic circuits using a hierarchy of 'goals' is discussed in [SCH68] and automated design of logic networks using factoring techniques are described in [DIE69].

Over the last ten years or so, there has been a great increase in research interest in this area of multilevel logic synthesis. Compared to traditional two-level minimization techniques, multi-level logic minimization offers much more flexibility in the design process. This means that there is a potential for obtaining more area efficient logic circuits than before [BRA90]. The process of factorization takes a two-level sum-of-products (sop) representation of a function and generates a factored form which is essentially a multi-level AND-OR representation of the function. The greater degrees of freedom in multi-level design, however, also make the problem of finding an optimum realization much more difficult to solve. Multilevel minimization techniques, therefore, tend to rely on heuristics that try to minimize the number of literals. Literal count is the primary cost function in most of these algorithms. There are a number of well established algorithms for carrying out factorization of Boolean expressions and forming multi-level AND-OR networks [BRA87], [BRA88], [MCM84]. These techniques have been used successfully for synthesizing VLSI systems [RUD89]. Such multi-level, factorized representations can also be readily implemented as a series-parallel network of switches. In terms of VLSI, a switch may be realized by a MOS transistor.

2.3.2 Basic Concepts of Factorization

In this section we will introduce some basic definitions related to factorization of Boolean expressions taken from [BRA82], [BRA84]. We will also discuss some algorithms for decomposing a given two-level function into a series-parallel (AND-OR) network. We will include in this section the terms and concepts which we will use in this thesis. In our discussions we have used +(*)to denote the operation OR (AND).

Definition 2.1 : A factored form is a representation of a logic function that is either a single literal or a sum or product of factored forms. According to this definition, a sum-of-products representation is just a special case of a factored form.

Definition 2.2: A logic function **g** *is a Boolean divisor of* **f** *if* **f = g*h + r**, *where* **h** *and* **r** *are logic functions and* **g*h** *is not equal to 0. Similarly,* **g** *is a Boolean factor of* **f** *if* **f=g*h**. *Thus a divisor is a factor of a subset of* **f**

Definition 2.3: The product **f*g** *of two Boolean functions,* **f** *and* **g**, *is called an algebraic product if* **f** *and* **g** *are orthogonal (i.e. they have disjoint sets of support); Otherwise,* **f*g** *is a Boolean product.*

Definition2.4: An operation / is called division, if given two functions **f** *and* **p**, *it generates* **q** *and* **r** ***(f/p) = <q,r>*** *such that* **f = p*q + r.**

If **p*q** is an algebraic product the operation / is called algebraic division; Otherwise it is called Boolean division. In the above definition, **p, q** and **r** are called the divisor, quotient and remainder respectively.

For example, the function **F1 = abg + acg + adf +aef + afg + bd + be + cd + ce**, after algebraic division, reduces to **F1 = (af + b + c)*(d + e) + ag*(f + b + c)**.

For AND/OR realizations, the literal count obtained by counting the number of occurrences of literals in a function F has been used successfully to estimate the cost of the circuit [WAN89]. For instance, the original SOP representation of F1 has a literal count of 23 while the multilevel factored form has only 11 literals. Following standard conventions, from now on, we will omit the AND operation * when it is obvious.

Using the notion of division for factorization requires two main tasks. The first is to find 'good' candidate divisors and the second is to actually carry out the division. The concept of kernels becomes important in the context of choosing divisors since they allow efficient identification of common subexpressions. Thus kernels turn out to be very good candidate divisors. The following definition of a kernel is taken from [WAN89].

Definition 2.5: A kernel of an expression f is defined by the following two rules:

1. A kernel ***k*** *of an expression f is the quotient of f and a cube* ***c*****;** ***k = f/c.***

2. A kernel ***k*** *is cube-free (****k*** *cannot be written as* ***dg****, where* ***d*** *is a non-trivial cube and* ***g*** *is an expression.*

A cube is a conjunction of literals, e.g. **ab**, **bdf**. The cube ***c***, in the above definition, is the *co-kernel* associated with the kernel ***k***.

As an example, we consider the following function represented in sum-of-products form : **F = abdf + bcdf + abh + bch + fg + dgh.** Then, **F/a = bdf + bh** is the quotient of **F** and the cube **a**. But, it is not a kernel since it can be written as **b(df + h)** and hence is not cube-free. However, **F/ab = df + h** is a kernel of **F** since it is the quotient of **F** and the cube **ab** and is cube-free. Using the kernel **(df + h)** as a candidate divisor and performing an algebraic division, **F/(df + h)** generates the quotient **q = ab + bc = b(a + c)** and the remainder **r = fg + gdh = g(f + dh)**. This leads to the final multilevel representation of **F** as **F = b(a + c)(df + h) + g(f + dh)**.

It is immediately evident that the multilevel form offers much more savings with literal count of only ten; whereas, the SOP form requires nineteen literals.

2.3.3 Factorization Algorithm

The generic factoring algorithm proposed by Brayton [BRA87b]is given below.

```
gfactor(F):
If ( | F | = 1) return F
K = choose_divisor(F)
(Q, R) = divide(F, K)
return ( gfactor(K) * gfactor(Q) + gfactor(R) )
```

This is a top-down, recursive algorithm based on the heuristic of reducing literals at each stage. The first step checks if the expression **F** can be factored. If not, the original function is returned as the answer. Otherwise, a candidate divisor **K** is chosen from among the possible ones i.e. from the set of kernels of **F**. In the third step the actual division algorithm is carried out in order to generate the quotient **Q** and the remainder **R**. Thus the original expression is broken down as **F = KQ + R**. In the fourth and final step each component (i.e. **K, Q** and **R**) is factorized by calling gfactor recursively with each component as an argument. Variations of this algorithm include *quick_factor* and *best_factor*. Different investigators have looked at a number of ways to choose the divisor [CAR91], [MAL89], [HSU92].

It is important to note that this class of algorithms represents a pragmatic way of reducing literal count since we choose a multilevel realization rather than two-level realization only when the literal count is reduced. This is a recursive algorithm and the decision to replace a two-level sum-of-products expression by a multilevel expression (when we see the potential for savings in terms of literal count) may be conveniently viewed as a greedy heuristic.

2.4 Binary Decision Diagrams

2.4.1 What is a BDD

Binary Decision Programs (BDP's) [KUZ77], [LEE59] depict Boolean functions using multilevel two way branchings. BDP's consist of a sequence of a statements of the form **{label}:{variable}, {label1}, {label2}** [CHA93]. If the variable being tested has the value **true** then the program branches to **{label1}**, else it branches to **{label2}**. There are two special **halt** instructions labelled by **Z** and **I**. If the program halts at **Z**, it outputs 0, else it outputs 1. ***Binary Decision Diagrams*** (BDD's) [AKE78] are labelled, directed graphs representing BDP's. For every instruction I_0 in the BDP there exists a node N in the BDD. If **{label1}** (**{label2**) of I_0 is I_1 (I_2), then there exists an edge from N to the node for the instruction $I_1(I_2)$ labelled by 1(0). The two nodes corresponding to the two halt instructions **Z** and **I** are the leaf nodes and are labelled by 0 and 1, respectively. We consider only acyclic BDD's. The following definitions are from [CHA93].

*Definition 2.6: Given a BDD **B**, path **p** in **B** and a variable X_i, X_i is said to **occur k times** in **p** if and only if there are **k** nodes in **p** labelled by X_i.*

*Definition 2.7: A variable that occurs more than once along a path is a **Repeated Variable**.*

*Definition 2.8: A **Free** BDD **B** is a BDD which contains no **Repeated Variable**. A BDD that is not free is said to be a **Repeated** BDD.*

Given a BDD B, we define a binary relation $\leq_B$ on the set of input variables as follows: for all $i \neq j$, $X_i \leq_B X_j$ if and only if there exists a path **p** in B such that X_i occurs before X_j in **p**.

Definition 2.9: A BDD B is an ***Ordered*** *BDD if and only if B is a free BDD and* $\leq_B$ *is a partial order.*

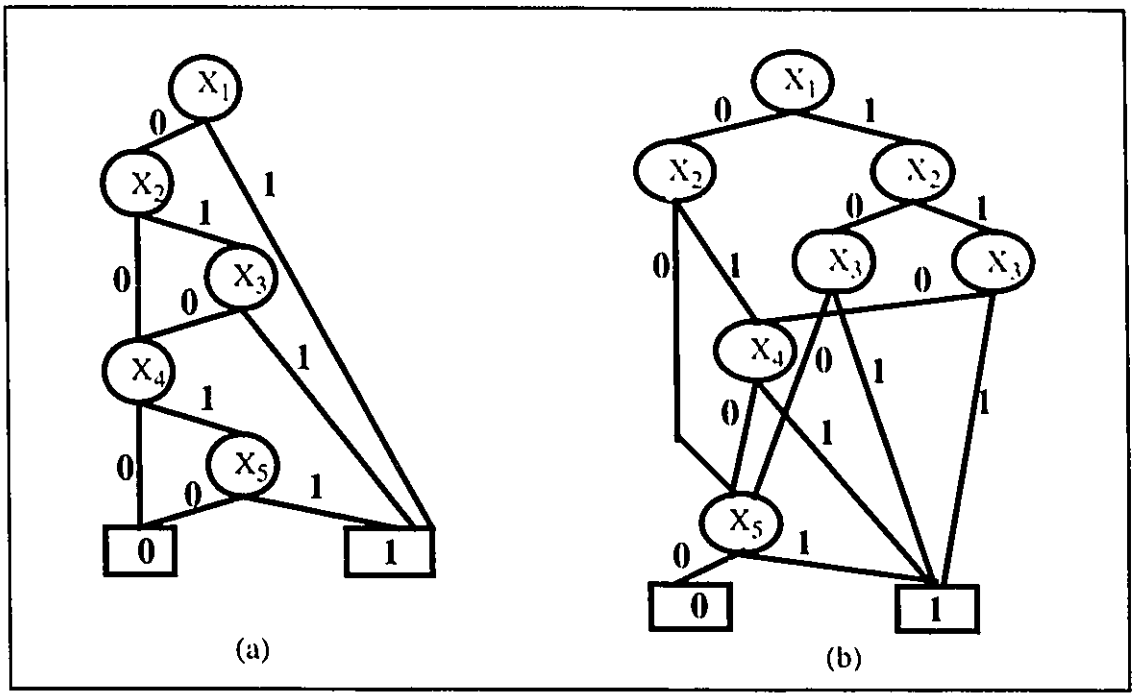

Figure 2.2 : OBDDs representing the function F = a + bc + de.

Ordered BDD's (OBDD's) are being used for synthesizing and analyzing Combinational as well as Sequential Circuits [ABA86], [ASH91], [BER89], [BUR90], [CHA89], [MAT89]. One of the most important factors determining the size of an ordered binary decision diagram (OBDD) representing a particular Boolean function is the ordering of the input variables[FRI90]. For example, Figures 2.2 (a) and (b) show two OBDD's for the function F = a + bc + de, variable orderings X_1=a; X_2=b; X_3=c; X_4=d; X_5=e and X_1=b; X_2=d; X_3=c; X_4=e; X_5=a respectively. Obviously, the first is a much better choice.

2.4.2 Important Definitions and Properties of OBDDs

According to [BRY86] an OBDD is represented as follows.

- It is a rooted, directed graph containing two types of vertices
- A nonterminal vertex, denoted by a circle, has an indexed argument X_i, $1 \leq i \leq n$, associated with it
- The index of a nonterminal vertex must be greater than that of its parent
- A nonterminal vertex has exactly two children connected to it through two distinct edges labelled 0 and 1
- A terminal vertex, denoted by a square, has a value 0 or 1

*Definition 2.10: The **child** of a node N, at the i^{th} level, is any node at level j (j > i) which is connected to N through a single edge labelled 0 or 1.*

*Definition 2.11: An edge with label 0(1) from a node with label X_i is said to be **activated** if X_i is 0(1). A path in an OBDD is **activated** if all edges in the path are activated.*

A subgraph rooted at vertex **v**, with index **i**, can be considered to denote a specific Boolean function $\mathbf{f_v}$ of variables X_i, X_{i+1}, ... X_n. This function ($\mathbf{f_v}$) can be represented in terms of a bitmap. We will associate the bitmap representation of $\mathbf{f_v}$ with the node **v**. The bitmap associated with a terminal node having value 0(1) is 0(1).

Property 2.1: The function associated with a nonterminal node in an OBDD can always be determined from the label of the node and the functions associated with its children.

Property 2.2 : For a given input combination, there is always exactly one activated path from the root node to a terminal node in an OBDD.

Definition 2.12: Given two bitmaps B1 and B2 of length m, the operation B1 • B2= B creates a new bitmap B of length 2m by concatenating B2 to the end of B1. For example, let B1 = 0110, B2 = 1101 then B = B1 • B2 = 01101101.

2.4.3 How to Construct a BDD

In [BRY86] Bryant describes a procedure for obtaining a reduced OBDD from an arbitrary OBDD representing the same function. This is a bottom-up procedure starting from the terminal nodes up to the root. A reduced OBDD, for a given Boolean expression (which may be represented by a a bitmap), can also be readily constructed in a top-down manner. In this section, we will give a brief outline of how this can be done. Since our algorithm for synthesizing 123-DDs is top-down, this will help in comparisons later on.

Transformation 2.1: Expanding a nonterminal node N with a label X_i and an associated bitmap B of length 2m, generates two children $N_1(N_2)$ of N. $N_1(N_2)$ has label X_{i+1}, has an associated bitmap B1(B2) of length m, and is connected to N through an edge labelled 0(1), such that $B1 \bullet B2 = B$.

Transformation 2.2: If the two children of a node (N) have identical bitmaps associated with them, then the two children may be merged to form a single node and the node N can be deleted.

Transformation 2.3: If two (or more) nodes at the current level have identical bitmaps, then the subgraphs below them can be merged. This is done by expanding only one of the nodes and connecting the others to it.

Transformation 2.4: If the bitmap associated with a node N consists of all 0's (all 1's) then convert N to a terminal node with value 0(1).

The algorithm for constructing a reduced OBDD for a completely specified function $F(X_1, X_2, \ldots, X_n)$, is given below. F is defined in the form of a bit map of size 2^n. The procedure starts with a single node representing the entire function at level 1. The ordering of the inputs is $X_i < X_j$ if $i < j$. The bitmap is such that $X_1(X_n)$ represents the MSB(LSB).

1. Current level = 1
2. While current_level ≤ n do
   a. Expand each nonterminal node at the current level using Transformation 2.1.
   b. Current level = Current level + 1
   c. Apply transformation 2.2 and delete appropriate nodes at the previous level.
   d. Apply transformation 2.4 to appropriate nodes at the current level.
   e. Apply Transformation 2.3 to remaining nodes at the current level

For a completely specified function, any function can be reduced to a unique canonical representation using OBDDs [BRY86]. The above algorithm can be extended to include incompletely specified functions as well. However, there is no guarantee of a minimal canonical form in this case. Here, we will briefly state some basic properties and definitions regarding incompletely specified functions which will be used in the following chapters.

Definition 2.13: An incompletely specified function of m bits, can be represented by a bitmap of length 2^m *such that for each bit* b_i, $1 \leq i \leq 2^m$ $b_i \in \{0, 1, d\}$, *where d represents a don't care state.*

Definition 2.14: Two m-variable functions F1 and F2, with bitmaps b1 and b2, are ***equivalent*** ($F1 \equiv F2$) *if, for all i,* $1 \leq i \leq 2^m$ *either (i)* $b1_i = b2_i$ *or (ii)* $b1_i = d$ *and* $b2_i \in \{0, 1\}$ *or (iii)* $b2_i = d$ *and* $b1_i \in \{0, 1\}$.

Definition 2.15: Two ***equivalent*** *m-variable functions F1 and F2, with bitmaps b1 and b2, can be combined to form a new function F3, with bitmap b3. The new bitmap b3 is determined as follows.*

For all i, $1 \leq i \leq 2^m$, IF $b1_i = d$ THEN $b3_i = b2_i$; ELSE $b3_i = b1_i$. Henceforth we will denote this operation as $(b1, b2) \Rightarrow b3$.

2.5 Conclusions

In this chapter we have reviewed the current state of PTL design techniques. Formal methodologies for PTL networks are generally limited to two-level implementations. In the remainder of the thesis we will develop a number of multi-level PTL synthesis techniques which show considerable improvement over existing techniques.

Chapter3

Factorization for Pass Transistor Logic

3.1 Introduction

This chapter discusses a technique of using multilevel logic synthesis to design pass transistor logic (PTL) based on algebraic factorization [BRA84]. We show that techniques already applied to conventional AND-OR type networks are not useful for factorization of PTL networks. Our synthesis technique is a greedy heuristic which we show is capable of synthesizing PTL arrays with an average of 14.02% savings over conventional two-level design techniques.

Although algebraic factorization techniques have been discussed in the literature for conventional logic synthesis, such existing techniques cannot be directly applied to PTL for the following reasons:

(i) If we do not allow repeating of input variables, then all factors must maintain valid input ordering.

(ii) Literal count is not an accurate indicator of the cost of implementing a PTL circuit.

(iii) Conventional factorization techniques start from a minimized sum-of-products (SOP) representation of a function; this does not lead to significant improvements in case of PTL.

In addition, as mentioned in [PAS91], ad hoc design of PTL networks can lead to short circuits (see section 3.2.4). Since we use the set of all prime pass implicants as the starting point for our design, we can easily avoid this problem.

In the algorithm presented in this chapter, we take account of the above factors by starting with the set of all prime pass implicants [RAD85] and integrating the steps of selecting a cover and factorizing a function. From many examples that we have tried, using MCNC benchmark circuits, our algorithm achieves considerable improvement over PTL circuits obtained from conventional two-level design methods.

3.2 Why Factorization Algorithm Must be Modified for PTL

An example of a multilevel PTL network is shown in Figure 3.1. In a 2-level PTL network, the basic building block is a column of transistors. In a multilevel PTL network, the basic building block is

- a 2-level PTL network or
- an array of transistors, A, where the signal to the bottom transistor in each column is generated by another multilevel PTL network, B

For example, in Figure 3.1, the array of transistors A is demarcated by dotted line DL_A and the PTL network B is demarcated by dotted line DL_B. The signal to the bottom transistor in each column in A is generated by B.

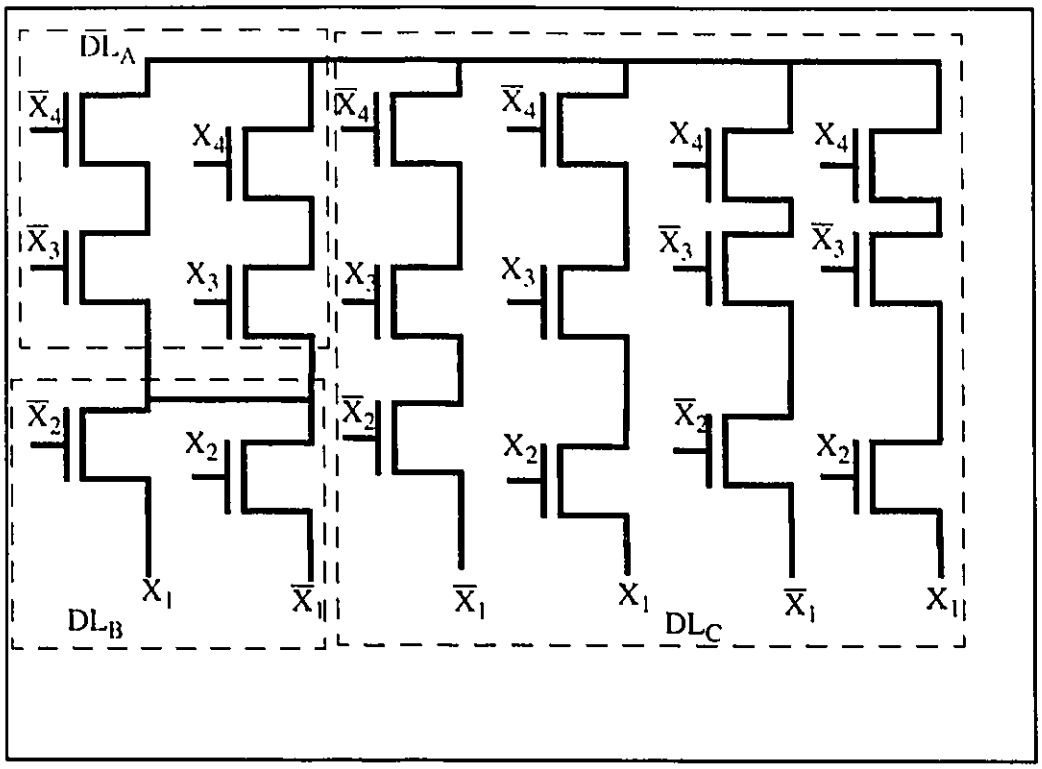

Figure 3.1 : A Multilevel PTL Network

Our task is to identify possible multilevel building blocks so that the number of columns of transistors needed to realize any arbitrary function is as small as possible.

The problem of synthesizing multilevel PTL may be tackled in a variety of ways. In this chapter, we have used the following simplifying assumptions :

Assumption (i) for a given column of transistors, the same input is never used more than once as a control variable. However, an input may be used as a control variable in one or more columns and/or as a pass variable in one or more of the remaining columns.

Assumption (ii) the ordering of input variables does not change as we go from one column to another.

Assumption (iii) the multilevel PTL network B is a 2-level network with only 2 columns.

Assumption (i) allows us to use algebraic techniques and ensures that the number of rows in our approach is identical to that in 2-level synthesis. Assumption (ii) ensures a simple scheme for applying the inputs to the gates of transistors in different columns. Assumption (iii) is included since it keeps our problem tractable and, because our experiments indicated that the overwhelming majority of PTL divisions involve bicubic divisors.

3.2.1 Restrictions on Ordering of Input Variables

When we have an algebraic factorization, $\mathbf{f = q*d + r}$, the expression corresponding to $\mathbf{q(d)}$ has to be realized by network A(B) or by B(A). We adopt the convention that the array of transistors A realizes **q** and the 2-level PTL network B realizes the divisor **d**. For instance, in Figure 3.1, **q** is realized by the network shown by dotted line DL_A, **d** by dotted line DL_B, **r** by dotted line DL_C. It is important to note that when we have an algebraic factorization $\mathbf{f = q*d + r}$, none of the inputs appearing in $\mathbf{q(d)}$ can appear in between two inputs appearing in $\mathbf{d(q)}$. Let L_q and L_d denote the set of literals appearing in the cubes of **q** and **d** respectively. The ordering of variables in the PTL network corresponding to **f** should be such that all literals in L_d have a position below that of all literals in L_q.

In general, after getting the factored form $\mathbf{f = q*d + r}$, the remainder **r** has to be factorized in a similar manner so that, when the process is over, we obtain $\mathbf{f = q_1*d_1 + q_2*d_2 + ... + q_s*d_s + r_s}$ where $\mathbf{r_s}$ cannot be factorized any further. It is important that the restrictions in ordering of variables do not lead to a conflict.

Example 3.1

The function $\mathbf{f = ac(0) + acf(0) + bcd(0) + bdef(0) + abg(h) + ade(h) + bcg(h) + cde(h) + R}$ (where **R** represents the remaining prime pass implicants) may be factorized as $\mathbf{f = (a + bd)*[c(0) + ef(0)] + (bg + de)*[a(h) + c(h)] + R}$. If we implement this factorization using PTL, we do not satisfy our assumptions (i) and (ii) due to the following reasons:

- **(a + bd)*[c(0) + ef(0)]** implies that inputs **a, b** and **d** occur above **c, e and f.**
- **(bg +de)*[a(h) + c(h)]** implies that the position of inputs **b, d, g and e** must be above **a** and **c**.

Thus the input signal **a** must occur both above and below the signal **e**. These two conditions are contradictory. This is shown in Figure 3.2.

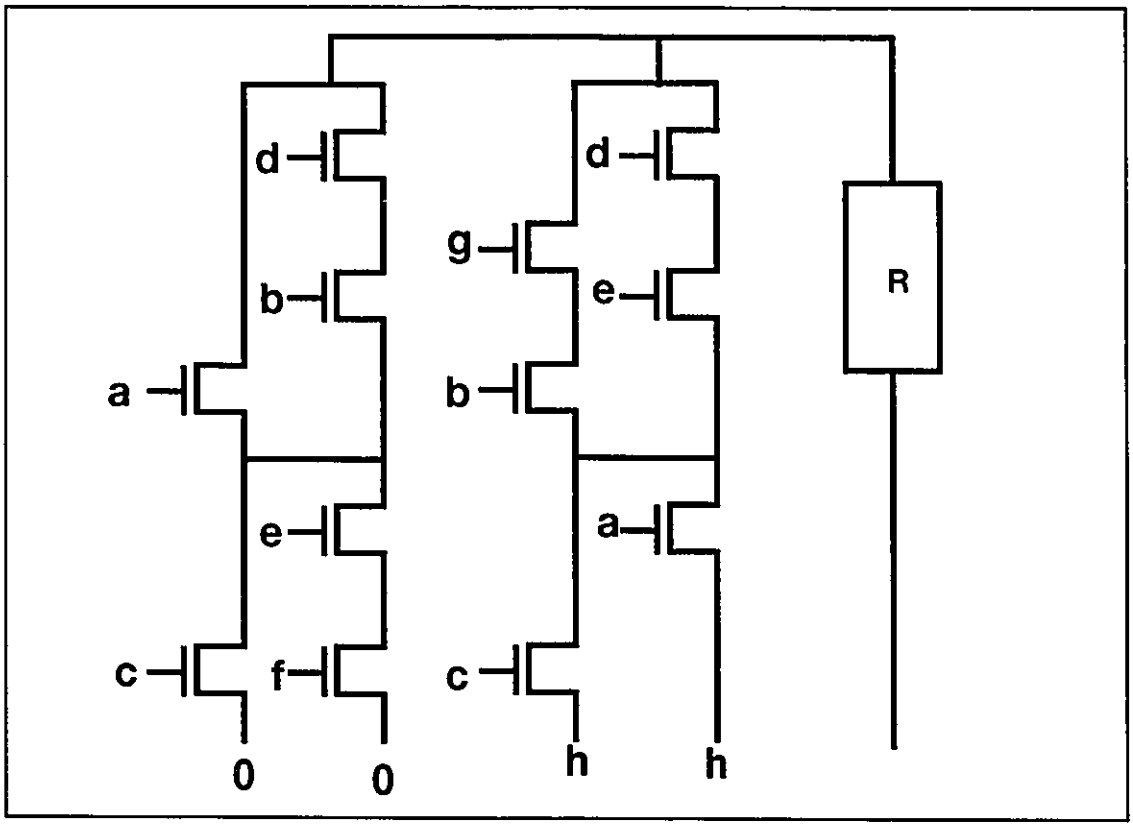

Figure 3.2 : Restrictions on input ordering

3.2.2 Need for a More Realistic Cost Estimator

The cost function used to measure the savings obtained by factorization must be modified to reflect the implementation strategy. In conventional factorization, the literal count is taken as a reasonable measure of the area occupied by the circuit [WAN89]. However for PTL networks, area, measured by the product of number of rows and number of columns,

is of critical concern. In our approach, the number of rows is determined by the number of input variables and therefore does not change when we factorize. Thus the cost is determined by the number of columns needed to implement a given function. A factorization which does not reduce the number of columns leads to zero savings, even if the literal count is reduced. One consequence is that a factorization which yields a single cube as a quotient is useless so far as reducing the number of columns is concerned. This has serious repercussions for PTL factorization heuristics since kernels are usually considered good candidate divisors. The rationale for this is simple for AND-OR networks; we are guaranteed that the co-kernel [BRA87b] will be in the quotient, and even if the quotient contains nothing more, we are assured of savings in terms of literal count.

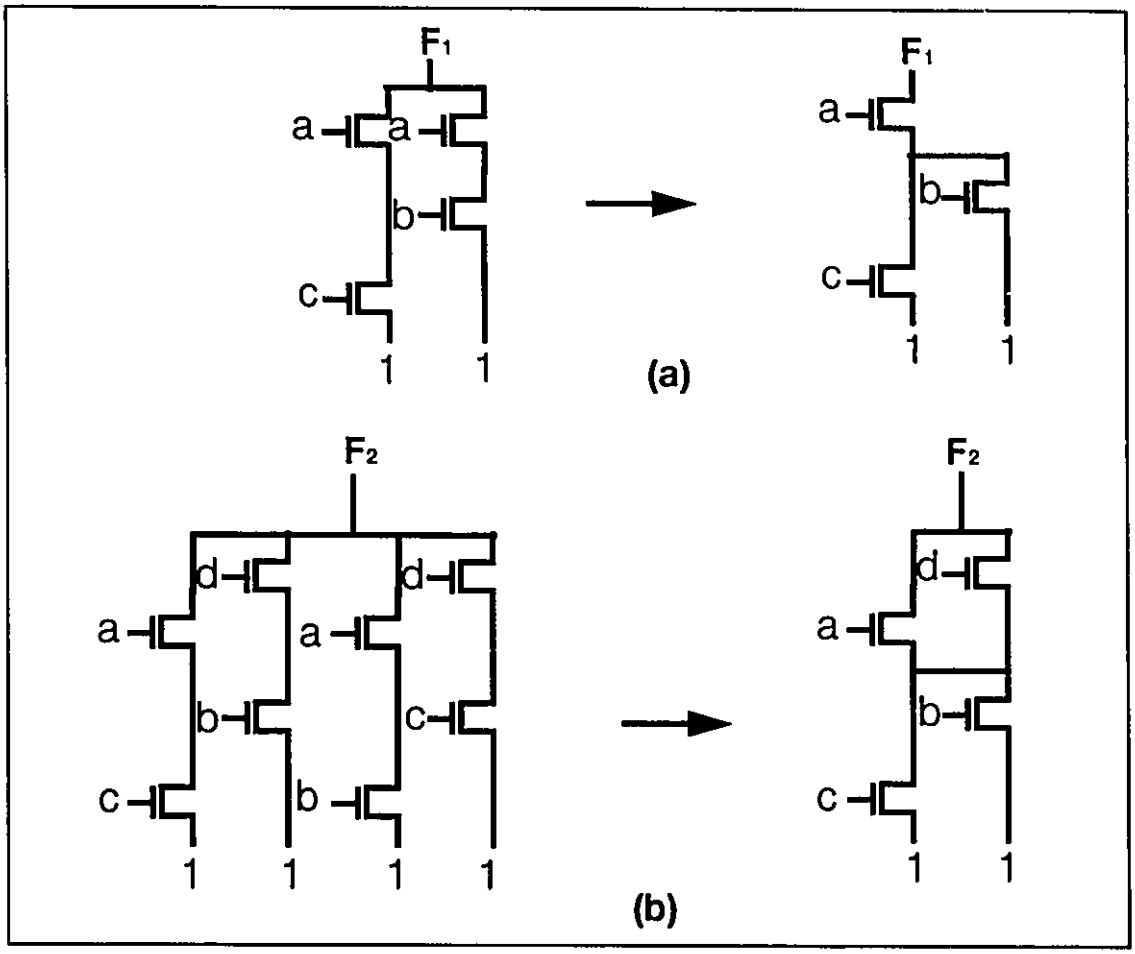

Figure 3.3 : Examples illustrating area reduction and factorization.

For example, the function **F1 = ab(1) + ac(1)** may be factorized as **F1 = a*[b(1) + c(1)].** This reduces the literal count from four to three. However the number of columns needed to implement it is still two (Figure 3.3(a)). The function **F2 = ab(1) + ac(1) + bd(1) + cd(1)**, on the other hand can be factorized as **F2 = (a + d) *[b(1) + c(1)]**. In this case, the literal count and the number of columns required to implement the function are both reduced (Figure 3.3(b)). Our algorithm selects only those factored forms which reduce both literal count and number of columns.

3.2.3 Two-level Minimal Sum-of-Products Covers Do Not Lead to Significant Savings

If we consider two-level covers as obtained by [RAD85] as our starting point and apply heuristics analogous to *quick-factor* and *best-factor* [WAN89] to this two-level cover, taking into account the restrictions mentioned above, the savings in terms of reduction in the number of columns turn out to be insignificant. Our analysis indicates that

(i) Many divisions using kernels as divisors give single cube quotients. As discussed in 3.2.2, such divisions are useless.
(ii) After picking up a few factorizations, the ordering restrictions discussed in 3.2.1 above disallowed further factorization.

Our experience is that an adaptation of *best-factor* gives improvements of the order of 0 - 5% even for relatively large functions (8 -12 inputs)

3.2.4 Restrictions Due to the Bidirectional Nature of MOS Circuits

Another point, which must be considered, is that, unless a PTL network is designed carefully, there is a possibility that short circuits might occur.

Example 3.2

We consider the function **F**, implemented as shown below (Figure 3.4). The condition **b =1** and **c = 1**, creates a short circuit.

In our algorithm this type of short circuit can never occur, since we only use prime pass implicants for factorization. For completely specified functions, we can never have two prime pass implicants (P_1, P_2) which simultaneously allow both 0 and 1 to be transmitted to the output (provided the prime pass implicants are constructed correctly). Therefore, a kernel formed from P_1 and P_2 can never create a short circuit.

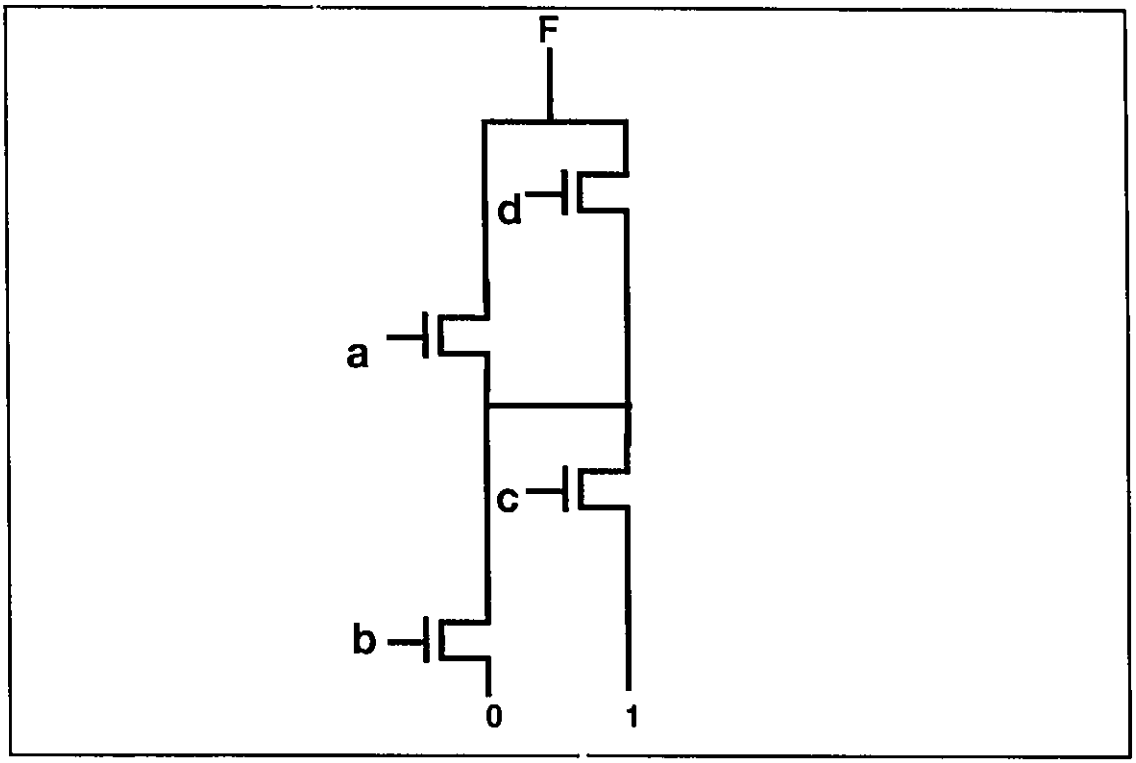

Figure 3.4 : Invalid PTL network leading to a short circuit

For incompletely specified functions, there may be two prime pass implicants (P_1, P_2) which cover the same don't care minterm, and one (say P_1) considers it to be 0 while the other (P_2) considers it to be 1. If both P_1 and P_2 are selected as part of the final cover, then there will be a short circuit. In our factorization algorithm, whenever a prime pass implicant P_i, covering a don't care minterm and treating it as a 0(1), is selected all other prime pass implicants treating that minterm as 1(0) are immediately removed from the list of prime pass implicants, and thus eliminated from further consideration. So, even for

incompletely specified functions, our algorithm guarantees that there will be no short circuits.

3.3 Outline of Our Algorithm

3.3.1 Some Definitions

*Definition 3.1: Given the prime pass implicants of any function F, a factored form FF is a pair (**q**, **d**) where **q** is a sum of cubes and **d** is the sum of two pass implicants such that*

- the set of literals appearing in **q** is disjoint from the set of literals appearing in **d** and
- all the pass implicants obtained by the product **q*d** are prime pass implicants of F

If the pair (***q***, ***d***) for a factored form FF contains m_1 cubes and m_2 cubes respectively, then the number of columns to realize the factored form FF using PTL is the larger of m_1 and m_2.

We have modified the set covering heuristic described in section 2.2.3 by including a greedy method for selecting, if possible, factored forms satisfying the restrictions mentioned earlier. The selection of each factored form imposes additional restrictions on ordering of input variables when selecting subsequent factored forms, so that the sequence of picking factored forms is important in determining which factored forms satisfy the restrictions on ordering of inputs as discussed in section 3.2.1.

In the description of our algorithm given below, we assume we have already selected factored forms FF_1, FF_2, ... , FF_{p-1} and we are now selecting the pair (***q***, ***d***) for the next factored form FF_p.

Definition 3.2: The factored form FF_p will be called ***ordering compatible factored form*** *if the ordering restrictions imposed by $FF_1, FF_2, \ldots, FF_{p-1}$ are compatible with restrictions implied by the factorization* ***q*d*** *for FF_p as outlined in 3.2.1.*

Definition 3.3: During the process of selecting prime pass implicants using the heuristic given above, if a prime pass implicant P_i covers m minterms which are not in Θ, then the ***score of prime pass implicant*** *P_i is m.*

Definition 3.4: If a factored form FF_i covers m minterms which are not in Θ, then the ***score of factored form*** *FF_i is m/n where n is the number of columns of transistors needed to realize FF_i using PTL.*

3.3.2 Our Heuristic

In this section we give an outline of our heuristic for factorizing a Boolean function F. In step 1, we generate the set of all prime pass implicants for a function using the procedure described in section 2.2.2. Next we initialize the set of covered minterms before starting the actual factorization process. Step 3 describes the process of factorizing the Boolean function and selecting a suitable cover for it. The details of this heuristic are described in Appendix A.

*1. **Generate prime pass implicants.***

Use the modified Quine-McCluskey method outlined by Radhakrishnan et al. in [RAD85], to generate all the prime pass implicants of the function

*2. **Set** $\Theta = \{\}$.*

Initialize the set of minterms of F covered so far, Θ, to an empty set.

*3. **While** ($\Theta \neq \bigcup S_r$) **repeat the following steps***

$\bigcup S_r$ is the set of all minterms of F. Step 3 should be repeated until all minterms in $\bigcup S_r$ are also covered by Θ

*a. **Select the next prime pass implicant P_i.***

Use the set covering heuristic given in section 2.2.3 to select the next prime implicant P_i, to be included in the cover Θ. P_i is the prime pass implicant covering the maximum number of minterms not already covered by Θ.

*b. **Try to select a suitable factored form FF_j***

- *Try to find a prime pass implicant P_j, which has at least one literal in common with P_i*
- *Generate a bi-cubic kernel, k, from P_i and P_j*
- *Divide F by k to generate a factored form FF_{ij}*
- *Check if FF_{ij} is ordering compatible with the existing factored forms*
- *Check if score of FF_{ij} > the score of P_i*

*c. **Select next term to be included in final cover.***

If step 3b is successful,

- *select FF_{ij}*
- *update Θ with all the minterms covered by the pass implicants of FF_{ij}*

Otherwise,

- *select P_i*

In step 3b, we use the function find_factored_form to try to obtain a factored form FF_{ij} which

- includes P_i
- is compatible with all the ordering restrictions imposed by previously selected factored forms $FF_1, FF_2, \ldots, FF_{j-1}$
- has a better score than P_i

To check whether a factorization ***q*d*** violates restriction 3.2.1, we use function check_ordering which has two parameters and returns true (false) if factorization ***q*d*** satisfies (violates) restriction 3.2.1. The two parameters of this function are described below :

- Parameter 1 is an N X N array of characters where N is the number of inputs. The entry in the $(i, j)^{th}$ cell of the array is '>' if there is a restriction, due to previously selected factored forms, that input i must have a position below input j
- Parameter 2 is also an N X N array of characters. The entry in the $(i, j)^{th}$ cell of the array is '>' if input i appears in divisor ***d*** and input j appears in quotient ***q***. This parameter represents the new restrictions due to the current factorization ***q*d***

check_ordering(array1, array2) ensures that wherever there is a '>' entry in array2, there is no contradictory entry in array 1. In other words, if cell (i,j) in array2 is '>', cell (j,i) in array1 must not be '>' since this indicates a contradiction. Details of this function are given in the appendix.

3.3.3 Justification of the Heuristic

In section 2.2.3 we described a heuristic for selecting a two-level cover for a Boolean function. According to this heuristic, the next prime implicant selected should be the one that covers the maximum number of minterms which are not covered by the prime

implicants selected so far. Each prime implicant in the cover can be realized as a single column in a PTL network. In our algorithm, we use a similar criterion for selecting the next term in the cover, which may be either a single prime implicant P_i or a factored form. The prime implicant P_i is the same one which would be chosen if we were selecting a SOP cover. A factored form is chosen in place of P_i only if the following conditions are satisfied.

- One of the prime implicants included in the factored form is P_i
- It satisfies the ordering restrictions in 3.2.1
- The score of the Factored Form is greater than that of P_i

This process is repeated until all the minterms are covered.

This heuristic intuitively makes sense due to the following reasons :

- P_i is known to be a "good" candidate so far as the number of uncovered minterms are concerned.
- We are looking at every possible prime pass implicant when determining a suitable factored form including P_i. We recall (section 3.2.3) that restrictions of PTL networks led to very little improvements when applied to sum-of-products cover. Our approach increases the chances of getting a factored form satisfying restriction 3.2.1 since we are looking at many potential factored forms. Our experiments with functions of up to 10 variables indicate that it is computationally feasible to look at all possible prime pass implicants for PTL networks of practical size.
- Our policy of using the score to evaluate a factored form ensures that the number of minterms covered by each column to realize the factored form is better than that of P_i .

3.3.4 Complexity of the Heuristic

In this section, we will estimate the worst case complexity of the factorization algorithm, by determining the complexity of each step after generating the set of all prime pass implicants. Let the total number of prime pass implicants for a function of n inputs be N. Each prime implicant is represented by two bit maps, one to denote the literals present in it, the second to denote which literals are complemented. We assume that each bitmap is small enough to fit into one word.

The complexity of step 3a, which selects the prime implicant P_i covering the maximum number of uncovered minterms is $O(N)$.

Step 3b has several components. Finding prime pass implicant P_j involves looking at all the prime pass implicants and has complexity $O(N)$. Determining the kernel, k, for the pair of prime pass implicants P_i and P_j involves a fixed number of bit manipulations so that its complexity is $O(1)$. Dividing a single pass implicant by another pass implicant is of complexity $O(1)$. Dividing a sum of N prime pass implicants, F, by a single pass implicant p involves dividing each prime pass implicant in F by p and its complexity is $O(N)$. Therefore, dividing F by the two cubes of the kernel has complexity $O(N)$. The final result is obtained by the taking the intersection of the two intermediate quotients. Each quotient may have at most (N-1) terms and each term in one quotient has to be compared with all the terms in the other quotient. So, the worst case complexity of the intersection is $O(N^2)$. The total complexity for performing the division is $O(1) + O(N) + O(N^2) = O(N^2)$.

The ordering information for the previously selected factored forms is kept in an nXn array with n^2 elements. In order to check if the current factored form FF is ordering compatible, we take each pair of inputs, one each from the quotient and divisor and insert the ordering relation between them into a new array. The elements of the first array is then compared with the new one and updated if there is no conflict. Performing this operation once for the entire array has complexity $O(n^3)$. In order to take the closure of the

relationships, this may have to be done at most n times. Therefore the complexity of checking ordering compatibility is $O(n^4)$.

Determining the score of a factored form, FF, involves taking the union of the minterms covered by each prime implicant in FF and dividing the total number of minterms covered by the number of rows required to implement FF. If the number of new minterms covered by each prime implicant is $O(m)$, then complexity of performing the union is $O(m^2)$. Therefore, the complexity of checking the score is $O(m^2)$.

So, the worst case complexity of step 3b is $O(N^3 + N.n^4 + N.m^2)$.

In step 3c, after selecting the next term in the cover, all prime implicants must be updated by removing the minterms covered by the selected term. If the number of elements covered by both a prime implicant and a selected term is $O(m)$, updating a single prime implicant is $O(m^2)$. For all N prime implicants, the complexity of step 3c is $O(Nm^2)$.

Now, step 3 may be repeated at most N times. So, the process of factorization, given the set of all prime pass implicants has complexity $O(N^4 + N^2.n^4 + N^2.m^2)$. We note that both N and m in the above expression are $O(2^n)$. Also, step 1 has complexity $O(2^n)$. Therefore, the worst case complexity for the entire algorithm is $O(2^{4n})$.

3.4 Example of Multilevel PTL Network Design

We consider the function F = Σ(2-5, 8, 9, 14, 15, 18, 20, 21, 23-25, 27, 30). The list of all prime pass implicants are given below:

$P_0 = x_4\bar{x}_2(\bar{x}_3)$, $P_1 = x_3\bar{x}_2(\bar{x}_4)$, $P_2 = \bar{x}_3\bar{x}_2(x_4)$, $P_3 = \bar{x}_4\bar{x}_2(x_3)$ $P_4 = x_5x_3x_1(\bar{x}_4)$, $P_5 = x_5x_4x_1(\bar{x}_3)$, $P_6 = \bar{x}_5x_3x_2(x_4)$, $P_7 = x_3x_2\bar{x}_1(x_4)$, $P_8 = \bar{x}_5x_4x_2(x_3)$, $P_9 = x_4x_2\bar{x}_1(x_3)$, $P_{10} = \bar{x}_5x_4x_3(x_2)$, $P_{11} = x_4x_3\bar{x}_1(x_2)$, $P_{12} = x_5\bar{x}_4x_1(x_3)$, $P_{13} = x_5\bar{x}_3x_1(x_4)$, $P_{14} = \bar{x}_5\bar{x}_4x_2(\bar{x}_3)$, $P_{15} =$

$\bar{x}_5\bar{x}_3x_2(\bar{x}_4)$, $P_{16} = \bar{x}_4x_2\bar{x}_1(\bar{x}_3)$, $P_{17} = \bar{x}_3x_2\bar{x}_1(\bar{x}_4)$, $P_{18} = \bar{x}_5\bar{x}_4x_3(\bar{x}_2)$, $P_{19} = \bar{x}_4x_3\bar{x}_1(\bar{x}_2)$, $P_{20} = \bar{x}_5x_4\bar{x}_3(\bar{x}_2)$, $P_{21} = x_4\bar{x}_3\bar{x}_1(\bar{x}_2)$, $P_{22} = \bar{x}_5\bar{x}_4\bar{x}_3(x_2)$, $P_{23} = \bar{x}_4\bar{x}_3\bar{x}_1(\bar{x}_2)$, $P_{24} = x_5x_4x_3x_2(\bar{x}_1)$, $P_{25} = x_4x_3x_2x_1(\bar{x}_5)$, $P_{26} = x_5x_4\bar{x}_3x_2(x_1)$, $P_{27} = x_5\bar{x}_4x_3x_2(x_1)$, $P_{28} = x_4\bar{x}_3x_2x_1(x_5)$, $P_{29} = \bar{x}_4x_3x_2x_1(x_5)$, $P_{30} = x_5\bar{x}_4\bar{x}_3x_2(\bar{x}_1)$, $P_{31} = \bar{x}_4\bar{x}_3x_2x_1(\bar{x}_5)$.

Initially, the prime pass implicants, P_0, P_1, P_2 and P_3 each cover eight minterms. All other prime pass implicants cover less than eight minterms. So, we select the first one, $P_0 = x_4\bar{x}_2(\bar{x}_3)$. Next we find $P_3 = \bar{x}_4\bar{x}_2(x_3)$ and form co-kernel/kernel = $\bar{x}_2/[x_4(\bar{x}_3) + \bar{x}_4(x_3)]$. If we divide the sum of all the pass prime implicants by the kernel $[x_4(\bar{x}_3) + \bar{x}_4(x_3)]$, we get quotient, $\bar{x}_2 + x_5x_1$. The PI's involved are: P_0, P_3, P_5 and P_{12} and the resulting factored form is $FF_1 = (\bar{x}_2 + x_5x_1)*[x_4(\bar{x}_3) + \bar{x}_4(x_3)]$. The score of this factored form is 10, since the total number of minterms covered by it is 20 and the number of columns required to implement it is 2. We select the factored form since its score is greater than that of P_0(=8). Next we select P_6, since it covers a maximum number of new minterms (4). Proceeding in a fashion identical to above, we find a second factored form $FF_2 = (\bar{x}_5x_2 + x_2\bar{x}_1)*[x_3(x_4) + \bar{x}_3(\bar{x}_4)]$, involving prime pass implicants P_6, P_7, P_{15} and P_{17}. We verify that the ordering restriction imposed by FF_2 (x_1 and x_5 must be above x_3) is compatible with that imposed by FF_1 (x_1, x_2 and x_5 must be above x_4). Therefore there is no conflict and we accept FF_2. After this step there are no more uncovered minterms, so the process terminates. Figure 3.5(a) shows the two-level implementation of the function F. Figure 3.5(b) shows a multilevel implementation of F corresponding to the factorized form $F = (\bar{x}_2 + x_5x_1)*[x_4(\bar{x}_3) + \bar{x}_4(x_3)] + (\bar{x}_5x_2 + x_2\bar{x}_1)*[x_3(x_4) + \bar{x}_3(\bar{x}_4)]$ obtained in this example.

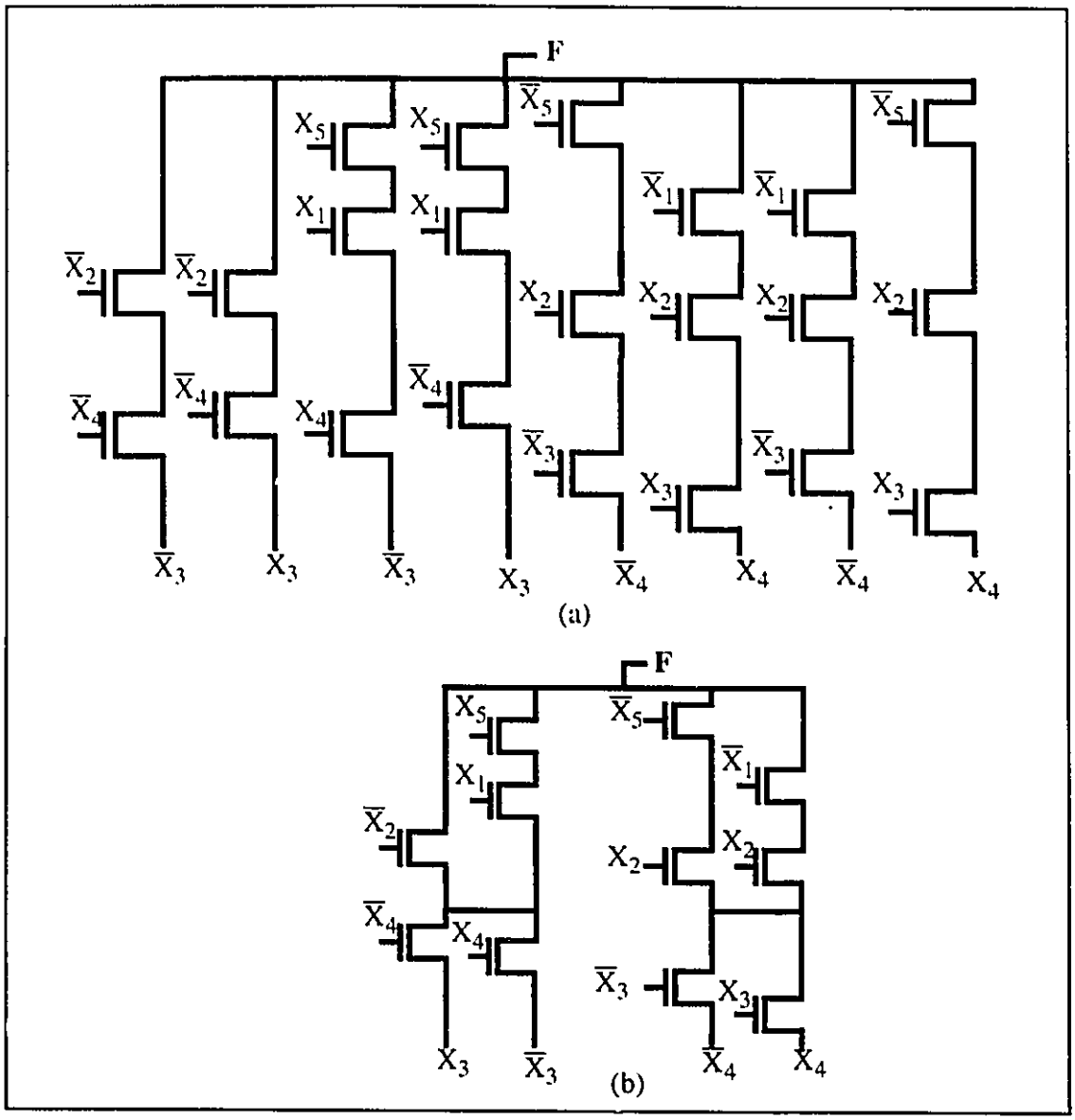

Figure 3.5 : Two-level and multi-level PTL networks representing the function F.

3.5 Conclusions

In this chapter we have outlined a new technique for synthesizing PTL networks. This is based on algebraic factorization. Factorization for PTL is significantly different from factorization for conventional logic networks and requires a totally different approach. Experiments with benchmark circuits (section 5.3) reveal that for many circuits, this

technique gives significantly better results. Heuristics for identifying factored forms is quite crucial in this approach. It is quite possible that more informed heuristics would give even better results and we suggest this as a possible area for further investigations.

Chapter 4

PTL Synthesis and 123-Decision Diagrams

4.1 Introduction

In this chapter we discuss multilevel logic synthesis for PTL networks using a decision diagram approach. We have developed a model for PTL networks which we call the *123 Decision Diagram (123-DD) Model.* This is derived from the Binary Decision Diagram (BDD) model proposed for representing Boolean functions [BRY86]. We have developed a number of transformations to manipulate a 123-DD. Our synthesis technique repeatedly uses these transformations to obtain a model of a PTL network. One interesting aspect of our approach is that the synthesis technique is always guided by layout considerations (including interconnection cost) and the 123-DD model of a PTL network may be directly mapped into silicon.

4.2 123-Decision Diagrams: A Graph Model for PTL

4.2.1 Can We Use Decision Diagrams for PTL Networks

It is possible to use ordered binary decision diagrams to directly model a PTL network. This is done by observing that

- Each edge starting from a node represents a transistor
- The label associated with a node represents an input variable
- An edge with label 1(0) corresponds to a transistor with the input variable (complement of the input variable) applied to its gate
- Each node represents a net in the circuit
- The leaf nodes represent pass signals

However, networks realized directly from OBDD's in this manner may be inefficient due to the following reasons:

- OBDD's allow only 0s and 1s to be used as pass signals. Arbitrary inputs or their complements cannot be used as pass signals
- OBDD's do not take into account relative placement of transistors at each level
- OBDD's do not consider the cost of interconnection between transistors
- It may be possible to get a more efficient realization if we relax the restriction of having exactly two edges (transistors) from each node

We now establish the above observations with some examples. Figure 4.1(a) shows a PTL network using only 0's and 1's as pass variables. Figure 4.1(b) shown a more efficient realization of the same function, using additional pass variables. Figure 4.2 shows that it is important to take into account relative placement of transistors at each level and the cost of interconnection between transistors. 4.2(a) shows interconnections in a PTL network where transistors with complemented inputs are arbitrarily placed to the left of the ones with true inputs. 4.2(b) shows the same connections without any cross-over of interconnecting wires.

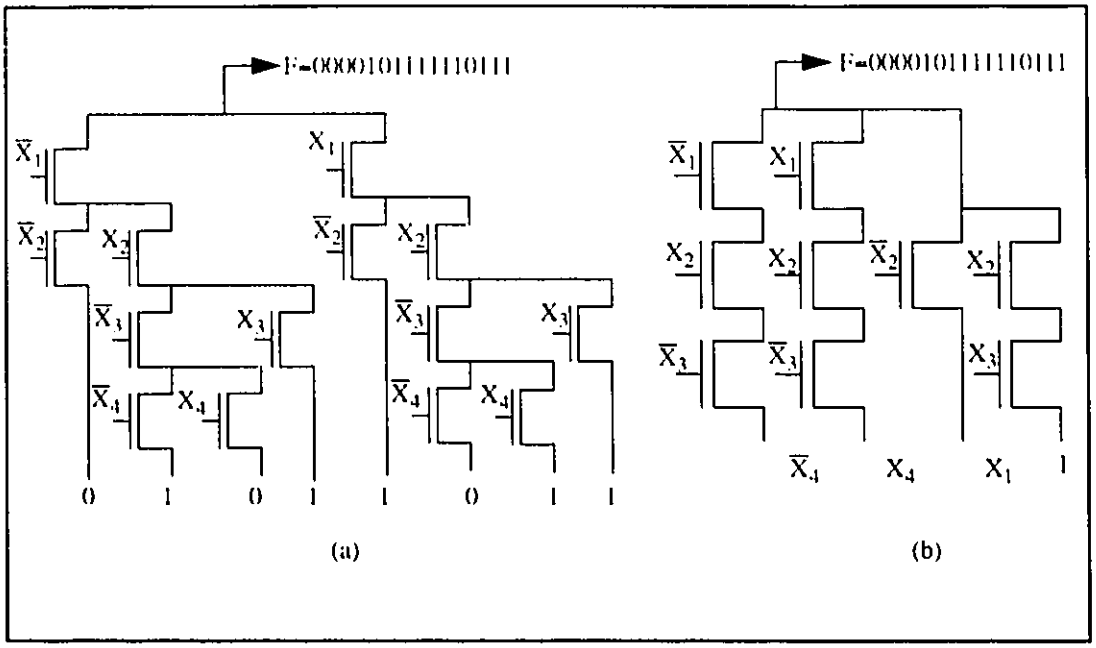

Figure 4.1 : Two PTL networks based on (a) BDDs and (b) 123-DDs.

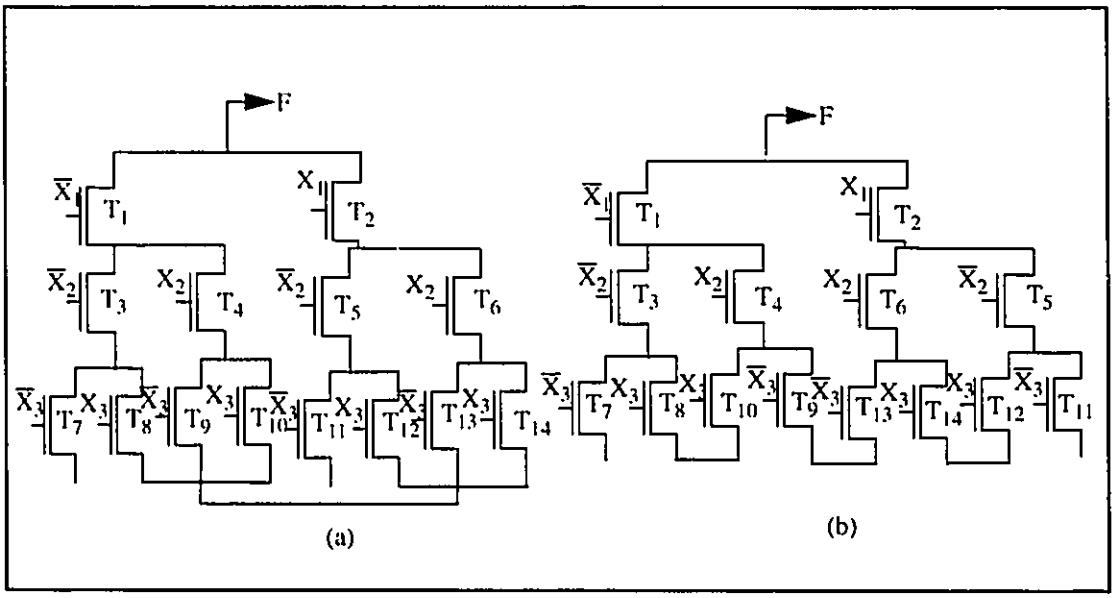

Figure 4.2 : Two PTL networks with different transistor placements

4.2.2 Our Layout Strategy

In order to construct an actual PTL network, based on decision diagrams, we need to have a layout strategy which allows direct, easy mapping from the model to silicon. We consider a double-metal layout strategy with the following simple rules:

(i) Transistors in series are connected vertically using the first metal layer.
(ii) Input signals are carried to the gates of transistors using horizontal lines through the second metal layer.
(iii) Horizontal connections between adjacent columns of transistors are also done using the first metal layer. This type of connection is called a *local merge*.
(iv) Horizontal connections between non-adjacent columns are done with the second metal layer. These are called *nonlocal merges*.

We will see later that major savings in the area of a PTL network results from sharing of subnetworks. We therefore try to maximize the number of such sharings. Local merges involving adjacent columns of transistors are always feasible and our heuristic always looks for local merges. It is relatively more difficult to connect two non-adjacent columns of transistors. We obviously cannot use the first metal layer and the second metal layer has to be used with care since it is already used to carry the input signals. Each row of transistors has two horizontal lines to carry an input signal and its complement. The space between the horizontal lines in successive rows of transistors determine the extent to which nonlocal merges are possible. We have verified that, for a specific technology(1.2 micron double metal technology) and an existing design of a transistor, it is possible to have one horizontal line for nonlocal merges. This scenario may change as we move from one technology to another. For example, availability of another metal layer may allow more horizontal lines for nonlocal connections. In general, for our synthesis approach, we allow a fixed number of horizontal lines for nonlocal merges, the number being determined by the layout strategy we use.

4.2.3 123-Decision Diagram Model

Our model, 123-Decision Diagram (123-DD), overcomes the limitations associated with directly using the OBDD model for synthesis of PTL networks. As the name suggests, 123-DDs relax the restriction of *binary* decision diagrams of having exactly two edges associated with each node. In 123-DDs each node may have one, two or three edges associated with it as needed.

The main characteristics of 123-Decision Diagrams are given below:

- The terminal nodes are shown as labelled squares. The label on each square node denotes a pass signal, which may be 0, 1, any of the input variables or their complements
- A nonterminal node is shown as a labelled circle. It represents a net in the circuit. A node at the i^{th} level has label X_i, denoting the i^{th} input variable
- Each node in the 123-DD has a unique number as its Node_id
- If two nonterminal nodes are connected by a dashed line, the dashed line represents the fact that the two nets are connected to form a single net
- Each nonterminal node has 1, 2 or 3 outgoing edges. Each edge has a label of 0, 1 or 2
- Each edge with a label 0(1) from a node with label X_i corresponds to a transistor with its gate connected to $\overline{X}_i$ (X_i)
- An edge with a label 2, indicates a metal link
- The relative position of the edges from a node is important and there may be restrictions on the ways these positions can be changed with respect to each other. This positional information is represented in terms of a list of the nodes at the current level. This is discussed in more detail in section 4.4.

The 123-Decision Diagram allows sharing of subgraphs similar to BDD's. However, such sharing of subgraphs is done only after an analysis to see if it is feasible, taking into account the specific VLSI technology used and the strategy to implement the PTL networks.

We would like to emphasize that the 123-DD model does not share the canonic representation property of BDD's. Its primary use is to represent PTL networks and its main advantage is that it can be directly mapped into silicon. An example of a 123-DD corresponding to the partially synthesized PTL network of Figure 4.2(b) is shown below.

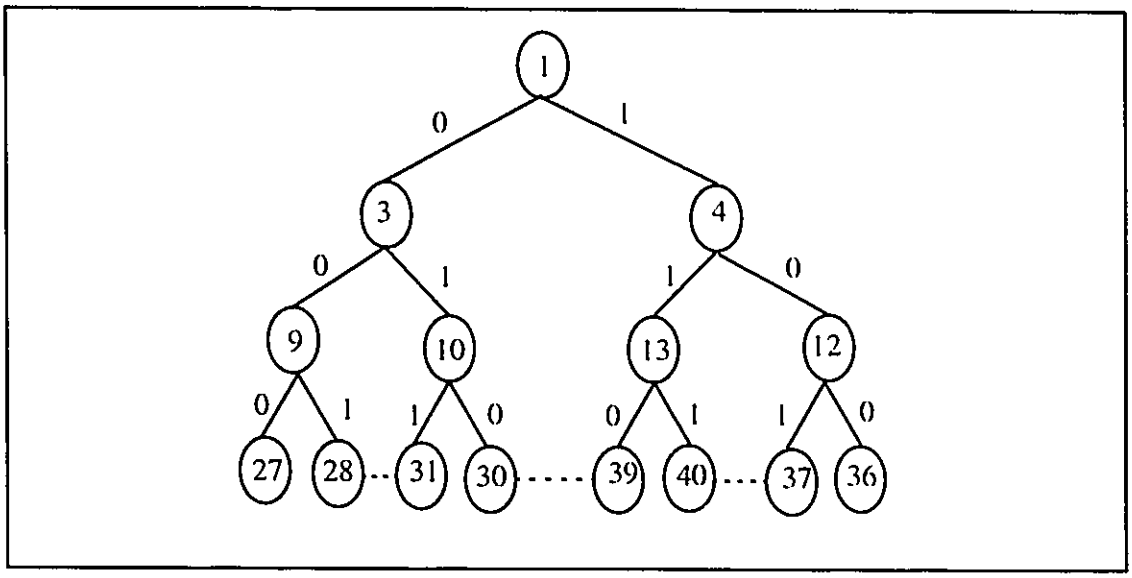

Figure 4.3 : Example of 123-DD

4.3 Useful Properties and Definitions for 123-DDs

Definition 4.1: An edge e labelled 0(1) from a node with label X_i is said to be ***activated*** *if the input X_i = 0(1). An edge with label 2 is always* ***activated.***

Definition 4.2: A path p between two nodes in a 123-DD is said to be ***activated****, for a given input combination, if and only if all edges in the path are* ***activated.***

As mentioned in chapter 2 (section 2.4.2), each node in a BDD represents a Boolean function which can be specified in terms of a bitmap. Similarly, a node in a 123-DD also denotes a Boolean function. However, a bitmap is not adequate for representing a function associated with a node in a 123-DD. We define a more generalized term below.

Definition 4.3 A ***signal string*** *S of length m is a string of symbols* S_i, $0 \leq i \leq m\text{-}1$, $S_i \in \{0, 1, d, Z, X_j, \neg X_j\}$ *, where Z denotes the high impedance state, and* X_j *can be any input variable and d represents a don't care.*

The high impedance state (Z) occurs if there is no path activated from the node to any pass signal for a given input combination. Henceforth we will use $S_i \in \{0, 1, d, Z, X_j, \neg X_j\}$ to represent the i^{th} symbol in the signal string S. The signal string associated with a node at the k^{th} level is of length $2^{(n-k+1)}$.

Definition 4.4: The ***child*** *of a node N at the* i^{th} *level is any node at level i+1, which is connected to N through a single edge labelled 0, 1 or 2*

Definition 4.5: Given two signal strings S1 and S2 of length m, the operation combine_signal(S1, S2) combines S1 and S2 and creates a new signal string S3 of length m. This operation will be denoted using infix notation as $S_1 \otimes S_2$. *The operation is given below.*

For all i, $0 \leq i \leq m-1$, IF $S1_i = Z$ THEN $S3_i = S2_i$

ELSE IF $S2_i = Z$ THEN $S3_i = S1_i$

ELSE IF $S1_i = S2_i$ THEN $S3_i = S2_i$

Definition 4.6: Given a parent node N_1, edge label e_{12} from N_1 to N_2, one of its child nodes, and the signal strings S of length m associated with the child, the operation table_map(S, e_{12}, N_1) determines the contribution of N_2 to the signal string of N_1. The operation is given below.

Let node N_1 have label X_i and let the signal strings $S_0 \ldots S_{m-1}\ S_m \ldots S_{2m-1}$ and $s_0 s_1 \ldots s_{m-1}$ be associated with the parent and the child node. The relationship between $S_0 \ldots S_{m-1} S_m \ldots S_{2m-1}$ and $s_0 s_1 \ldots s_{m-1}$ is given in Table 4.1. Given a signal string $s_0 s_1 \ldots s_{m-1}$ and label of the edge joining parent node and child node, this table determines what signal string will be associated with the parent node due to this child node.

If a parent node has more than one child, repeated applications of operations *combine_signal* and *table_map* allow us to determine the signal string S associated with any node N given the signal strings of its children. Henceforth we will call this operation *generate_signal_string*.

Table 4.1: Signal String of a Node from its Children

Index of parent = i	S_j and $S_{(j+m)}$ in signal string S of parent					
Label of Edge to Child	0		1		2	
j^{th} symbol in signal string of child (s_j)	S_j	$S_{(j+m)}$	S_j	$S_{(j+m)}$	S_j	$S_{(j+m)}$
$s_j \neq X_i$ and $s_j \neq \overline{X}_i$	s_j	Z	Z	s_j	s_j	s_j
$s_j = X_i$	-	-	-	-	0	1
$s_j = \overline{X}_i$	-	-	-	-	1	0

Example 4.1:

Let N be a node, at the i^{th} level, in a 123-DD with three children N_0, N_1 and N_2, connected to N through edges labelled 0, 1 and 2 respectively. Let S S_0, S_1 and S_2 be the signal strings associated with N, N_0, N_1 and N_2, such that S_0 = 0011ZZZZ, S_1 = 1001ZZZZ and S_2 = ZZZZ$X_i X_i$01. Then, according to Table 4.1, S = 0011000110011101.

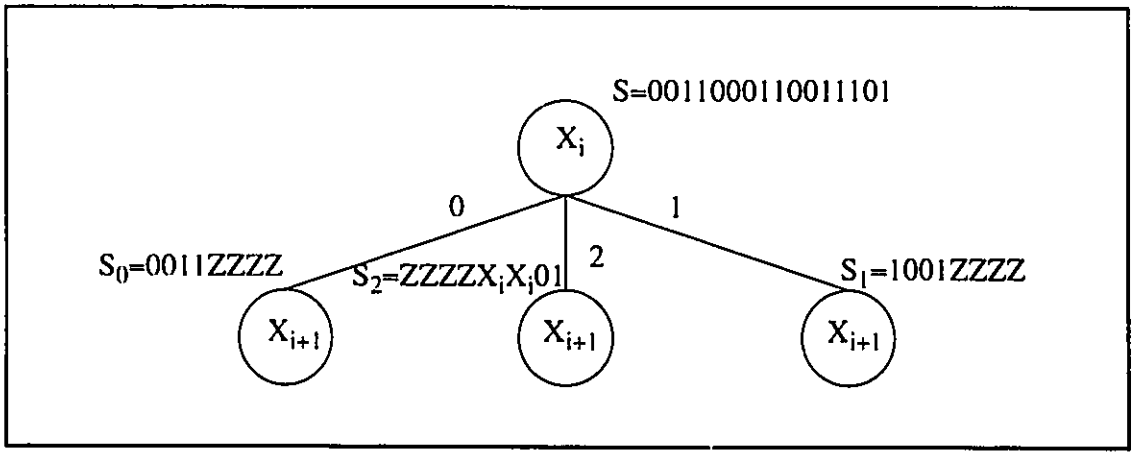

Figure 4.4 : Illustration of Example 4.1.

Property 4.1 : Each symbol s_j such that $s_j \neq Z$ in the signal string of a child connected to its parent through an edge labelled 0(1 or 2) occurs in the corresponding position in the first(second or both) half of the signal string of the parent. Thus the position of s_j in the signal string of the parent can be determined from the label of the edge connecting the parent and child node. Property 4.1 follows from Table 4.1.

Property 4.2 : Given a signal string s associated with a node at the current level, the position of s in the signal string of an ancestor of the node can be determined from the sequence of edge labels connecting the two nodes. *Property 4.2* is obtained by extrapolating *property 4.1*.

Property 4.3 : Let p1 = [N - e_{01} - N_{01} -e_{11} -N_{11} - ... - e_{k1} - N_{k1}] and p2 = [N - e_{02} - N_{02} -e_{12} -N_{12} - ... - e_{k2} - N_{k2}] be two paths from a node N to two nodes N_{k1} and N_{k2} at the current level. Let S, s_1 and s_2 be the signal strings associated with nodes N, N_{k1} and N_{k2}. The two paths p1 and p2 can be replaced by a third path p3 = [N - e_{03} - N_{03} -e_{13} -N_{13} - ... - e_{k3} - N_{k3}] (as shown in Figure 4.5) without changing S, if the following conditions are satisfied:

- The label of edge e_{01}(e_{02}) is 0(1)
- For all i, $1 \leq i \leq k$ label of e_{i1} = label of e_{i2}
- $(s_1, s_2) \Rightarrow s$
- each node N_{01}, N_{11}, ... ,N_{k1} in p1 and N_{02}, N_{12} , ... , N_{k2} in p2 has only one parent
- signal string of N_{k3} = s
- label of e_{03} = 2
- for all i, $1 \leq i \leq k$ label of e_{i3} = label of e_{i2}

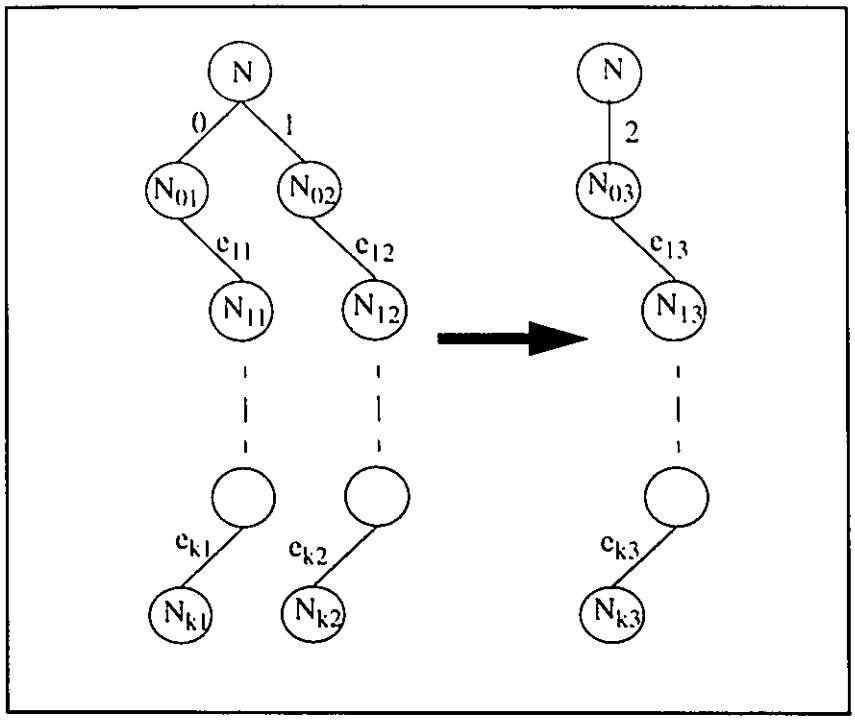

Figure 4.5 : Illustration of Property 4.3

According to property 4.2, the signal strings s_1 and s_2 , associated with N_{k1} and N_{k2} will occur in the same position in the signal strings associated with N_{01} and N_{02}. Therefore, property 4.3 follows by using properties 4.1 and Table 4.1.

Property 4.4 : Let p1 = [N - e_{01} - N_{01} -e_{11} -N_{11} - ... - e_{k1} - N_{k1}] and p2 = [N - e_{02} - N_{02} -e_{12} -N_{12} - ... - e_{k2} - N_{k2}] be two paths from a node N, at the j^{th} level, to two nodes N_{k1} and N_{k2} at the current level. Let S, s_1 and s_2 be the signal strings associated with nodes N, N_{k1} and N_{k2}. The two paths p1 and p2 can be replaced by a third path p3 = [N - e_{03} - N_{03} -e_{13} -N_{13} - ... - e_{k3} - N_{k3}] (as shown in Figure 4.6) without changing S, if the following conditions are satisfied:

- The label of edge e_{01}(e_{02}) is 0(1)
- For all i, $1 \leq i \leq k$ label of e_{i1} = label of e_{i2}

- s_1 consists of all 0's(all 1's) and s_2 consists of all 1's(all 0's)
- each node $N_{01}, N_{11}, \ldots, N_{k1}$ in p1 and $N_{02}, N_{12}, \ldots, N_{k2}$ in p2 has only one parent
- label of $e_{03} = 2$ and for $1 \leq i \leq k$ label of e_{i3} = label of e_{i2}
- signal string of $N_{k3} = s$, consists of all X_j's($\overline{X}_j$'s)

According to *property 4.2*, the signal string s_1 and s_2, associated with N_{k1} and N_{k2} will occur in the same position in the signal strings associated with N_{01} and N_{02}. Therefore, *property 4.4* follows by using properties 4.1 and Table 4.1.

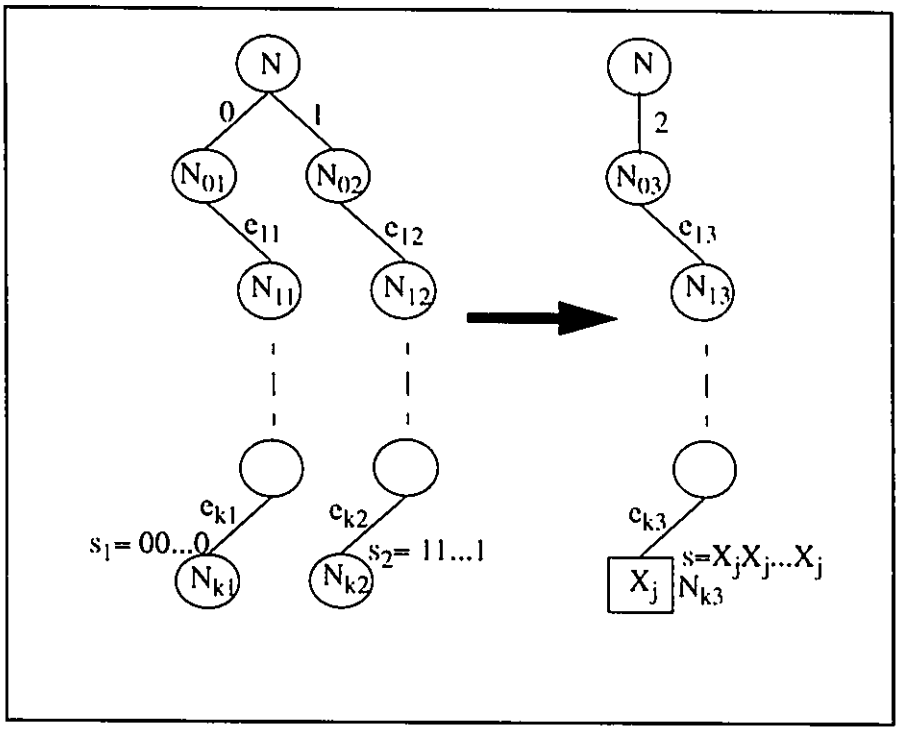

Figure 4.6 : Illustration of Property 4.4.

4.4 The List Structure

As mentioned earlier, we use the 123-DD to represent a function as realized by a PTL network. Therefore, in addition to the connectivity information, it must also keep track of

the positional information of the transistors in the actual circuit realization. This is done in terms of a *position list*. The position list is a list structure which contains the Node_ids of all the nodes at the current level. Each node in the 123-DD has a unique Node_id. The Node_id of the root node is always 1. For any other node, we generate the Node_id as follows.

Definition 4.7: A node N connected to its parent through an edge labelled e has a Node_id = 3n+e, where n is the Node_id of the parent.

There are two types of lists in our position list. Type I lists consist of a number of elements within square brackets, for example, $L_1 = [e_1, e_2, \ldots , e_m]$. Type II lists consist of a number of elements within braces, for example, $L_2 = \{ e_1, e_2, \ldots , e_k\}$. An element of a list can be:

(i) a Node_id of a node at the current level or

(ii) a list of Type I or II.

(iii) a Node_id of a *dummy* node[1] at the current level.

If certain nodes are removed from the current level as a result of a transformation on the 123-DD (sections 4.5.2 and 4.5.3) the corresponding Node_ids must also be removed from the position list. The current ordering of the elements in a Type I list is important. There are only two possible orderings for the elements in a Type I list. They must occur either in the specified order or in the reverse order. For a Type II list, however, the current ordering is not important. The elements in the list can be reorganized in any order. If a list contains sublists, we consider the entire sublist to be a single element for the purpose of reordering. The elements of the sublist can be reordered again according to the type of the sublist.

1. A dummy node is one which is not currently in the 123-DD, but may be inserted subsequently as a result of transformations described later.

As an example, we consider a 123-DD expanded up to the fourth level (Figure 4.7(a)). Each node, in Figure 4.7(a), has a unique Node_id, which is shown within each node. Then all possible orderings of the nodes at the current level are given by the position list PL_1 = {{{27, 28, 29_d}, {30, 31, 32_d}, {33_d, 34_d, 35_d}}, {{36, 37, 38_d}, {39, 40, 41_d}, {42_d, 43_d, 44_d}}, {{45_d, 46_d, 47_d}, {48_d, 49_d, 50_d}, {51_d, 52_d, 53_d}}}. We note that nodes with a subscript of d (e.g., 29, 32, 33) do not actually occur in the 123-DD. Some of the valid orderings for PL_1 are :

(i) [27, 28, 29_d, 30, 31, 32_d, 33_d, 34_d, 35_d, 36, 37, 38_d, 39, 40, 41_d, 42_d, 43_d, 44_d, 45_d, 46_d, 47_d, 48_d, 49_d, 50_d, 51_d, 52_d, 53_d]

(ii) [28, 27, 29_d, 30, 31, 32_d, 33_d, 34_d, 35_d, 39, 40, 41_d, 36, 37, 38_d, 42_d, 43_d, 44_d, 45_d, 46_d, 47_d, 48_d, 49_d, 50_d, 51_d, 52_d, 53_d]

(iii)[30, 31, 32_d, 33_d, 34_d, 35_d, 28, 29_d 27, 37, 36, 38_d, 39, 40, 41_d, 42_d, 43_d, 44_d, 45_d, 46_d, 47_d, 48_d, 49_d, 50_d, 51_d, 52_d, 53_d]

To show some features of this representation, let us impose the restriction that nodes 27 and 37 must always be adjacent to each other (Figure 4.7(b)), so that the transistor corresponding to the edge connecting nodes 9 and 27 is adjacent to the transistor connecting nodes 12 and 37. It may be verified that if the position list is changed to PL_2 = {[{{30, 31, 32_d}, {33_d, 34_d, 35_d}},{28, 29_d} 27, 37, {36, 38_d}, {{39, 40, 41_d}, {42_d, 43_d, 44_d}}], {{45_d, 46_d, 47_d}, {48_d, 49_d, 50_d}, {51_d, 52_d, 53_d}}}the above restriction is always satisfied. We will see later how this is actually achieved. We see that PL_2 only allows option (iii) from the above, but does not allow options (i) and (ii) since nodes 27 and 37 are not adjacent in those cases.

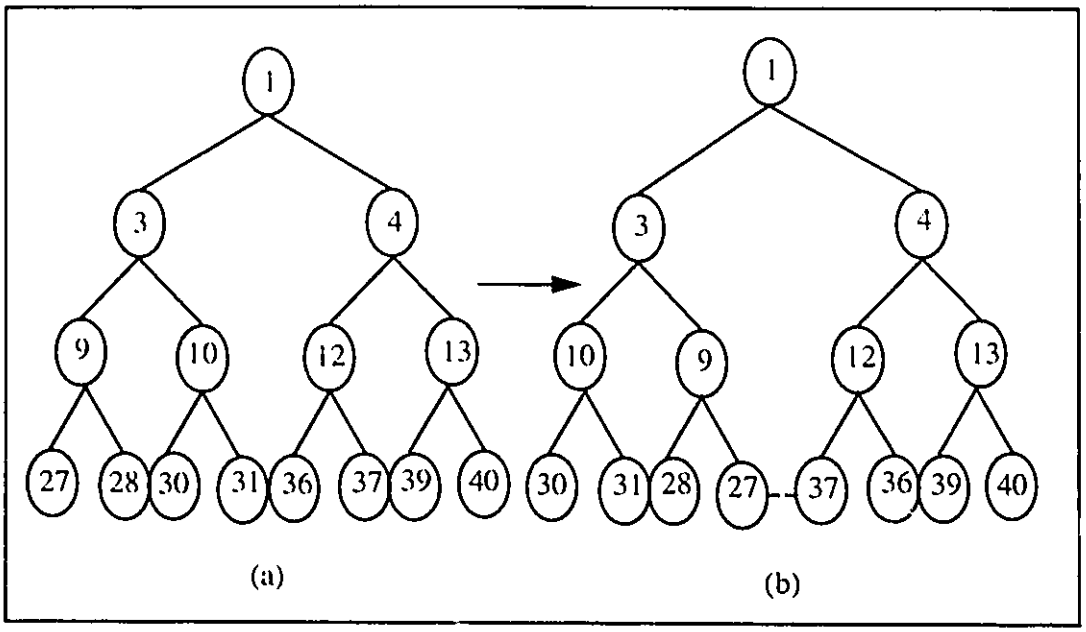

Figure 4.7 : Initial and Modified 123-DDs Corresponding to PL_1 and PL_2.

4.5 The Transformation rules

There are five different transformation rules, which are used to manipulate the 123-Decision Diagram. These are given below:

Transformation 4.1 : Generating children of a nonterminal node.

Transformation 4.2 : Replacing two distinct paths to two nonterminal nodes by a single common path.

Transformation 4.3 : Replacing two distinct paths to two nonterminal nodes by a single common path to a terminal node.

Transformation 4.4 : Converting nonterminal nodes to terminal nodes.

Transformation 4.5 : Merging subgraphs.

Some of these are very similar to the corresponding rules for OBDDs. As in OBDDs, for a completely specified function, a set of argument values X_1, X_2, ...X_n describes a path in

the 123-DD starting from the root to a terminal vertex. In this section, we will describe our transformation rules.

4.5.1 Generating Children of a Node

The first transformation rule is used to expand the 123-DD to the next level. This rule is almost identical to that used in BDD's (Transformation 2.1) except that signal strings are used in 123-DD in place of bitmaps in BDD's.

Transformation 4.1: From a nonterminal node N at the i^{th} level with an associated signal string S of length 2m, generate two children of N, $N_0(N_1)$ at the $(i+1)^{th}$ level, with signal strings $S_0(S_1)$ of length m, connected to N through edges labelled 0(1), such that $S_0 \bullet S_1 = S$. An example is shown in Figure 4.8

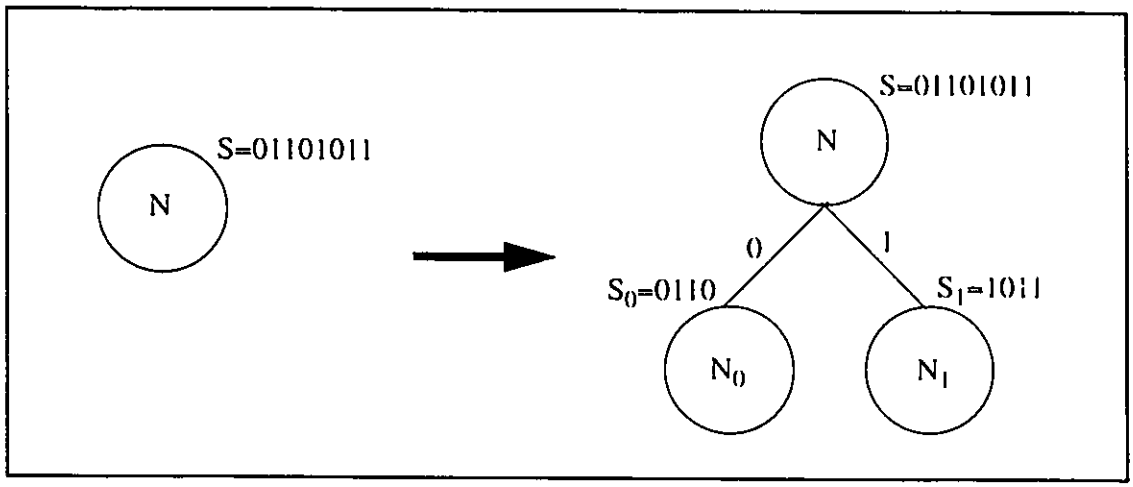

Figure 4.8 : Example of Transformation 4.1

4.5.2 Combining Two Paths I

Transformation 4.2 : If there are two paths p1 = [N - e_{01} - N_{01} -e_{11} -N_{11} - ... - e_{k1} - N_{k1}] and p2 = [N - e_{02} - N_{02} -e_{12} -N_{12} - ... - e_{k2} - N_{k2}] from a node N to two nodes N_{k1} and N_{k2} at the current level, with signal strings s_1 and s_2, such that they satisfy conditions (i)- (iv)

of property 4.3, then replace paths p1 and p2 by path p3 = [N - e_{03} - N_{03} -e_{13} -N_{13} - ... - e_{k3} - N_{k3}] where label of edge e_{03} = 2 and for all i, $1 \leq i \leq k$ label of e_{i3} = label of e_{i2} and signal string of N_{k3} is $(s_1, s_2) \Rightarrow s_3$. An example of Transformation 4.2 is shown in Figure 4.9.

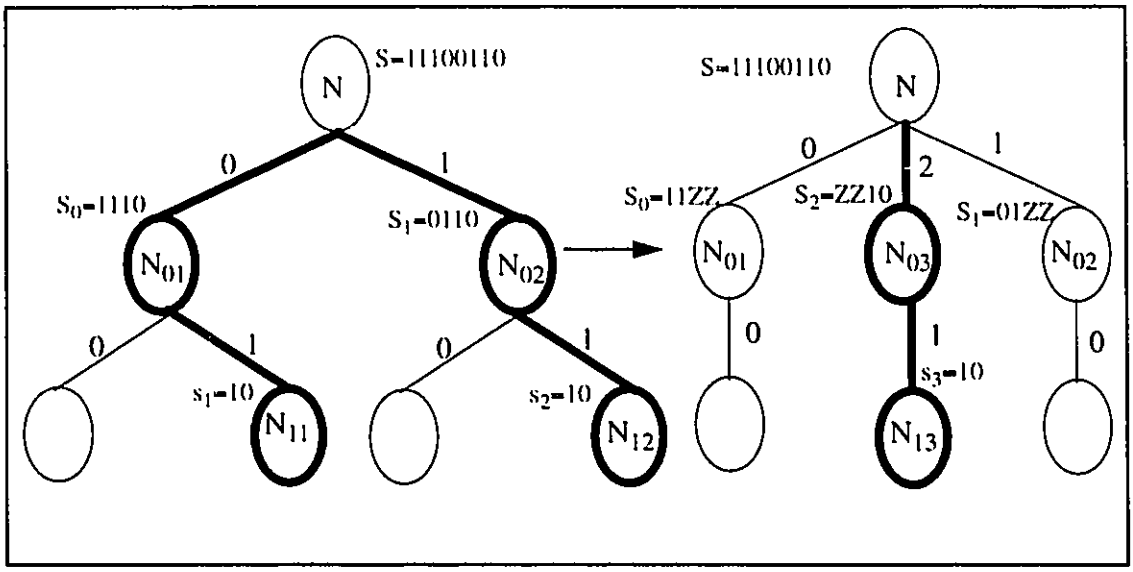

Figure 4.9 : Example of Transformation 4.2

4.5.3 Combining Two Paths II

Transformation 4.3 : If there are two paths p1 = [N - e_{01} - N_{01} -e_{11} -N_{11} - ... - e_{k1} - N_{k1}] and p2 = [N - e_{02} - N_{02} -e_{12} -N_{12} - ... - e_{k2} - N_{k2}] from a node N, with label X_j, at the j^{th} level, to two nodes N_{k1} and N_{k2} at the current level, with signal strings s_1 and s_2, such that they satisfy conditions (i) - (iv) of property 4.4, then replace paths p1 and p2 by path p3 = [N - e_{03} - N_{03} -e_{13} -N_{13} - ... - e_{k3} - N_{k3}] where label of edge e_{03} = 2 and for all i, $1 \leq i \leq k$ label of e_{i3} = label of e_{i2} and N_{k3} is a terminal node with pass signal $X_j(\overline{X}_j)$. An example of Transformation 4.3 is shown below in Figure 4.10.

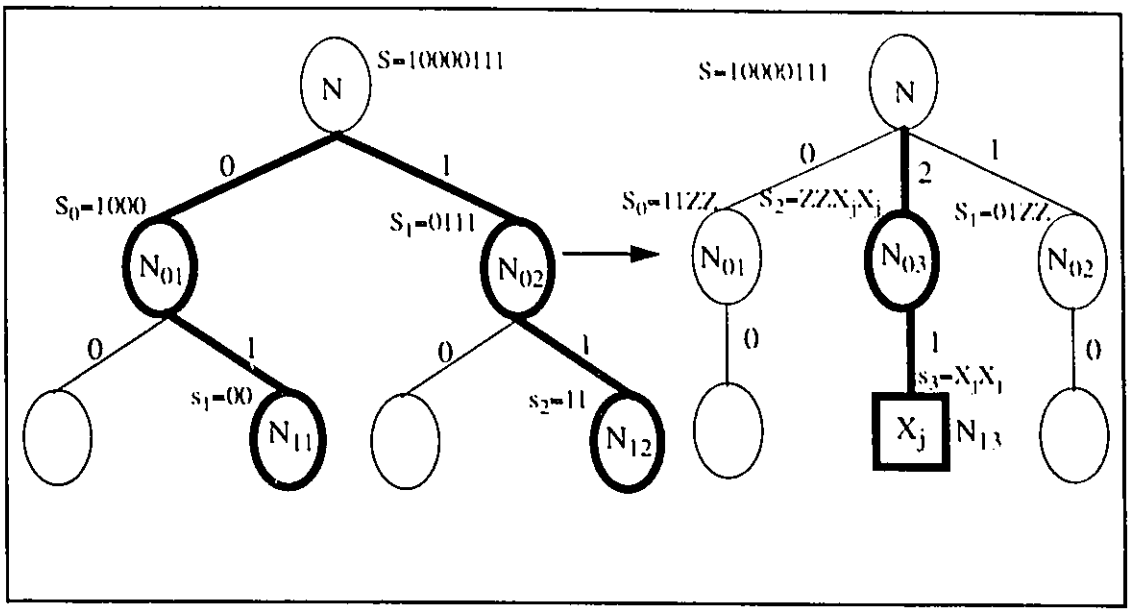

Figure 4.10 : Example of Transi mation 4.3

4.5.4 Converting Nonterminal Nodes to Terminal Nodes

Each branch in a PTL network must have a pass signal associated with it. When the pass signal has been determined for a particular node, then that node becomes a *terminal node*. Terminal nodes are never expanded to generate children at the next level. Transformation 4.4 is used to convert a nonterminal node to a terminal node and determine its pass signal. Terminal nodes can also be created as a result of Transformation 4.3 (as discussed earlier).

Transformation 4.4:

(i) If the signal string associated with a node N consists of all 0's (all 1's) then convert N to a terminal node with pass signal 0 (1).

(ii) If a node N, with label X_i, has a signal string S of length 2m such that the first m symbols of S consist of all 0's (all 1's) and the last m symbols consist of all 1's (all 0's), then convert N to a terminal node with pass signal X_i ($\overline{X}_i$)

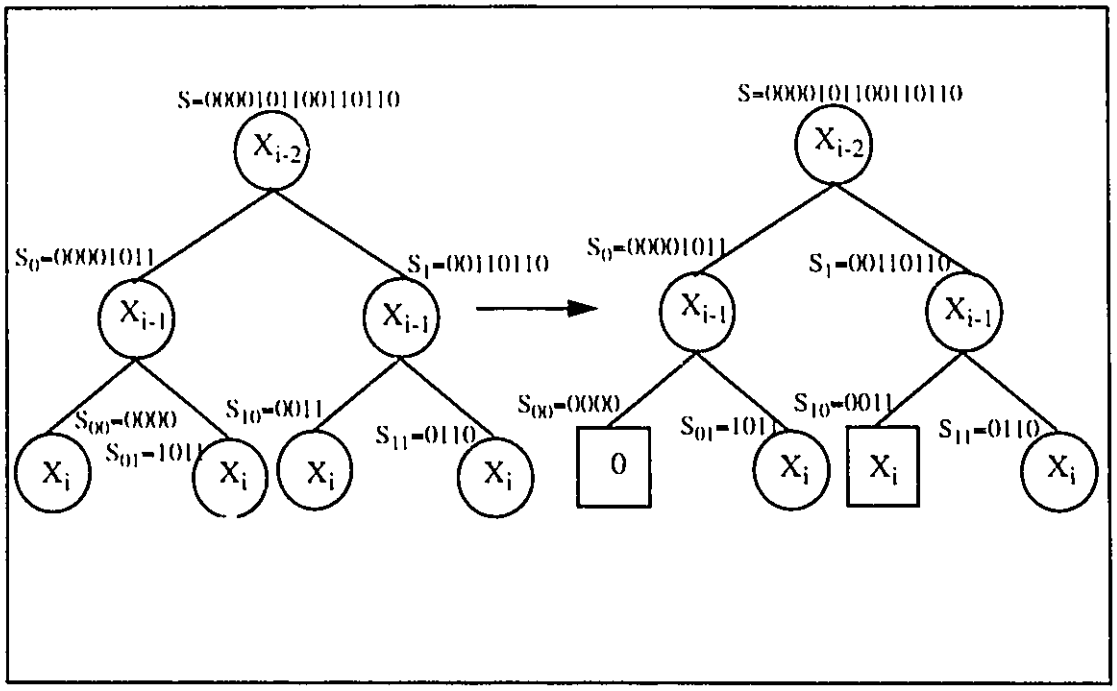

Figure 4.11 : Example of Transformation 4.4

4.5.5 Merging of Subgraphs

If two or more nodes have identical signal strings it means that the two nodes denote the same function. Instead of duplicating the subnetwork required to implement this function, the output from a single subnetwork may be connected to each node.

Transformation 4.5: If two or more nodes at the current level have equivalent signal strings associated with them, then merge the subgraphs below these nodes.

Transformation 4.5, for 123-Decision Diagrams is analogous to Transformation 2.3 for OBDDs and the justification for it is the same as in OBDDs [BRY86]. An example of Transformation 4.5 is given below.

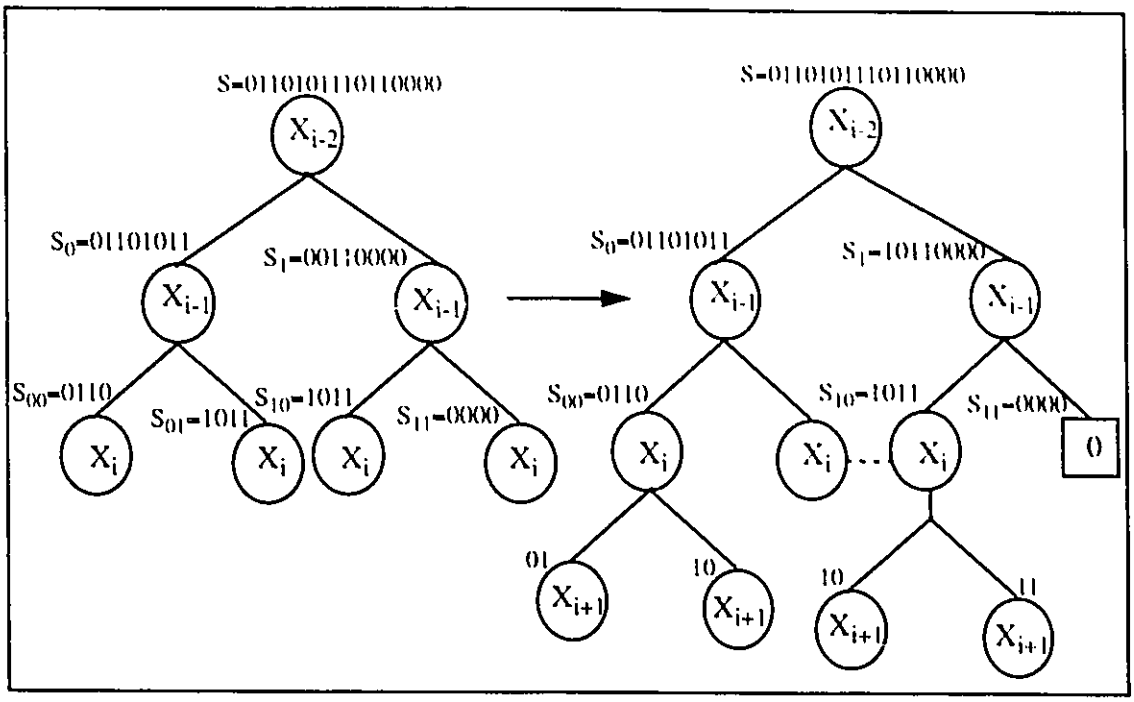

Figure 4.12 : Example of Transformation 4.5

4.5.6 Layout Factors Affecting Transformations

When carrying out the various transformations, certain layout considerations should be taken into account. As noted earlier, the 123-Decision Diagram, in addition to representing the Boolean function being implemented, also carries information on the relative placement of the transistors used to implement the function. We use this information to determine if certain transformations should not be carried even if the conditions for applying the transformation are satisfied. We always carry out Transformations 4.1 and 4.4 whenever the appropriate conditions, as mentioned in sections 4.5.1 and 4.5.4 are met. The other transformations are carried out only when relevant VLSI considerations are satisfied.

4.5.6.1 Layout Considerations for Transformation 4.5

In the synthesis of OBDDs, an arbitrary number of subgraph merges (Transformation 2.3) are allowed at each level. Similarly, we should be able to apply our corresponding rule

(Transformation 4.5) on an arbitrary number of nodes as long as they have equivalent signal strings. However, since the 123-DD represents an actual layout, we have to keep track of the placement of transistors and the interconnections between them. The positional information is stored the form of a position list (section 4.4).

If two nodes, with equivalent signal strings, can be brought next to each other in the position list, then we can always merge the corresponding subgraphs (local merge). A local merge is done using the first metal layer (as mentioned in section 4.2.2) and has no associated cost in our model. Therefore, a local merge is always advantageous and is carried out as long as the two nodes may be merged by application of Transformation 4.5.

However, if two nodes N_1 and N_2 having equivalent signal strings cannot be brought next to each other in the position list, then we have to consider a nonlocal merge (section 4.2.2). A nonlocal merge is done using the second metal layer. The number of horizontal lines for nonlocal merges is limited and we have to determine the feasibility of a horizontal line from N_1 to N_2 before carrying it out.

4.5.6.2 Layout Considerations for Transformations 4.2 and 4.3

As discussed earlier, both Transformation 4.2 and 4.3 reduce the number of nodes at the current level, but may increase the number of nodes at one or more of the previous levels. One of the factors determining the cost of a particular implementation of a PTL network is the area required for implementation. This in turn depends on the width of the network. In our implementation, the width of a PTL network is directly related to the maximum number of nodes (N_{max}) at any given level of the corresponding 123-DD. Therefore, one goal of our synthesis algorithm is to reduce N_{max} as much as possible. Another consideration is that, in certain cases, there may be no way to place the additional transistors in the new path obtained by applying Transformation 4.2 or 4.3, according to our layout strategy (section 4.2.2). Therefore, before actually applying Transformations 4.2 or 4.3, we check if

(i) N_{max} is increased as a result of this transformation and

(ii) The new node, which will be created at the current level as a result of the transformation, occurs as a *dummy* node in the position list.

The transformation is carried out only if both of the above conditions are satisfied. The first condition ensures that the width of the PTL network does not increase as a result of the transformation. The second condition ensures that there is an efficient way of placing the transistors in the new path, which is consistent with our layout strategy.

4.6 Synthesis Procedure

4.6.1 Synthesis procedure in a nutshell

Our procedure for generating a 123 DD for a given Boolean function F of n variables is given below. Here the function F is defined in the form of a bit map of size 2^n . This is a top-down recursive, heuristic which proceeds level by level. Each level corresponds to one input variable (in true or complemented form). The inputs are arranged in order from the most significant (top level) to the least significant (bottom level).

The first step is to determine a suitable variable ordering. After this, the procedure starts with a single node, at level 1, which corresponds to the most significant variable X_1. The signal string associated with the initial node at level 1, specifies the entire function that has to be implemented. At this point, the current level is 1. The 123-DD is expanded level by level and transformations are carried out on the model based on the signal strings associated with the nodes at the current level.

1. Pick an optimum ordering of input variables

2. Redefine the bitmap

If the original variable ordering has been changed, modify the initial bitmap to reflect this change.

3. Current level = 1

Initialize the current level to 1. Step 4 must be repeated for each level in the 123-DD.

4. REPEAT

a. Generate children.

Use Transformation 4.1, to generate two children from each nonterminal node at the current level.

b. Current level = Current level + 1.

The next level, consisting of nodes generated in 4a, becomes new current level.

c. Reduce the number of paths in the 123-DD.

- *For each pair of nodes (N_1, N_2) at the current level, check if the two paths from the common ancestor to N_1 and N_2 can be combined into a single path by applying Transformation 4.2. or Transformation 4.3*
- *Apply these transformations only if all relevant layout constraints (section 4.5.6.2) are satisfied*

d. Convert nonterminal nodes to terminal nodes.

Check each nonterminal node at the current level and convert to a terminal node if it satisfies the conditions of Transformation 4.4.

e. Merge subgraphs

For each pair of nonterminal nodes (N_1, N_2) at the current level check if Transformation 4.5 is applicable; if so

- *connect N_1, N_2 by local merge if possible*
- *else connect N_1, N_2 using nonlocal merge if feasible*

UNTIL *there are no nonterminal nodes at the current level.*

At each level, the various transformations may be applied to the nodes to manipulate the graph. The main objective of such transformations is to try to reduce the 'width' of the graph, which is defined as the maximum number of nodes in one level. This is the primary cost factor in the synthesis procedure. When no more transformations can be applied to any of the nodes at the *current level*, the nodes are expanded to generate the next level, which becomes the new *current level*. This process continues for each level until pass signals for every node in the current level have been determined.

4.6.2 Complexity of the Algorithm

In this section we will determine the complexity of synthesizing the 123-Decision Diagram for an n-input function, given a specific variable ordering. This is done by considering the complexity of each transformation applied at a particular level. Suppose, we are currently considering the i^{th} level. The maximum number of nodes at this level is $N=2^i$. Transformation 4.1, which generates the children of a node, involves simple bit manipulations and has complexity $O(1)$. Applying this to all the nodes gives us a complexity $O(N)$.

Transformations 4.2, 4.3, and 4.5 involve comparing each pair of nodes at the current level. Since there are N such nodes, the number of such comparisons is N(N-1)/2. For each pair of nodes, we must check and update certain conditions at each level upto the common ancestor. The common ancestor may be at most i levels above the current level. So Transformations 4.2, 4.3 and 4.5 have complexity $O(N^2.i)$. Transformation 4.4 involves checking the bitmap of a node and has complexity $O(1)$. Therefore, for all N nodes, the complexity of $O(N)$.

Therefore the complexity for applying all the transformations at the i^{th} level is $O(N) + O(N^2.i) + O(N) = O(N^2.i) = O(2^{2i}.i)$. Since there are n levels in the 123-DD, the total complexity is $O(n.2^n)$.

4.6.3 "Flipping" of Nodes in the 123-DD

In the 123-DD we keep track of the relative positions of the nodes, as well as the interconnection between them. The positional information is maintained in the form of a position list (section 4.4). Any change in the relative position of the nodes in the 123-DD implies a corresponding modification of the position list. An outline of the procedure implementing this is given below.

The function try_to_make_adjacent(PL_1, N_1, N_2) takes two nodes N_1 and N_2 at the current level, the initial position list PL_1, and returns the modified position list PL_2, after N_1 and N_2 are brought as close together as possible. Initially, before any subgraph merge has taken place, each node in the 123-DD is free to be the left, middle or right child of its parent. When Transformation 4.5 is applied to two nodes (N_1, N_2), these two nodes must be brought as close to each other as possible and subsequent merges should not move them farther apart. Thus, each subgraph merge imposes some additional restrictions on the allowed positions of a node, and these restrictions must be taken into account for subsequent merges. The position list is modified after each application of Transformation 4.5 to reflect these changes.

try_to_make_adjacent(PL_l, N_{k1}, N_{k2})

Find the common ancestor N of N_{k1} and N_{k2} ;

Find the path p1 = [N - e_{01} - N_{01} -e_{11} -N_{11} - ... - e_{k1} - N_{k1}] from N to N_{k1};

Find the path p2 = [N - e_{02} - N_{02} -e_{12} -N_{12} - ... - e_{k2} - N_{k2}] from N to N_{k2};

Make N_{01} and N_{02} adjacent if possible;

IF N_{01} is to the left(right) of N_{02} THEN

FOR i = 1 to k, DO

{

Move N_{i1} as much to the right(left) as possible;

Move N_{i2} as much to the left(right) as possible;

}

Update the position list to reflect this change;

IF all nodes between N_{k1} and N_{k2} in updated position list, are dummy nodes THEN

Remove all dummy nodes between $N_{k1}$and N_{k2} in the updated position list;

RETURN the updated position list;

Example 4.2:

Suppose we want to merge the subgraphs below nodes 27 and 37 in Figure 4.13(a). The position list for this structure is PL_l = {{{27, 28, 29_d}, {30, 31, 32_d}, {33_d, 34_d, 35_d}}, {{36, 37, 38_d}, {39, 40, 41_d}, {42_d, 43_d, 44_d}}, {{45_d, 46_d, 47_d}, {48_d, 49_d, 50_d}, {51_d, 52_d, 53_d}}}.

step 1: We find the common ancestor of nodes 27and 37, which is node 1.

step 2: We find p1 = [1 - e_{01} - 3 - e_{11} - 9 - e_{21} - 27].

We find p2 = [1 - e_{02} - 4 - e_{12} - 12 - e_{22} - 37].

step 3: Nodes 3 and 4 are adjacent in the 123-DD.

step 4(a): We make 9 the right child of 3 by switching the positions of nodes 9 and 10.

We make 27 the right child of 9 by switching the positions of nodes 27 and 28.

step 4(b): Keep node 12 as the left child of node 4.

We make 37 the left child of 12 by switching the positions of nodes 36 and 37.

Now nodes 27 and 37 are adjacent in the 123-DD (Figure 4.13(b)), and we modify PL_1 to reflect the fact that nodes 27 and 37 must always be adjacent to each other. The modified position list is given by PL_2 = {[{{30, 31, 32_d}, {33_d, 34_d, 35_d}},{28, 29_d} 27, 37, {36, 38_d}, {{39, 40, 41_d}, {42_d, 43_d, 44_d}}], {{45_d, 46_d, 47_d}, {48_d, 49_d, 50_d}, {51_d, 52_d, 53_d}}}.

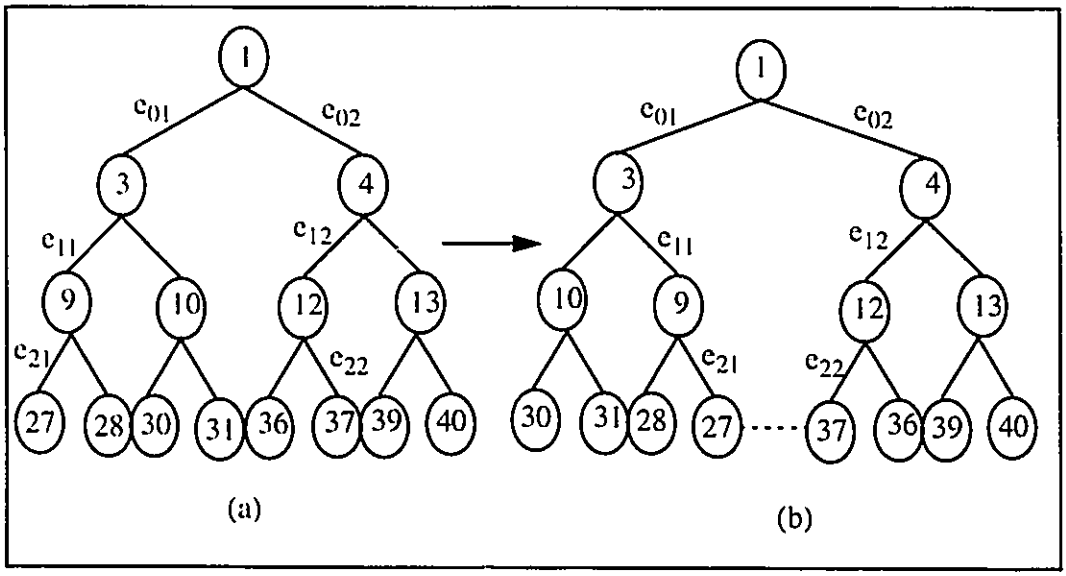

Figure 4.13 : Example of "Flipping" nodes in the 123-DD

4.7 Interesting properties of 123-DD

4.7.1 Sneak Paths in MOS Circuits

Due to the bidirectional nature of MOS transistors, spurious paths from the output to the terminal nodes may be created through the circuit due to merging of subgraphs [JUI.94]. Such spurious paths are called *sneak paths*. The existence of *sneak paths* in a design can lead to incorrect circuit implementations as shown below.

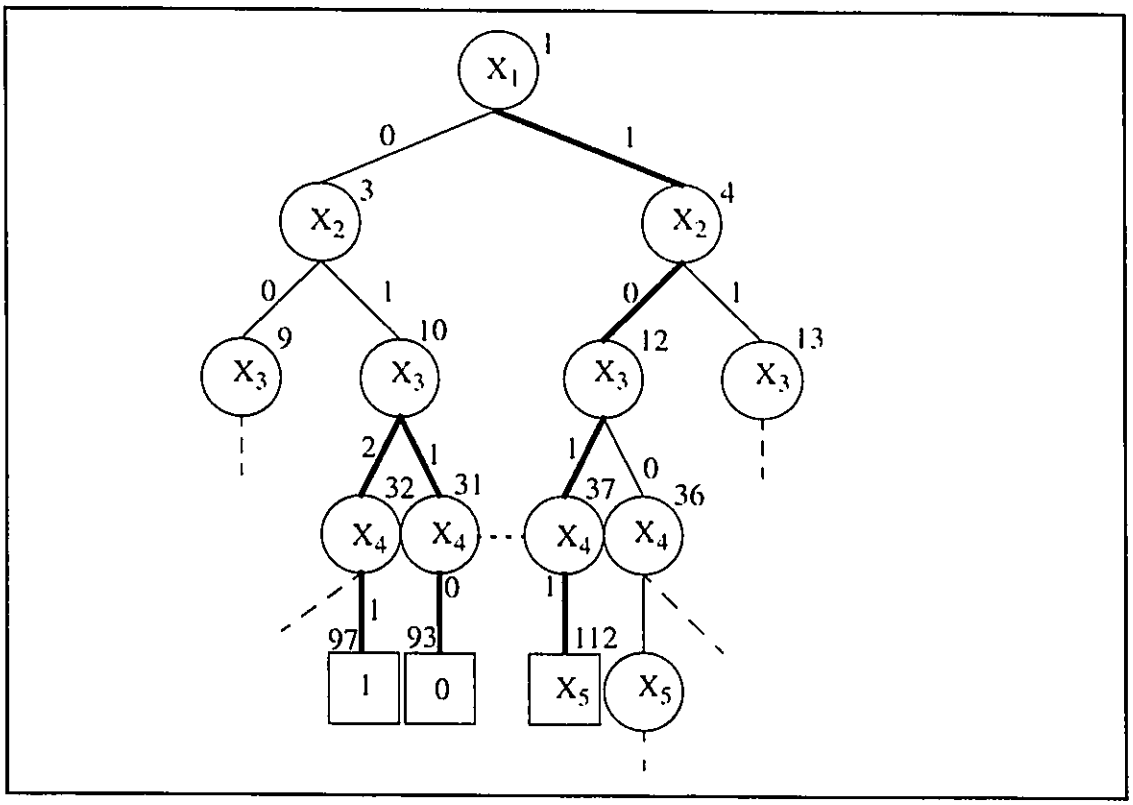

Figure 4.14 : Example of Sneak Path

Figure 4.14 shows a partial 123-DD for a five variable function. In this diagram, nodes 31 and 37, having identical signal strings, are merged. Now, we consider an input condition $X_1 = 1$; $X_2 = 0$; $X_3 = 1$; $X_4 = 1$; $X_5 = 0$; Under these conditions the activated path, from the root to the current level, should be 1->4->12->37->112 and the correct pass signal transmitted to the output should be **0**. However, the merge has also activated a second

incorrect path 1->4->12->37->31->10->32->97 for the same set of inputs. Following this path results in a pass signal **1** being transmitted for the same input.

Detecting the existence of such sneak paths is an inefficient and time consuming process. Therefore, it is desirable to have a design process which guarantees that spurious paths will never be created.

4.7.1.1 Sneak Paths in 123-DDs

In this section we will show that synthesis procedure in section 4.6 can never lead to sneak paths in 123-Decision Diagrams.

Property 4.5: There is always at most one activated path from any node in the 123-DD to the current level.

Proof: We will now show that Property 4.5 is always satisfied by a 123-DD created using the synthesis procedure outlined in this chapter. The proof proceeds in two parts. In the first part we show that a 123-DD synthesized without applying Transformations 4.2 and 4.3 always satisfies Property 4.5. In the second part we show that, a 123-DD which satisfies Property 4.5, continues to do so even after applying Transformation 4.2 or Transformation 4.3.

Part 1: Edges with label 2 are created only as a result of Transformations 4.2 or 4.3. Therefore, the 123-DD created without applying these transformations will only have edges with label 0 or 1. Consider a node with label X_i in such a 123-DD. For any input combination, at most one edge from this node can be activated, connecting it to its child at the next level. By similar reasoning, there is at most one active path from the child to the following level and so on until the *current level* is reached. So there can never be more than one activated path from any node in the 123-DD to the *current level.*

Part 2: Applying Transformation 4.2 or 4.3, results in elimination of two paths p1 = [N - e_{01} - N_{01} -e_{11} -N_{11} - ... - e_{k1} - N_{k1}] and p2 = [N - e_{02} - N_{02} -e_{12} -N_{12} - ... - e_{k2} - N_{k2}] from a node N to the current level and creation of a new path p3 = [N - e_{03} - N_{03} e_{13} -N_{13} - ... - e_{k3} - N_{k3}] (see section 4.5.2). Since the sequence of edge labels form N_{01} N_{02} N_{03} to the current level are identical, path p3 is activated under exactly the same conditions as either path p1 or p2. Since, the 123-DD initially satisfied Property 4.5, there was no other activated path from N to the current level for the conditions under which p1(or p2) was activated. Thus if when p3 is activated, it will be the only activated path from N to the current level. So, the 123-DD will continue to satisfy Property 4.5 after Application of Transformation 4.2 or 4.3.

Property 4.6: The synthesis procedure in section 4.6 guarantees that the final design will be free from sneak paths.

Proof: sneak paths are created when

(i) for a given input combination, there are two (or more) *activated* paths from a node in the 123-DD to two (or more) distinct nodes at the current level and
(ii) these nodes have different signal strings associated with them

So, if the 123-DD always satisfies the following Property 4.5, it ensures that the final implementation will be free of sneak paths.

4.7.2 Performance Guarantee w.r.t. OBDDs

In order to compare the results of synthesizing Boolean functions using 123-Decision Diagrams and Binary Decision Diagrams, we need to make some simplifying assumptions. When synthesizing OBDDs, Transformation 2.3 is always applied when two (or more) nodes at the current level have identical bitmaps. We do not consider whether they may be made adjacent, or any other layout factors before merging. For 123-DDs, on the other hand, merging of subgraphs takes place only after it is found to be feasible. In

order to make a fair comparison between OBDDs and 123-DDs, we will relax our synthesis procedure so that in 123-DD synthesis procedure, we ignore the "feasibility" of a merge. Our assumptions are given below.

(i) When implementing a Boolean function, both the OBDD and the 123-DD use the same variable ordering.
(ii) We are dealing with completely specified functions.
(iii) A design is 'better' than another, if its width is less than the other. The width is defined as the maximum number of nodes at any level.
(iv) At a given level, two nodes are merged as long as they have identical bitmaps(signal strings).

We also note that:

a. Transformations 2.1 and 4.1 are equivalent
b. Transformations 2.3 and 4.5 are equivalent
c. Transformation 2.2 is a special case of Transformation 4.2 where the common ancestor is exactly one level above the current level.
d. Transformation 2.4 is equivalent to part (i) of Transformation 4.4

Therefore OBDDs can be considered to be a special case of 123-DDs. In this section we will consider each transformation which is used only for 123-DDs and not for OBDDs (4.2, 4.3 and 4.4 part ii) and show that these transformations never increase the width of the 123-DD over the corresponding OBDD.

Part 1: Transformation 4.2

We know that Transformation 4.2 reduces the number of nodes at the current level and is applied only if it does not increase the current maximum width of the 123-DD (section 4.5.6.2). Transformation 4.5 can still be applied on the new node at the current level if other node(s) with the same signal string exist. So, Transformation 4.2 does not affect the

number of nodes at subsequent levels. Therefore application of Transformation 4.2 never increases the maximum width of the final 123-DD.

Part 2: Transformation 4.3

This is almost identical to Transformation 4.2 except that the new node at the current level is a terminal node. As in Part 1, this transformation does not increase the current maximum width and reduces the number of nodes at the current level. Also since the new node is a terminal node it is never expanded at subsequent levels and thus can never increase the number of nodes at subsequent levels. Therefore application of Transformation 4.3 never increases the maximum width of the final 123-DD.

Part 3: Transformation 4.4

Transformation 4.4 simply converts a nonterminal node into a terminal node. So, it does not affect the number of nodes at previous levels or at the current level. Again, since a terminal node is not expanded at subsequent levels, Transformation 4.4 can never increase the number of nodes at lower levels either. Therefore application of Transformation 4.4 never increases the maximum width of the final 123-DD.

4.8 Example of PTL Synthesis with 123-DD

In this section, we will take a simple example and synthesize a 123-DD for a specific function following the steps given in section 4.6.1. We consider a five variable Boolean function F = 0000 1001 0000 0011 1111 1001 1111 1000 specified as a bitmap of 32 bits. This bitmap is obtained after reordering the input variables in step 1. Figure 4.15 shows the 123-DD at various stages in the synthesis procedure.

Figure 4.15(a) shows the root node with the entire function F as the signal string associated with it. Figure 4.15(b) shows the 123-DD after application of Transformation

4.1 to the root node to generate its children, nodes 3 and 4. Figure 4.15(c) shows the 123-DD expanded upto the fourth level after applying Transformation 4.1 to each node in levels 2 and 3. The List Structure corresponding to it is $PL_1 = \{\{\{27, 28, 29_d\}, \{30, 31, 32_d\}, \{33_d, 34_d, 35_d\}\}, \{\{36, 37, 38_d\}, \{39, 40, 41_d\}, \{42_d, 43_d, 44_d\}\}, \{\{45_d, 46_d, 47_d\}, \{48_d, 49_d, 50_d\}, \{51_d, 52_d, 53_d\}\}\}$.

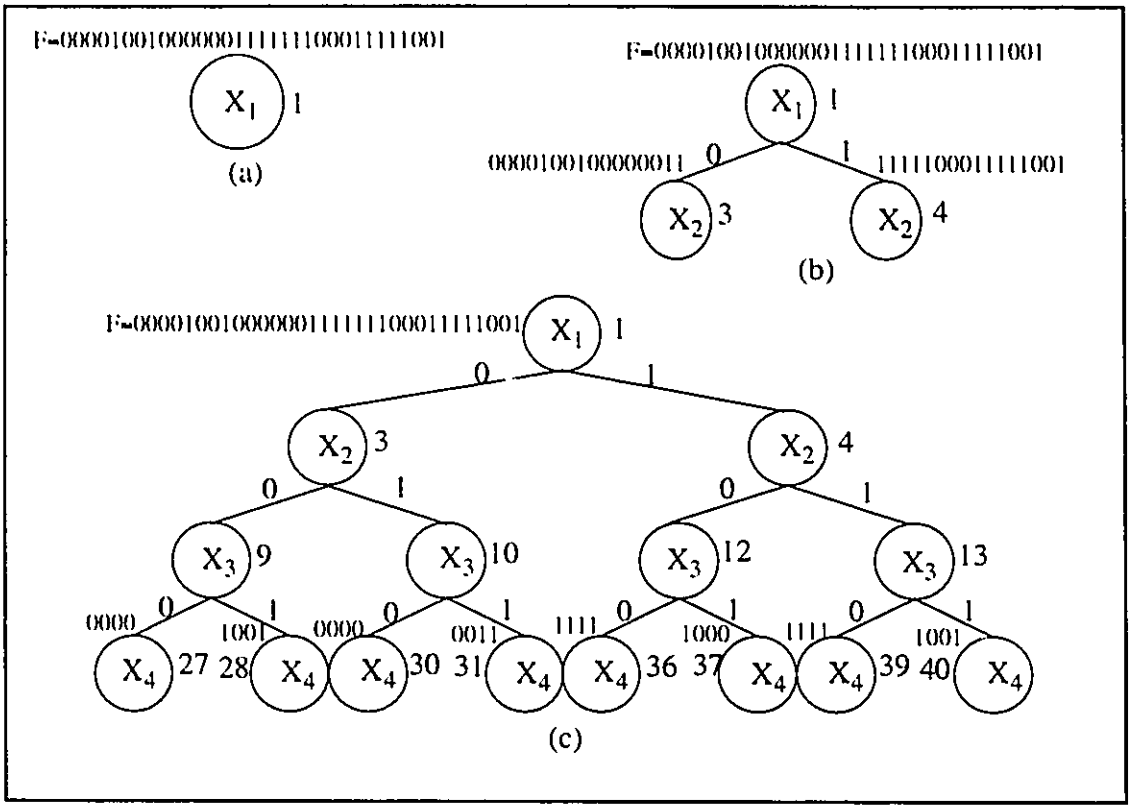

Figure 4.15 : Example of Synthesis Procedure.

We note that at this point nodes 27 and 30 have identical signal strings, as do nodes 36 and 39. We now check and find that the conditions for applying Transformation 4.2 are satisfied in both cases. So, Transformation 4.2 is applied to both sets of nodes and the two new nodes 33 and 42 are added to the current level. The state of the 123-DD at this point is shown in Figure 4.15(d).

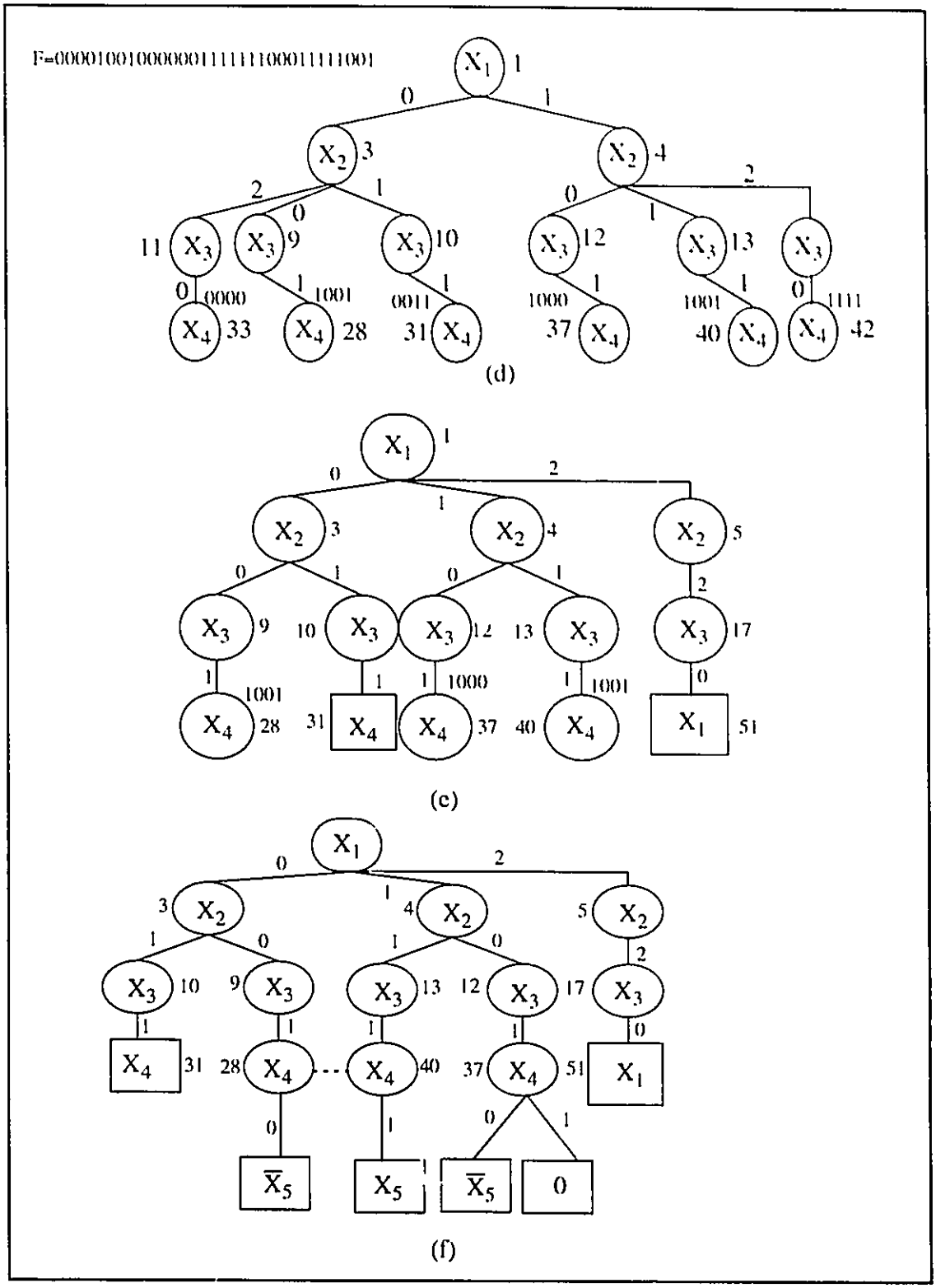

Figure 4.15: Example of Synthesis Procedure.

We now see that Transformation 4.3 can be applied to nodes 33 and 42 and Transformation 4.4 can be applied to node 31 to convert it into a terminal node. Figure 4.14(e) shows the 123-DD after applying these transformations.

Finally, we note that nodes 28 and 40 have identical signals strings and we can apply Transformation 4.5. However, since they are not adjacent to each other, we make them adjacent, by switching the positions of nodes 9 and 10 and nodes 12 and 13, before applying Transformation 4.5. The updated position list now becomes $PL_2 = [\{ \{31, 32_d\}, \{34_d, 35_d\}\}, 29_d, 28, 40, 41_d, \{\{37, 38_d\}, \{43_d, 44_d\}\}, \{\{45_d, 46_d, 47_d\}, \{48_d, 49_d, 50_d\}, \{51, 52_d, 53_d\}\}]$.

Finally, we apply Transformation 4.1 to the nonterminal nodes at level 4 and generate the nodes at the fifth level. All these nodes can be converted to terminal nodes using Transformation 4.4. So, the synthesis procedure ends at this point. The final 123-DD for the entire function is shown in Figure 4.15(f).

4.9 Conclusions

In this chapter we have discussed a new model for representing PTL networks. This model is based on the concept of decision diagrams. We have also described some transformation rules for manipulating our model and synthesizing actual PTL networks. The results of our synthesis algorithm compared to existing synthesis techniques are given in the next chapter.

Chapter5

Experimental Results

5.1 Introduction

In this chapter we discuss the results of our experiments in synthesizing PTL networks using the methodologies outlined in chapters 3 and 4. We have tested both techniques on benchmark circuits and have performed simulations on the synthesized circuits. In section 5.3 and 5.4, we have presented the results for the factorization approach and the decision diagram based approach.

Our synthesis procedure is for single output functions. In order to test these heuristics on benchmark circuits, we need to obtain a network of single output functions corresponding to each of the benchmark circuits. We have used the Boolean Network [BRA88] approach when testing our synthesis procedure on benchmark circuits. A Boolean network consists of a set of interconnected nodes, each representing a smaller single output function. The area complexity of the network is the sum over all nodes of the area complexity of each node. The concept of Boolean networks is explained in detail in section 5.2.

In our approach we address this problem of synthesizing multi-level pass transistor logic networks using our heuristics, for each of

the individual single output functions in a Boolean network. To determine the efficiency of our approach, we have compared our results to those obtained by implementing each node as a two-level PTL network, using standard synthesis techniques [RAD85].

We have used the area of the array of transistors to estimate the *cost* of a circuit. This area for a single output circuit is estimated by the product N * I where N is the number of columns of transistors and I is the number of inputs to the circuit. The cost of a multi-output circuit is determined by computing the sum of the areas of all the single output functions in the Boolean Network corresponding to the multi-output circuit.

5.2 Boolean Networks

A Boolean network is a technology independent multi-level logic structure which can be used to represent multi-output Boolean functions. A large multi-output function is decomposed into a number of smaller functions, which are then interconnected to realize the original function. We have used a multi-level logic optimization system, MIS [BRA87], to synthesize the Boolean network. The following example illustrates how Boolean networks are used to represent logic functions.

Example 5.1

We consider the function cm162a from the MCNC benchmarks. This function has 14 primary inputs {a, b, c, d, e, f, g, h, i, j, k, l, m, n} and five primary outputs {o, p, q, r, s}, defined below.

$$o = cdejn + \bar{a}cejn + cde\dot{i} + d\bar{e}i + \bar{a}\bar{e}i + \bar{c}di + \bar{a}\bar{c}i + \bar{a}\bar{d} + \dot{f}$$

$$p = cdejn + \bar{b}cejn + cde\dot{i}\bar{k} + c\dot{i}k + \bar{b}ik + d\bar{e}k + \bar{b}\bar{e}k + \bar{c}dk + \bar{b}\bar{c}k + \bar{b}\bar{d} + \dot{f}$$

$$q = cde\dot{i}\bar{k}l + ce\bar{g}jn + cdejn + \bar{g}kl + dkl + \bar{g}il + dil + \bar{e}\bar{g}l + \bar{c}\bar{g}l + d\bar{e}l + \bar{c}dl + \bar{d}\bar{g} + \dot{f}$$

$$r = cde\bar{i}\bar{k}l\bar{m} + ce\bar{h}jn + cdejn + \bar{h}lm + dlm + \bar{h}km + dkm + \bar{h}im + dim + \bar{e}\bar{h}m + \bar{e}\bar{h}m + \bar{d}\bar{h} + \bar{f} + d\bar{e}m + \bar{e}dm$$

$$s = ejn$$

The initial Boolean network (Figure 5.1) consists of five distinct nodes, each implementing one primary output.

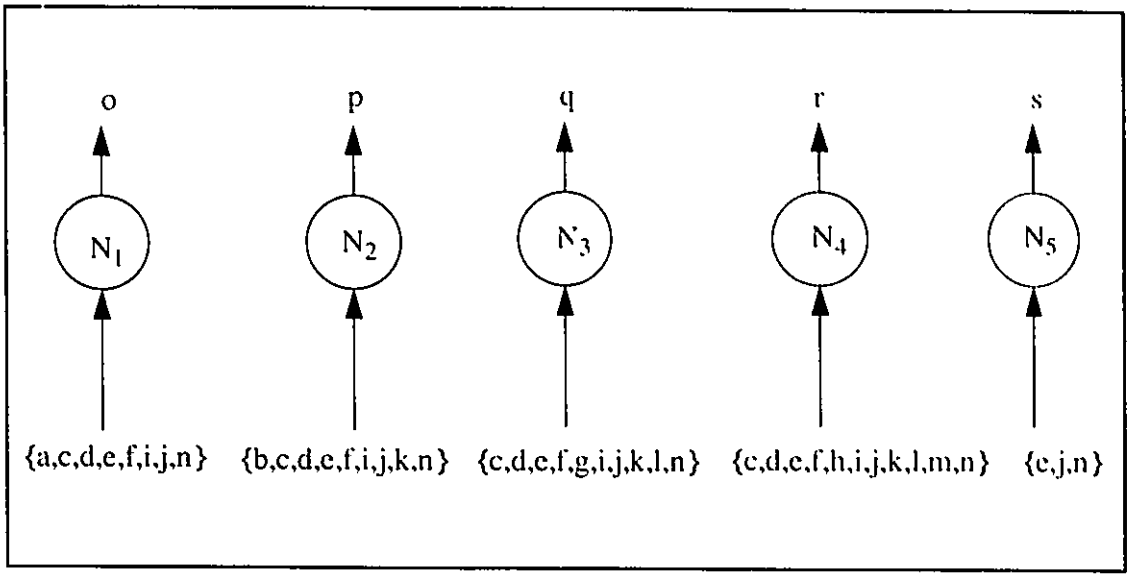

Figure 5.1 : Initial Boolean network for cm162a.

We note that q and r are functions of 10 and 11 inputs respectively. To enhance performance, we might want to decompose a large functions into a number of smaller function. We use the MIS program to extract common sub-expressions and create intermediate nodes. The primary outputs are then re-expressed in terms the intermediate inputs as well as some of the primary inputs. For example, we could create intermediate functions y, a_0 and h_0 and redefine the primary outputs in terms of these, as shown below.

$$y = \bar{d}fh_0 + \bar{d}f\bar{n} + \bar{d}\bar{f}j$$

$$a_0 = df h_0 + df\bar{n} + dfj$$

$$h_0 = \bar{e} + \bar{d} + \bar{c}$$

$$o = i\bar{y}\bar{h}_0 + \bar{a}i\bar{h}_0 + i\bar{y}h_0 + \bar{a}ih_0 + \bar{y}\bar{a}_0 + \bar{a}\bar{a}_0$$

$$p = ik\bar{y}\bar{h}_0 + \bar{b}ik\bar{h}_0 + k\bar{y}h_0 + \bar{b}kh_0 + ik\bar{y} + \bar{b}ik + \bar{y}\bar{a}_0 + \bar{b}\bar{a}_0$$

$$q = ikl\bar{y}\bar{h}_0 + \bar{g}ikl\bar{h}_0 + l\bar{y}h_0 + \bar{g}lh_0 + kl\bar{y} + il\bar{y} + \bar{g}kl + \bar{g}il + \bar{y}a_0 + \bar{g}\bar{a}_0$$

$$r = ikl\bar{m}\bar{y}\bar{h}_0 + \bar{h}ikl\bar{m}\bar{h}_0 + m\bar{y}h_0 + \bar{h}mh_0 + lm\bar{y} + km\bar{y} +$$
$$im\bar{y} + \bar{h}lm + \bar{h}km + \bar{h}im + \bar{y}\bar{a}_0 + \bar{h}\bar{a}_0$$

$$s = ejn$$

The Boolean network corresponding to this decomposition is shown in Figure 5.2. To illustrate our approach, we consider the synthesis of one node using our heuristics. In the above network, the node N_1, realizes the output o where

$$o = i\bar{y}\bar{h}_0 + \bar{a}i\bar{h}_0 + i\bar{y}h_0 + \bar{a}ih_0 + \bar{y}\bar{a}_0 + \bar{a}\bar{a}_0$$

If we realize this function using a 2-level PTL network [RAD85] we get

$$o = a_0\bar{y}\bar{h}_0(i) + a_0\bar{y}h_0(i) + a_0\bar{a}h_0(i) + \bar{a}a_0\bar{h}_0(i) + \bar{y}\bar{a}_0(1) + y\bar{a}_0(\bar{a}) + ay(0)$$

This requires seven columns (prime pass implicants) to implement and the area needed for this network is 7 X 5 = 35. The multilevel representation for the same function is: $o = \bar{a}_0\bar{y}(1) + \bar{a}_0 y(\bar{a}) + ay(0) + [\bar{h}_0(i) + h_0(i)]\,(a_0\bar{a} + \bar{y}a_0)$. This requires only 5 columns to implement and the area needed is 5 X 5 = 25.

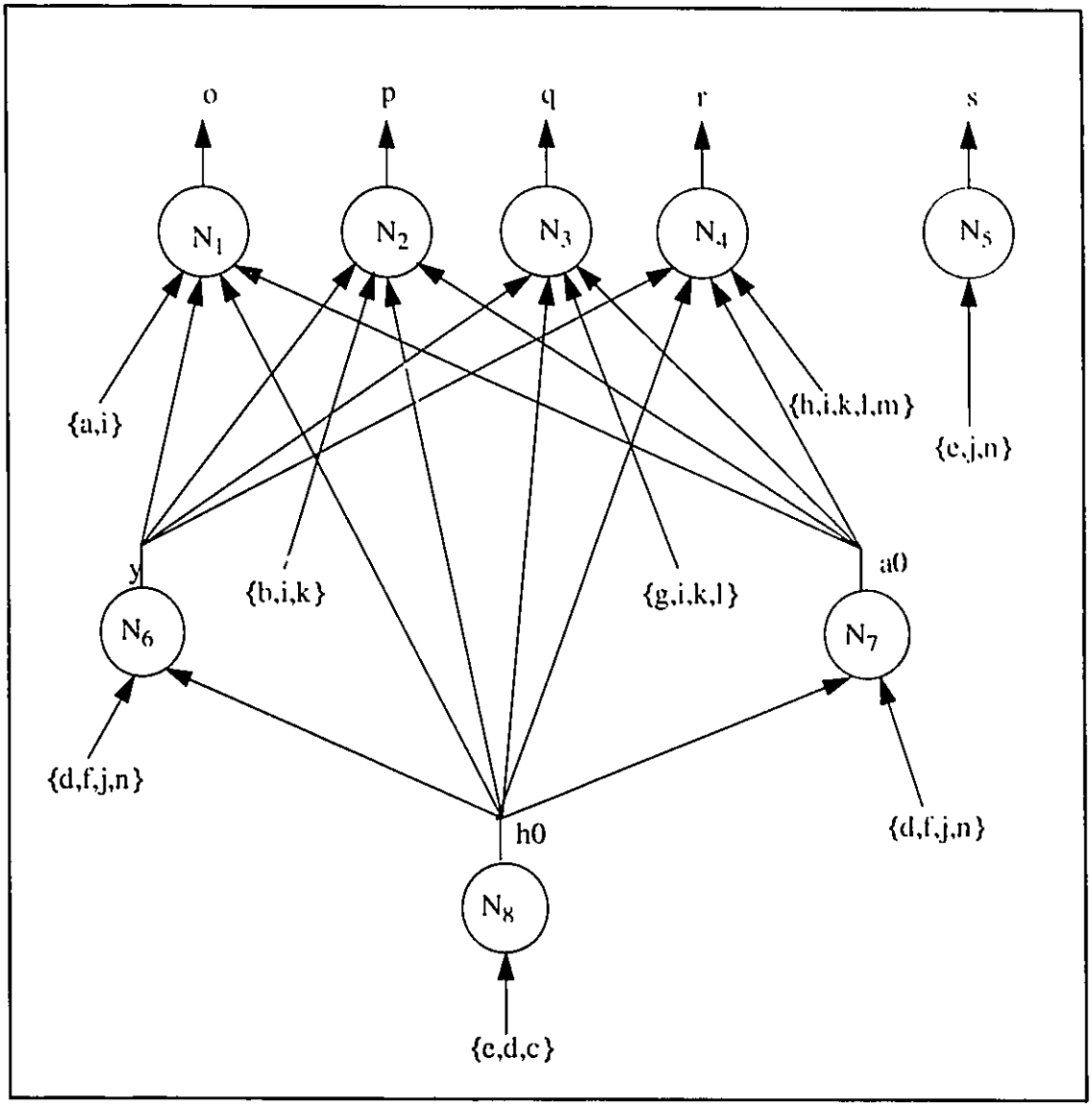

Figure 5.2 : A decomposed Boolean network for the function cm162a.

Each node in the Boolean network is implemented in a similar way, as a PTL network. The results for the individual nodes are shown in Table 5.1.

Table 5.1: Detailed Area Comparison for Circuit cm162a

Node	Number of Inputs	Number of Columns in		Area for	
		2-level Form	Multi-level Form	2-level Form	Multi-level Form
N_1	5	7	5	35	25
N_2	6	9	5	54	30
N_3	7	11	5	77	35
N_4	8	13	5	104	40
N_5	3	3	3	9	9
N_6	5	5	3	25	15
N_7	5	5	3	25	15
N_8	3	3	3	9	9
TOTAL	14	46	32	308	178

5.3 Factorization Approach

5.3.1 Area Requirements for Benchmark Circuits

Table 5.2 shows the cost comparison for a number of MCNC benchmark circuits synthesized using the factorization approach and the traditional two-level minimization techniques.

Table 5.2: Area Comparisons for Benchmark Circuits

Name of Circuit	Number of Inputs	Number of Outputs	Cost for		Percentage Improvement
			2-level Implementation	Factorized Circuit	
count	8	4	808	648	19.8
alu2	10	6	1902	1686	11.35

Name of Circuit	Number of Inputs	Number of Outputs	Cost for		Percentage Improvement
			2-level Implementation	Factorized Circuit	
C17	5	2	36	28	22.22
comp	32	3	2244	1796	19.96
decod	5	15	400	400	0
major-ity	5	1	40	35	12.5
my_adder	33	17	1773	1255	29.21
parity	16	1	1248	1068	14.42
pcle	19	9	550	398	27.63
unerg	36	12	384	384	0
z4ml	7	4	422	262	37.91
sct	19	15	682	596	14.43
cu	14	11	381	378	0.78
pm1	16	13	410	410	0
cmb	16	4	245	238	2.85
C1355	41	32	1824	1336	25.90
C1908	33	25	2093	1895	9.46
C5315	178	123	6219	5836	6.15
apex7	49	37	1779	1396	21.52
x3	135	99	4909	4555	7.21
x4	94	71	8969	6478	27.77
dsip	228	197	532	397	25.37
mm30a	33	30	5113	4295	16
mm4a	7	4	707	531	33.14
mm9a	12	9	1405	1175	16.37
mult16a	17	1	488	488	0
sbc	40	56	814	666	18.18
C2670	233	140	2877	2454	14.7
b9	41	21	1014	850	16.17
cc	21	20	348	334	4.02
cht	47	36	508	508	0
cm151a	12	2	46	46	0

Name of Circuit	Number of Inputs	Number of Outputs	Cost for		Percentage Improvement
			2-level Implementation	Factorized Circuit	
cm152a	11	1	160	160	0
cm 162a	14	5	338	228	32.54
cm163a	16	5	225	225	0
cm82a	5	3	122	97	20.49
cm85a	11	3	856	598	30.14
exampl e2	85	66	1673	1377	17.69
f51m	8	8	477	402	15.72
lal	26	19	901	704	21.86
i1	25	16	187	187	0
pcler8	27	17	562	400	28.82
term1	34	10	3414	2709	20.65

5.3.2 Effect of Function Size on Area Reduction

As mentioned in chapter 3, a factored form must satisfy a number of specific conditions in order to be included in the final cover. If the number of prime pass implicants for a function is small, it is difficult to find good factored forms from the set. As the number of prime pass implicants increases, there are more choices available and it is more likely that a good factored form can be found. To demonstrate this, we have taken the individual single output functions from all the benchmarks and ordered them in terms of the number of input variables. Table 5.3 shows the average improvement for a function of n inputs, where n varies from 2 to 10. As expected, for relatively small functions (2-3 inputs), the savings obtained by the multilevel representation are negligible, since suitable factored forms usually could not be found. But, as the number of inputs increases, the savings become more significant.

Table 5.3: Effect of Function Size on Area Improvement

Total number of Functions	Number of Inputs	Total Number of Columns in		Percentage Improvement
		2-level implementation	Factorized Implementation	
572	2	1143	1143	0
465	3	1391	1376	1.08
344	4	1384	1273	8.02
326	5	1909	1667	12.68
202	6	1351	1103	18.36
101	7	873	704	19.36
44	8	798	637	20.18
66	9	874	719	17.73
39	10	1088	776	28.68

5.3.3 Time Requirements for Factorization Algorithm

Our factorization algorithm is fairly complex. We have discussed in section 3 why it was necessary to start with a set of all prime pass implicants and form quotients with selected bi-cubic divisors. If we compare this approach to the standard factorization algorithms, where only a minimum cover is used for all trial divisions, our approach may seem to be computationally infeasible. We now show that the time needed to factorize using our heuristic is comparable to the time needed for a 2-level synthesis. The key observation is that selecting a good two-level cover is itself known to be a NP-complete problem. We find that although factorization takes slightly longer for larger functions, the time requirements for both processes are comparable. Table 5.4 shows the timing requirements for generating a two-level cover and a factorized expression for functions of 5 to 10 inputs.

Table 5.4: Time Required for Synthesis

Number of Inputs	Average Time (s) Required to Generate		
	All PIs	2-level Cover	Factorized Form
5	0.05	0.043	0.05
6	0.050	.071	0.085
7	0.16	0.21	0.25
8	0.86	1.01	1.3
9	7.69	4.28	5.88
10	41.44	57.93	85.09

5.4 Decision Diagram Approach

5.4.1 Area Requirements for Benchmark Circuits

In this section we compare the results of two-level PTL synthesis techniques with those obtained from the decision diagram based approach (chapter 4). When synthesizing this type of circuits, the number of nonlocal connections at each level has some effect on the number of columns. For the layout strategy we have used (section 4.2.2) we can have one nonlocal connection at no cost. With this layout strategy, we now carry out our area experiments in a manner similar that used in Table 5.2.

Table 5.5 gives the results of our synthesis algorithm assuming at most one nonlocal line per level.

Table 5.5: Area Comparisons for Benchmark Circuits

Name of Circuit	Number of Inputs	Number of Outputs	Cost for		Percentage Improvement
			2-level Implementation	123-DD Based Circuit	
count	8	4	808	512	36.63
alu2	10	6	1902	1356	28.70
C17	5	2	36	28	22.22

Name of Circuit	Number of Inputs	Number of Outputs	Cost for		Percentage Improvement
			2-level Implementation	123-DD Based Circuit	
comp	32	3	2244	504	77.54
decod	5	15	400	400	0
major-ity	5	1	40	25	37.5
my_adder	33	17	1773	580	67.28
parity	16	1	1248	72	94.23
pcle	19	9	550	384	30.18
unerg	36	12	384	384	0
z4ml	7	4	422	116	72.51
set	19	15	682	444	34.89
cu	14	11	381	374	1.83
pm1	16	13	410	302	26.34
cmb	16	4	245	211	13.87
C1355	41	32	1824	928	49.12
C1908	33	25	2093	1625	22.36
C5315	178	123	6219	5404	13.10
apex7	49	37	1779	1071	39.79
x3	135	99	4909	3866	21.24
x4	94	71	8969	2775	69.06
dsip	228	197	532	305	42.67
mm30a	33	30	5113	2947	42.36
mm4a	7	4	707	356	49.64
mm9a	12	9	1405	793	43.55
mult16a	17	1	488	488	0
sbc	40	56	814	407	50
C2670	233	140	2877	2114	26.42
b9	41	21	1014	583	42.5
cc	21	20	348	297	14.65
cht	47	36	508	508	0
cm151a	12	2	46	46	0
cm152a	11	1	160	160	0

Name of Circuit	Number of Inputs	Number of Outputs	Cost for		Percentage Improvement
			2-level Implementation	123-DD Based Circuit	
cm 162a	14	5	338	180	46.74
cm163a	16	5	225	175	22.22
cm82a	5	3	122	62	49.18
cm85a	11	3	856	181	78.85
exampl e2	85	66	1673	1049	37.3
f51m	8	8	477	274	42.55
lal	26	19	901	413	54.16
i1	25	16	187	183	2.14
pcler8	27	17	562	351	37.54
term1	34	10	3414	2090	38.78

Since the number of allowable nonlocal connections may vary with technology as well as the layout strategy, we have also conducted several different experimental runs, each allowing a different number of nonlocal connections between levels. The results of all these runs are summarized below, in Table 5.6.

Table 5.6: Effect of Nonlocal Connections on Area

Name of Circuit	Percentage improvement of 123-DD over 2-level implementation with (n) nonlocal connections allowed per level				
	n = 0	n = 1	n = 2	n = 3	n = 5
count	15.84	36.63	42.57	43.56	43.56
alu2	17.24	28.70	32.91	35.85	40.37
C17	22.22	22.22	22.22	22.22	22.22
comp	75.93	77.54	77.54	77.54	77.54
decod	0	0	0	0	0
majority	37.5	37.5	37.5	37.5	37.5
my_adder	65.25	67.28	68.30	68.30	68.30
parity	84.29	94.23	94.23	94.23	94.23
pcle	28.9	30.18	30.18	30.18	30.18

Name of Circuit	Percentage improvement of 123-DD over 2-level implementation with (n) nonlocal connections allowed per level				
	n = 0	n = 1	n = 2	n = 3	n = 5
unerg	0	0	0	0	0
z4ml	72.51	72.51	72.51	72.51	72.51
sct	11.43	34.89	34.89	34.89	34.89
cu	1.83	1.83	1.83	1.83	1.83
pm1	12.68	26.34	26.34	26.34	26.34
cmb	13.87	13.87	13.87	13.87	13.87
C1355	33.33	49.12	49.12	49.12	49.12
C1908	11.42	22.36	22.36	22.36	22.36
C5315	10.58	13.10	13.26	13.26	13.26
apex7	31.25	39.79	39.79	39.79	39.79
x3	11.49	21.24	21.24	21.24	21.24
x4	51.02	69.06	69.13	69.13	69.13
dsip	33.27	42.67	42.67	42.67	42.67
mm30a	0.68	42.36	42.36	42.36	42.36
mm4a	32.10	49.64	49.64	49.64	49.64
mm9a	1.0	43.55	43.55	43.55	43.55
mult16a	0	0	0	0	0
sbc	34.4	50	50	50	50
C2670	26.42	26.42	26.42	26.42	26.42
b9	29.1	42.5	44.28	44.28	44.28
cc	10.92	14.65	14.65	14.65	14.65
cht	0	0	0	0	0
cm151a	0	0	0	0	0
cm152a	0	0	0	0	0
cm 162a	39.05	46.74	48.81	48.81	48.81
cm163a	11.55	22.22	22.22	22.22	22.22
cm82a	49.18	49.18	49.18	49.18	49.18
cm85a	62.5	78.85	78.85	78.85	78.85
example2	33.05	37.3	37.3	37.3	37.3
f51m	33.33	42.55	45.91	45.91	45.91
lal	42.28	54.16	54.16	54.16	54.16
i1	2.14	2.14	2.14	2.14	2.14

Name of Circuit	Percentage improvement of 123-DD over 2-level implementation with (n) nonlocal connections allowed per level				
	n = 0	n = 1	n = 2	n = 3	n = 5
peler8	33.45	37.54	37.54	37.54	37.54
term1	31.13	38.78	38.78	38.78	38.78

As we can see from Table 5.6, even PTL networks with only local connections show considerable improvement over two level implementations. Allowing one nonlocal connection per level significantly reduces the area of the PTL network, in many cases, over that obtained with only local connections. As we further increase the number of nonlocal lines per level, the improvements obtained in most cases are not very significant. Only two of the benchmark circuits, **alu2** and **count**, show any significant improvement as the number of nonlocal lines is increased from 1 to 5.

5.4.2 Timing Requirements for Synthesis Using 123-DD

In the two-level synthesis of PTL networks, there are two main steps involved. The first is to generate the set of all prime pass implicants for a Boolean function, and the second is to choose a suitable cover from this set [RAD85]. Both of these are fairly complex, time-consuming tasks. Synthesis of PTL networks with 123-DDs, takes less time and gives better results. Table 5.7, shows a comparison of the average time required to synthesize PTL circuits using both methods.

Table 5.7: Timing Requirements of 2-level and 123-DD Synthesis

Number of Inputs	Average Time (s) Required to Generate	
	2-level Network	123-DD based Network
5	0.093	0.035
6	0.121	0.05
7	0.37	0.1
8	1.87	0.21
9	11.97	0.28
10	99.37	2.87

5.5 Simulation Results

Figure 5.3 shows the schematic for the MSB of a mod7 multiplier. Latches are placed at each output stage for pipelining purposes. We have used a true single phase clocking(TSPC) scheme [YUA87], [YUA89]. The schematic of the true single phase latch is shown in Figure 5.4

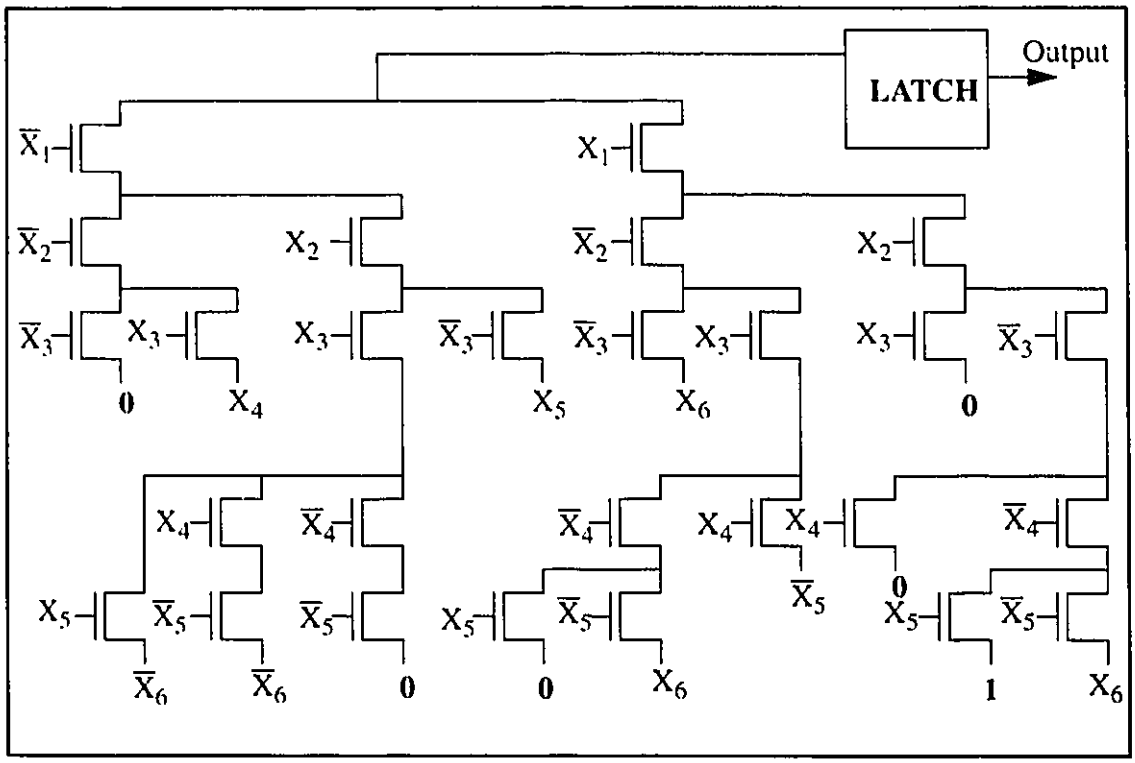

Figure 5.3 : Schematic of MSB of mod7 Multiplier

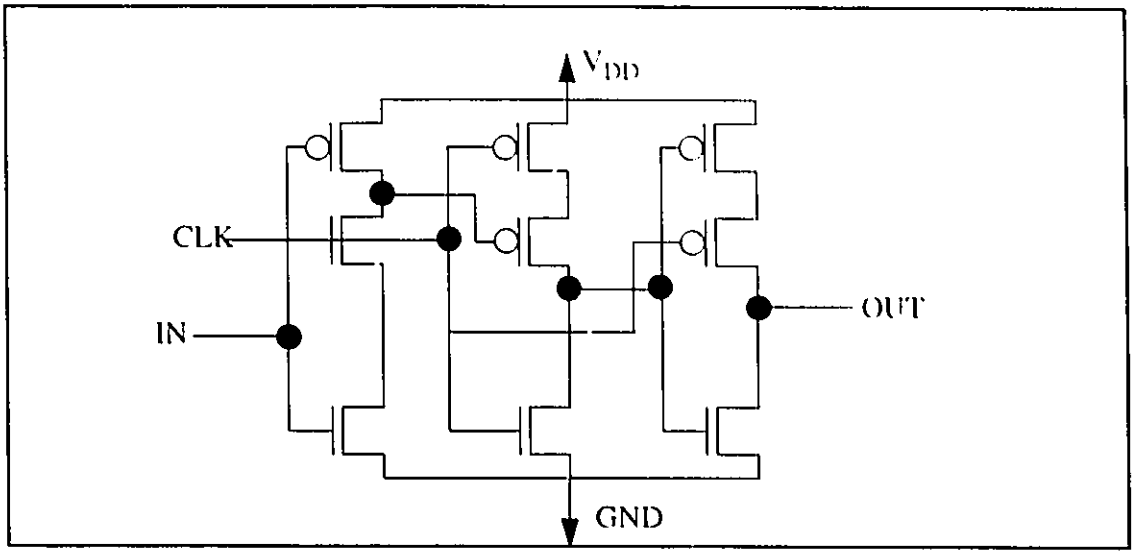

Figure 5.4 : Schematic of the Single Phase Latch

The PTL network implementing the mod7 multiplier was synthesized using the algorithm outlined in section 4.6. Simulations run on the circuit, with an output load of 100ff show that it performs satisfactorily at clock speeds of 140MHz. This is comparable to the performance of NMOS logic blocks for dynamic pipelined systems, synthesized using conventional logic [SID94]. [SID94] shows that NMOS blocks implemented in 1.2μ technology, with a maximum of six transistors in series, can run at speeds of up to 142MHz. Logic blocks with a maximum of 10 transistors in series were found to perform satisfactorily at speeds of 100MHz.

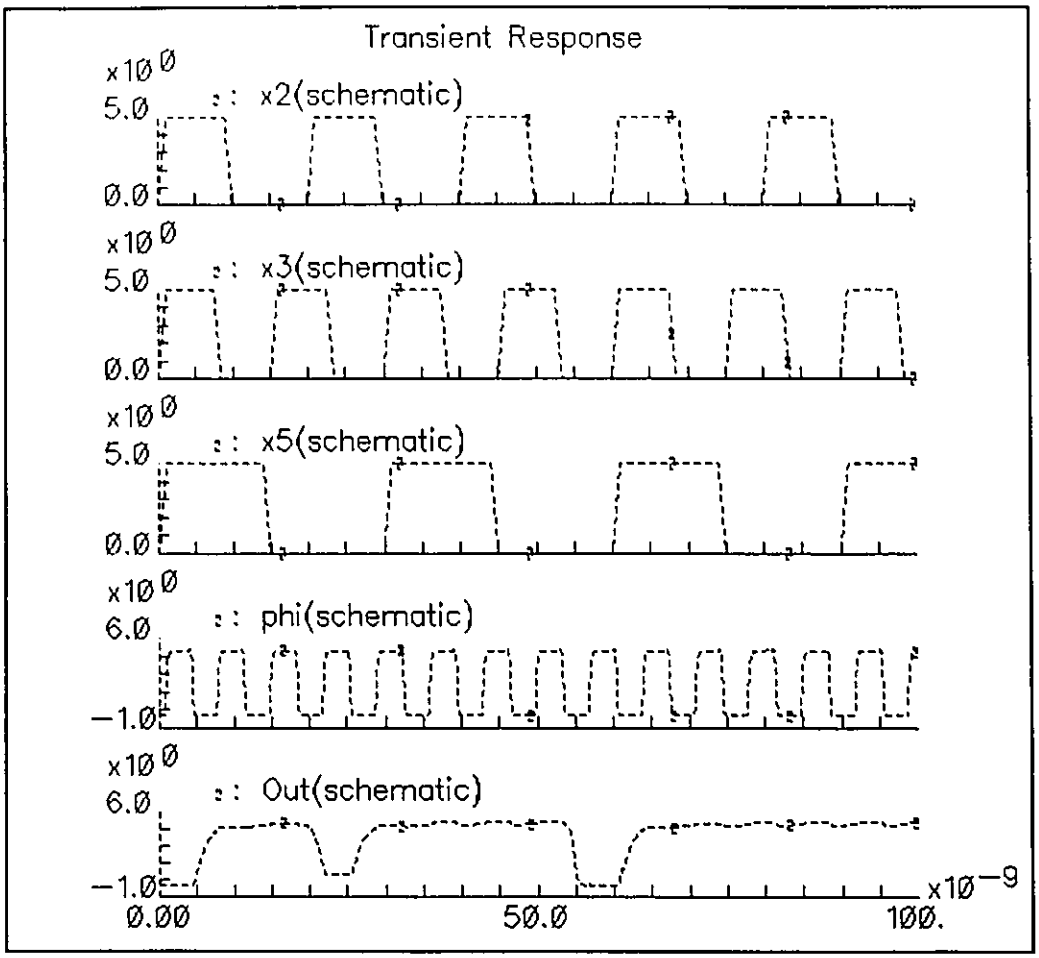

Figure 5.5 : Simulation Results for mod7 multiplier.

5.6 Conclusions

In this chapter we have reported on the results of synthesis experiments for multilevel PTL networks. Both the factorization and decision diagram based approaches resulted in significant improvements over two-level design techniques.

Chapter6

Conclusions and Future Work

This thesis deals with the development of methodologies for the systematic design of multi-level PTL networks. Existing PTL design techniques such as those discussed in [RAD85] and [PED88] are limited to two-level PTL networks. In conventional logic design, it is well known that multi-level synthesis techniques provide considerable savings over two-level design. It is reasonable to expect that similar savings can be obtained by applying multi-level logic synthesis techniques to PTL design. In this thesis, we have investigated two such multi-level logic synthesis techniques for PTL networks. Both approaches have shown significant savings over known synthesis techniques for PTL networks.

6.1 Conclusions

The first approach we have looked at is based on the concept of algebraic factorization. In this approach we have adapted conventional factorization techniques to synthesize PTL networks. Several modifications were required for this in order to satisfy restrictions for PTL circuits. In conventional factorization, selecting a good two-level cover is followed by factorizing the selected cover. We found that this approach does not work in our case due to a number of restrictions in PTL circuits. In our

approach, we have integrated these two steps (selecting a good two-level cover and factorizing the selected cover) and we have obtained average improvements of 15% - 20% over two-level PTL designs for practical sized circuits (upto 10 variables). The time required to obtain the factorized circuits was comparable to that for two-level circuits.

The second approach in this thesis is based on the concept of decision diagrams. We introduced a new model, the 123-Decision Diagram, for representing PTL. This model is an extension of the binary decision diagram and includes a second data structure to keep track of transistor placements. This approach is interesting for two reasons.

- Our synthesis technique gives dramatic improvements (over 80%) for a number of benchmark circuits and an average of 30% improvements in the remaining cases.
- Our model may be mapped directly into a VLSI layout. We keep track of transistor placements during the process of network synthesis itself and our synthesis decisions are guided by layout considerations (e.g., technology used).

In our investigations, we looked at these two models independently of one another. Both of these models offer significant improvements over existing techniques. If we compare these two models, it is obvious that the decision diagram approach is superior in terms of number of columns needed, time to synthesize a network, and ability to handle don't cares.

6.2 Future Work

In chapter 3, we have discussed a heuristic for factorizing Boolean expressions for PTL implementation. Additional heuristics may be developed for this purpose. Heuristics for Boolean factorization would be of particular interest.

The most interesting aspect of this investigation is the fact that we use a model (the 123-DD) which allows us to develop layout driven logic synthesis techniques leading to very compact layouts for some benchmark circuits and good improvements in others. The

computation time needed for our synthesis heuristics is considerably less than the time for conventional synthesis. We feel that more investigation on reducing the delay and power in multilevel PTL logic networks is likely to be fruitful. Some possible approaches are given below :

Since the practical limit of these multilevel techniques will be determined by the maximum number of transistors that may lie in a series path, a more practical and useful cost measure taking into account the area of buffers and amplifiers may be used. This would help in developing CAD tools that integrate these algorithms into the design .

Our data structures make it easy for us to keep track of transistors and their placements, calculating parameters such as the number of transistors in series may be done on the fly, and transistor sizing may be done during this step. Future heuristics should include these aspects

To increase speed, a single function can be further decomposed into a number of smaller functions. The Boolean Network analyzer may be invoked by the synthesis program do this.

A major portion of the time required in our 123-DD approach used to find the best ordering. Investigation of better heuristics to find an optimum variable ordering is desirable.

References

[ABA86] M.S. Abadir and H.K. Reghabati, "Functional test generation for digitalcircuits described using binary decision diagram," *IEEE Trans. Comput.*, vol. C-35, no. 4, pp. 375-379, Apr. 1986.

[AFG90] M. Afghani and C.A. Svensson, "A unified single pahse clocking scheme for VLSI systems", *IEEE J. Solid-State Circuits*, SC-25, pages 255-233, Feb. 1990.

[AKE78] S.B. Akers, "Binary decision diagrams," *IEEE Trans. Comput.*, vol. C-27, no. 6, p. 509-516, June 1978.

[ALA91] W. Al-Assadi et al. "Pass-transistor logic design," *International J. Electronics*, 70:739-749, 1991.

[ASH91] P. Ashar, S. Devadas, and K. Keutzer, "Testability properties of multi-level logic networks derived from binary decision diagrams," *Proc. Santa Cruz Conf. Advanced Res. VLSI*, Mar. 1991, pp. 35-54.

[BER89] C.L. Berman, "Ordered binary decision diagrams and circuit structures," *Proc. IEEE Int. Conf. Comput. Design: VLSI in Comput.*, 1989, pp. 392-395.

[BRA82] R.K. Brayton and C. McMullen, "The decomposition and factorization of Boolean functions," *Proceedings of the International Symposium on Circuits and Systems*, 1982.

[BRA84] R.K. Brayton, "Factoring Logic Functions," *IBM Journal of Res. and Development.*, pages 187-198, 1984.

[BRA87] R.K. Brayton et al. "MIS: A multiple logic optimization system," *IEEE Trans. on Computer-Aided Design*, 6:1062-1081, 1987.

[BRA87b] R.K. Brayton, "Algorithms of multilevel logic synthesis and optimization," *Design Systems for VLSI Circuits - Logic Synthesis and Silicon Compilation*. Martinus-Nijhoff Publishers, 1987.

[BRA88] R.K. Brayton et al. "Multilevel logic minimization using implicit don't cares," *IEEE Trans. on Computer-Aided Design*, 7:723-740, 1988.

[BRA90] R.K. Brayton et al. "Multilevel logic synthesis," *Proceedings of the IEEE*, 78:264-300, 1990.

[BRY86] R.E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 677-691, 1986.

[BUR90] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill, "Sequential circuit verification using symbolic model checking," *Proc. 27th IEEE/ACM Design Automat. Conf.*, pp. 46-51, 1990.

[CAR91] G. Caruso, "Near optimal factorization of Boolean functions," *IEEE Trans. on Computer-Aided Design*, 10:1072-1078, 1991.

[CHA89] S. Chakravarty, "A testable multiplexer realization of CMOS combination circuits," *Proc. 1989 IEEE Int. Test Conf.*, pp. 509-518.

[CHA92] A.P. Chandrasekharan et al. "Low power CMOS digital design," *IEEE J. Solid State Circuits*, SC-27, pages 473-83, 1992.

[CHA93] S. Chakravarty "A characterization of binary decision diagrams," *IEEE Trans. Comput.*, vol. 42, no. 2, pp. 129-136, Feb. 1993.

[DIE69] D.L. Dietmeyer and Y-H Su, "Logic automation of fan-in limited NAND networks" *IEEE Trans. Comput.*, vol. C-18, no. 1, Jan. 1969.

[ELL65] D.T. Ellis, "A synthesis of combinational logic with NAND or NOR elements" *IEEE Trans. Electron. Comput.*, vol. EC-14, p. 701-705, Oct. 1965.

[FRI90] S.J. Friedman and K.J. Supowit, "Finding the optimal variable ordering for binary decision diagrams" *IEEE Trans. on Computers*, vol.39, No.5, pp 710-713, May 1990.

[GHO94] D. Ghosh et al. "TWTXBB: A low latency high throughput multiplier architecture using a new 4 to 2 compressor," Proc. *7th Int. Conf. on VLSI Design*, pages 77-82, Jan. 1994.

[GOP91] S.K. Gopalkrishnan and G.K. Maki, "State assignment selection tests for pass transistor asynchronous sequential circuits," *Proc. 25th Asilomar Conference on Signals, Systems and Computers*, pages 516-520.

[HAC67] G.D. Hachtel and R.A. Rohrer, "Techniques for the optimal design and synthesis of switching circuits" Proc. *IEEE*, vol. 55, p. 1864-1867, Nov. 1967.

[HOR78] S. Horowitz and S. Sahani, *Fundamentals of Computer Algorithms*. Computer Sciencs Press Inc. 1978.

[HSU92] W.J. Hsu and W.J. Shen, "Coalgebraic division for multi-level logic synthesis," *Proceedings Design Automation Conference*, pages 438-442, Jun 1992.

[JUL94] G.A. Jullien et al. "Dynamic computational blocks for bit-level systolic arrays," *IEEE J. of Solid-State Circuits*, vol. 29, no. 1, pages 14-21, Jan. 1994.

[KAN94] Y. Kanie et al. "4-2 compressor with complementary pass-transistor logic," *IEICE Transactions on Electronics*, vol. E77-C, Iss. 4, pages 647-649, Apr. 1994.

[KUZ77] O.P. Kuznetsov, "Program realization of logical functions and automata Part 1: Analysis and synthesis of binary processes," *Automata and Remote Contr.*, vol. 38, pp. 1077-1087, 1977.

[LEC70] R.J. Lechner, "Atransform approach to logic design" *IEEE Trans. Comput.*, vol. C-19, no. 7, Jul. 1970.

[LEE59] C. Lee, "Representation of switching circuits by binary decision programs," *Bell Syst. Tech. J.*, vol. 38, pp. 985-999, July 1959.

[MAL89] A.A. Malik et al. "Logic minimization for factored forms," *Proceedings Int. Conf. on Computer Design*, 1989.

[MAT89] Y. Matsunaga and M. Fujita, "Multilevel logic optimization using binary decision diagrams," *Proc. IEEE Int. Conf. Comput.-Aided Design*, 1989, pp. 556-559.

[MCM84] C. McMullen et al. "Synthesis and optimization of multistage logic," *Proceedings Int. Conf. on Computer Design*, 1984.

[MEN69] K.S. Menger, " A transform for logic networks" *IEEE Trans. Comput.*, vol. C-18, no. 3, Mar. 1969.

[MIT92] M. Mittal and C.A.T. Salama, "DPTL 4-b carry look ahead adder," *IEEE J. of Solid-State Circuits*, pages 1644-1647, Nov. 1992.

[NAI86] R. Nair and D. Brand, "Construction of optimal DCVS trees," *Tech. Report RC11863*, IBM T.J. Watson Research Centre, N.Y. 1986.

[PAS85] J.H. Pasternak et al. "CMOS differential pass transistor logic design," *IEEE J. of Solid-State Circuits*, 22:216-222, 1985.

[PAS89] J.H. Pasternak et al. "Differential pass-transistor logic partial-product generator for iterative multipliers," *Proceedings of ECCTD*, Brighton, U.K., 1989.

[PAS91] J.H. Pasternak, *High-Speed Differential Pass-Transistor Logic*. PhD thesis, University of Toronto, 1991.

[PAS91b] J.H. Pasternak et al. "Differential pass-transistor logic for GaAs E/D MESFET technologies," *Proceedings Symp. on VLSI Circuits*, 1991.

[PAS91c] J.H. Pasternak and C.A.T. Salama, "Design of submicrometer CMOS differential pass-transistor logic circuits," *IEEE J. of Solid-State Circuits*, vol. 26, Iss. 9, pages 1249-58. Sept. 1991.

[PAS93] J.H. Pasternak and C.A.T. Salama, "Differential pass-transistor logic," *IEEE Circuits and Devices Magazine*, vol:9, pages 23-28, July 1993.

[PED88] C. Pedron et al. "Analysis and Synthesis of Combinational pass transistor circuits," *IEEE Trans. on Computer-Aided Design*, 7:775-785, 1988.

[QUI55] W.V. Quine, "A way to simplify truth functions," *The American Mathematical Monthly*, vol.62, pages 627-631, Nov. 1955.

[RAD85] D. Radhakrishnan et al. "Formal design procedures for pass transistor circuits," *IEEE J. of Solid-State Circuits*, 25:531-536, 1985.

[RUD89] R.L. Rudell, *Logic Synthesis for VLSI Design*. PhD thesis, University of California, Berkeley, 1989.

[SAL93] O. Saloman and H. Klar, "Self-timed fully pipelined multipliers," *IFIP Transactions A [Computer Science and Technology]*, vol. A-28, pages 45-55, 1993.

[SCH68] P.R. Schneider and D.L. Dietmeyer, "An algorithm for synthesis of multiple-output combinational logic" *IEEE Trans. Comput.*, vol. C-19, no. 3, Mar. 1970.

[SHE70] Y-S Shen and A.C. McKellar, "An algorithm for the disjunctive decomposition of switching functions" *IEEE Trans. Comput.*, vol. C-27, no. 6, p. 509-516, June 1978.

[SID94] S.K. Siddiq, *Module Generators from Topological Descriptions and Graph Theoretic Approach*. Masters thesis, University of Windsor, 1989.

[SUZ93] M. Suzuki et al. "A 1.5ns 32-b CMOS ALU in double pass-transistor logic," *Proceedings ISSCC93*, 1993.

[WAN89] A. Wang *Algorithms for Multi-Level Logic Optimization.* PhD thesis, University of California, Berkeley, 1989.

[WHI83] S. Whittaker. "Pass-transistor networks optimize NMOS logic," *Electronics*, pages 144-148, Sept. 1983.

[WHI92] S. Whittaker and G.K. Maki, "Self-synchronized asynchronous sequential pass transistor circuits," *IEEE Trans. on Computers*, C-41, pages 1344-1348, Oct. 1992.

[YAN90] K. Yano et al. "A 3.8ns CMOS 16X16-b multiplier using complementary pass-transistor logic," *IEEE J. Solid-State Circuits*, 25:388-395, 1990.

[YAU70] S.S. Yau and C.K. Tang, "Universal logic modules and their applications" *IEEE Trans. Comput.*, vol. C-19, no. 2, Feb. 1970.

[YUA87] J. Yuan et al. "A true single-phase-clock dynamic CMOS circuit technique," *IEEE J. Solid-State Circuits*, 22:899-901, 1987.

[YUA89] J. Yuan and C. Svensson, "High-speed CMOS circuit technique," *IEEE J. Solid-State Circuits*, vol.24, No. 1, pages 62-70, Jan 1989.

Glossary

BDD: Binary Decision Diagram - a directed acyclic representing a Boolean function.

Best-factor: A modification of Brayton's generic factoring algorithm which chooses the best divisor (in terms of literal count) for factorization at each step.

Boolean network: A technology independent multilevel logic structure, used to represent multi-output Boolean functions.

Control signal: The signals applied to the gates of the transistors in a PTL network. The associated variables are called control variables.

Factorization: Process of decomposing a two-level function into a multilevel AND/OR form.

Kernel: A cube-free subexpression of a function which can be used as a divisor for factorization.

OBDD: Ordered BDD - a BDD where the ordering of the inputs in the various paths are the same.

PTL: Pass Transistor Logic

PTL network: A network of transistors realizing a Boolean function, where inputs may be applied to the Gate or Source/Drain of the transistors.

Pass implicant: A single cube with an associated pass signal which covers a group of minterms.

Pass signal: The signals that are passed through the PTL network to the output. The associated variables are called pass variables.

Prime Pass implicant: A pass implicant which is not covered by any other single pass implicant.

Signal string: A modified bitmap representing the function associated with a node in a 123-DD.

Sneak path: Spurious paths introduced in a MOS circuit due to the bidirectional nature of MOS transistors.

123-DD: A modification of the binary decision diagram used to represent PTL networks.

Appendix A

Algorithms Used in Multilevel Synthesis

In this appendix we will give the details of some of the important algorithms outlined in chapters 3 and 4.

A.1 Algorithms for Factorization

find_factored_form

Given a prime pass implicant P_i, this function tries to find a factored form FF, which contains P_i, is ordering compatible with the previously selected factored forms FF_1, FF_2, ... , FF_{j-1} and has a score greater than that of P_i. find_factored_form also takes as inputs the score of P_i, an array of prime pass implicants and Θ, a set of minterms already covered.

find_factored_form(P_i, score_of_P_i, array_of_previous_factored forms, array_of_prime_pass_implicants, Θ)

1. Select, from array_of_prime_pass_implicants, all prime pass implicants P_k such that

   a. P_k covers at least one element not in P_i

   b. P_i and P_k have at least one common literal

2. REPEAT

   a. Using algebraic division, compute q_{ki} where $F = q_{ki} * d_{ki} + R_{ki}$. Here F is the sum of all prime pass implicants and q_{ki} is a sum of cubes ($q_{ki} = c_1 + c_2 + ... + c_t$) of literals that do not appear in d_{ki}.

   b. Discard from q_{ki} all cubes c_i such that $c_i * d_{ki}$ does not cover any minterms not in Θ. Let q'_{ki} be the sum of remaining cubes.

   c. If the ordering requirements of (q'_{ki}, d_{ki}) do not conflict with those imposed by FF_1, FF_2, ... , FF_{j-1} compute the score for the ordering compatible factored form (q'_{ki} , d_{ki})

UNTIL (all prime pass implicants P_k selected in step 1 have been considered) OR (score of current factored form (q'_{ki}, d_{ki}) > score_of_P_i)

3. IF (score of current factored form > score_of_P_i) RETURN (q'_{ki}, d_{ki})

ELSE RETURN "failed"