Automated Synthesis and Optimization of Multilevel Logic Circuits

by

Lingli Wang

A thesis presented in partial fulfilment
of the requirements for the degree of

Doctor of Philosophy

Napier University

School of Engineering

2000

Declaration

I declare that no portion of the work referred to in this thesis has been submitted in support of an application of another degree, qualification or other academic awards of this or any other university or institution of learning.

[illegible]

Lingli Wang

Acknowledgements

I am deeply indebted to my research supervisor, Professor A. E. A. Almaini, School of Engineering, Napier University, for his constant guidance, encouragement, friendship, and interest to all that I did during this research. I am grateful to Professor Almaini for taking his precious time to provide regular comments and invaluable suggestions. Many ideas of this thesis were produced during our weekly meetings. He also taught me patiently the proper style of scientific writing in English and corrected my research papers and this thesis.

I would like to thank Mr Alex Bystrov, a former member of the digital techniques group, for his essential support to help me learn Linux operating system and GNU C language programming environment when I started this research.

Thanks are due to other members of the digital techniques group, especially Mr Yinshui Xia and Mr Belgasem Ali, for their enjoyable working environment and various discussions.

I would like to thank our technician, Mr Jay Hoy for his kindly support to install Linux operating system.

This project was funded by the School of Engineering (formerly the Department of Electrical, Electronic and Computing Engineering), Napier University, Edinburgh. This support is gratefully acknowledged.

Finally, I would like to thank my wife, Xiaowen Zhong, for her everlasting love, understanding and patience throughout my research work.

Contents

List of Abbreviations **vi**

List of Figures **ix**

List of Tables **xi**

Abstract **xii**

1 Introduction **1**

1.1 A historic perspective of logic design 1

1.2 VLSI chip design methodology based on logic synthesis 4

1.3 Two-Level versus multilevel logic synthesis 7

1.4 Reed-Muller logic based on AND/XOR operations 8

1.5 Structure of this thesis 10

2 Conventional Multilevel Logic Synthesis **12**

2.1 Algorithmic approach 12

2.2 Rule-Based approach 15

2.3 BDD approach 16

2.4 FPGA approach 18

2.5 Several other approaches based on perturbation 20

3 Multilevel Logic Minimization Using Functional Don't Cares **22**

3.1 Introduction 22

3.2 Multilevel Karnaugh map technique 23

3.3 Multilevel logic synthesis for large Boolean functions 29

3.4 Experimental results 38

3.5 Summary 40

4 Multilevel Minimization for Multiple Output Logic Functions **41**

4.1 Introduction 41

4.2 Review of Functional Don't Cares 41

4.3 Simplification for single output functions 44

4.4 Multilevel minimization for multiple output functions 46

4.5 Experimental results 49
4.6 Summary 50

5 Polarity Conversion for Single Output Boolean Functions 51
5.1 Introduction 51
5.2 Basic definitions and terminology 51
5.3 Conversion of the coefficients with zero polarity 53
5.4 Conversion of the coefficients with a fixed polarity 56
5.5 Conversion algorithm for large Boolean functions 61
5.6 Algorithm and experimental results 65
5.7 Summary 70

6 Conversion Algorithm for Very Large Multiple Output Functions 71
6.1 Introduction 71
6.2 Algorithm 74
6.3 Experimental Results 76
6.4 Summary 77

7 Exact Minimization of Fixed Polarity Reed-Muller Expressions 78
7.1 Introduction 78
7.2 Background 79
7.3 Properties of the polarities for SOP and FPRM expressions 83
7.4 Best polarity for single output functions 93
7.5 Best polarity for multiple output functions 95
7.6 Experimental results 98
7.7 Summary 99

8 Optimization of Reed-Muller PLA Implementations 101
8.1 Introduction 101
8.2 Review of the decomposition method 103
8.3 Improved decomposition method for large single output functions 106
8.4 Generalization to multiple output functions 116
8.5 Experimental results 122
8.6 Summary 123

9 Conclusions and Future Work 125

Publications 128

References and Bibliography 129

Disk Containing the Software 141

List of Abbreviations

ASIC	Application Specific Integrated Circuit
ATPG	Automatic Test Pattern Generation
BDD	Binary Decision Diagram
BDP	Binary Decision Program
BLIF	Berkeley Logic Interchange Format
CAD	Computer Aided Design
CLB	Configurable Logic Block
CNF	Conjunction Normal Form
CMOS	Complementary Metal-Oxide-Semiconductor
CPU	Central Process Unit
DAG	Directed Acyclic Graph
DC	Don't Care
DD	Decision Diagram
DNF	Disjoint Normal Form
EDA	Electronic Design Automation
EDIF	Electronic Design Interchange Format
ESOP	Exclusive Sum-Of-Products
EXOR(XOR)	EXclusive OR operation
FDD	Functional Decision Diagram
FPGA	Field Programmable Gate Array
FPLA	Field Programmable Logic Array
FPRM	Fixed Polarity Reed-Muller
GA	Genetic Algorithm

GCC	GNU Compiler Collection
GDB	GNU DeBugger
HDL	Hardware Description Language
IC	Integrated Circuit
IWLS	International Workshop on Logic Synthesis
KISS	Keep Internal State Simple
LSB	Least Significant Bit
LUT	Look-Up Table
MCNC	Microelectronics Center North Carolina
MIS	Multilevel logic optimization system developed at Berkeley
MPGA	Mask Programmable Gate Array
MSB	Most Significant Bit
MUX	Multiplexer
NP	Nondeterministic Polynomial, Non-Polynomial
NRE	Non-Recurring Engineering
OBDD	Ordered Boolean Decision Diagram
ODC	Observability Don't Care
OFDD	Ordered Functional Decision Diagram
OVAG	Output Value Array Graph
P&R	Placement & Routing
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PROM	Programmable Read-Only Memory
PPRM	Positive Polarity Reed-Muller
RMPLA	Reed-Muller Programmable Logic Array
ROBDD	Reduced Ordered Binary Decision Diagram

RTL	Register and Transfer Level
SCRL	Split-level Charge Recovery Logic
SDC	Satisfiability Don't Care
SIS	Sequential Interactive System
SLIF	Stanford Logic Interchange Format
SOC	System-On-Chip
SOP	Sum-Of-Products
SRAM	Static Random Access Memory
VHDL	Very large scale Hardware Description Language
VLSI	Very Large Scale Integration
XNF	Xilinx Net-list Format

List of Figures

1.1 Classical symbols for contacts 2
1.2 Moore's law — the growth of Intel microprocessors 3
1.3 Historic perspective of logic design 4
1.4 A VLSI design model 5
1.5 An ASIC chip design flow 6
1.6 Comparison between two-level and multilevel structures 8
1.7 Structure of this thesis 11

2.1 DAG representation of a combinational Boolean network 13
2.2 A circuit pattern and its replacement 15
2.3 BDD representation of the Boolean function in example 2.1 17
2.4 Logic circuit from fig.2.3(b) 18
2.5 Typical FPGA architecture 19
2.6 Different partitions for the same Boolean network 20

3.1 Comparison between two-level and multilevel K-map methods 26
3.2 Multilevel K-map method for an incompletely specified function 29
3.3 Two examples for definition 3.3 30
3.4 Examples of theorem 3.3 33
3.5 An example of procedure 3.2 38

4.1 Explanation of the functional DCs 43
4.2 Example of functional DCs 44
4.3 Results of *ryy6* with 16 inputs 46
4.4 Simplification of Boolean relation 48

5.1 Bidirectional conversion between SOP and FPRM forms 60
5.2 Conversion algorithm using multiple segment technique 65
5.3 Bidirectional conversion algorithm for large Boolean functions 67
5.4 CPU Time versus the number of on-set coefficients for conversion 69
5.5 CPU time for parity function conversions 70

6.1 Functional decision diagrams for $f = x_2 \oplus x_1x_0 \oplus x_3x_2x_1x_0$ 73
6.2 An example for algorithm 6.1 75

7.1 Example for a 3-variable 2-output Reed-Muller function 98

8.1 Limitation of $\frac{3}{4}$-majority m-cubes 104
8.2 Structures of Reed-Muller PLAs 104
8.3 Example of procedure 8.1 108
8.4 Order of $\frac{3}{4}$-majority cubes 110
8.5 Example of procedure 8.2 112
8.6 Boolean function of example 8.4 113
8.7 Example for procedure 8.3 115
8.8 Example for procedure 8.4 117

List of Tables

1.1 Some commercial design synthesis tools 5

2.1 Brief comparison of FPGA programming techniques 19

3.1 Comparison for single output functions run on the same PC 39

4.1 Literal numbers with different encodings for *table5* 49
4.2 Results for multiple output functions 49

5.1 Truth table of the criterion function g_j 55
5.2 Example of multiple segment technique 64
5.3 Conversion results of some random coefficient sets 68
5.4 Test results for conversion of parity functions 69

6.1 Distribution of dependent variables 74
6.2 Experimental results of very large functions from IWLS'93 benchmark . . . 76

7.1 Test results for small Boolean functions 100
7.2 Test results for large Boolean functions 100

8.1 Comparison with ESPRESSO for general multiple output functions 123
8.2 Comparison with ESPRESSO for very large multiple output functions 123

Abstract

With the increased complexity of Very Large Scaled Integrated (VLSI) circuits, multilevel logic synthesis plays an even more important role due to its flexibility and compactness. The history of symbolic logic and some typical techniques for multilevel logic synthesis are reviewed. These methods include algorithmic approach; Rule-Based approach; Binary Decision Diagram (BDD) approach; Field Programmable Gate Array(FPGA) approach and several perturbation applications.

One new kind of don't cares (DCs), called functional DCs has been proposed for multilevel logic synthesis. The conventional two-level cubes are generalized to multilevel cubes. Then functional DCs are generated based on the properties of containment. The concept of containment is more general than unateness which leads to the generation of new DCs. A separate C program has been developed to utilize the functional DCs generated as a Boolean function is decomposed for both single output and multiple output functions. The program can produce better results than script.rugged of SIS, developed by UC Berkeley, both in area and speed in less CPU time for a number of testcases from MCNC and IWLS'93 benchmarks.

In certain applications, AND/XOR (Reed-Muller) logic has shown some attractive advantages over the standard Boolean logic based on AND/OR operations. A bidirectional conversion algorithm between these two paradigms is presented based on the concept of polarity for sum-of-products (SOP) Boolean functions, multiple segment and multiple pointer facilities. Experimental results show that the algorithm is much faster than the previously published programs for any fixed polarity. Based on this algorithm, a new technique called redundancy-removal is applied to generalize the idea to very large multiple output Boolean functions. Results for benchmarks with up to 199 inputs and 99 outputs are presented.

Applying the preceding conversion program, any Boolean functions can be expressed by fixed polarity Reed-Muller forms. There are 2^n polarities for an n-variable function and the number of product terms depends on these polarities. The problem of exact polarity minimization is computationally extensive and current programs are only suitable when $n \leq 15$. Based on the comparison of the concepts of polarity in the standard Boolean logic and Reed-Muller logic, a fast algorithm is developed and implemented in C language which can find the best polarity for multiple output functions. Benchmark examples of up to 25 inputs and 29 outputs run on a personal computer are given.

After the best polarity for a Boolean function is calculated, this function can be further simplified using mixed polarity methods by combining the adjacent product terms. Hence, an efficient program is developed based on decomposition strategy to implement mixed polarity minimization for both single output and very large multiple output Boolean functions. Experimental results show that the numbers of product terms are much less than the results produced by ESPRESSO for some categories of functions.

Chapter 1

Introduction

1.1 A historic perspective of logic design

In 1854, when George Boole published his principle work, *An Investigation of the Laws of Thought on Which Are Founded the Mathematical Theories of Logic and Probabilities*, a new branch of mathematics, *Symbolic Logic*, was established[22]. Logic propositions and operations started to be represented by symbols although it had already been expressed by natural language in times of Aristotle, the fourth century B.C. Ever since, great progress has been made in the field of *Symbolic Logic* by the broad application of algebraic methods and facilities[36, 79, 95]. Until 1938, *Symbolic Logic* was used only as another tool of philosophy and mathematics. In that year, a graduate student at Massachusetts Institute of Technology recognized the connection between electronic switching circuits and *Symbolic Logic* and published a classic paper, entitled "*A Symbolic Analysis of Relay and Switching Circuits*"[135]. Based on that paper, both logic values, "TRUE" and "FALSE" can be mapped into "MAKE" and "BREAK" states of a two-state device so that logic manipulations can be realized by switching circuits. Since then, many switching circuits and systems for communication, automatic control, and data processing have been designed by applying *Symbolic Logic*, which later came to be known as *Boolean Algebra*[63, 77, 84, 96, 99, 117].

During the early days, logic functions usually expressed by truth tables, Karnaugh maps[84], and algebraic formulas, either DNFs(disjoint normal forms) or CNFs(conjunction normal forms).[1] Besides, a two-state switch or contact is usually realized by relay, magnetic core, rectifying diode, electron tube, or cryotron operated with the aid of electromagnet[77]. The physical nature of the two stable states may take such forms as conducting versus non-conducting; closed versus open; charged versus discharged; positively magnetized versus negatively magnetized; high potential versus low potential etc. The two states of a switch were generally called "break" and "make" contacts whose common symbols are shown in

[1]They are equivalent to SOP(sum of products) and POS(product of sums) forms respectively.

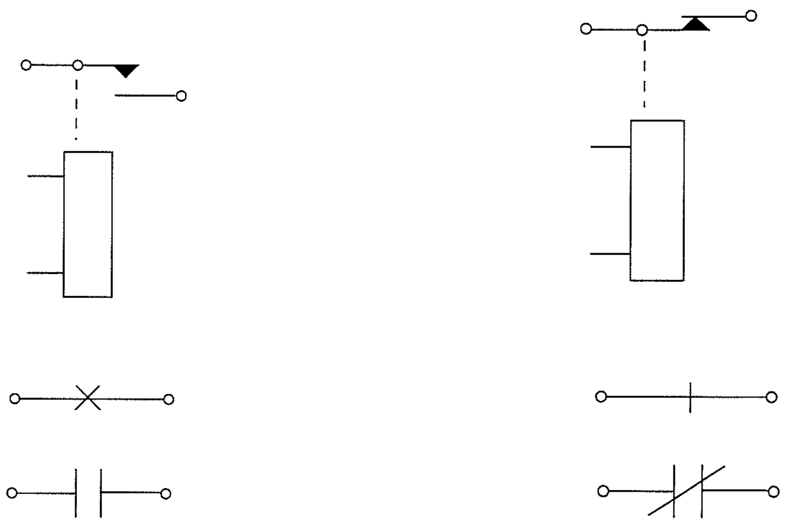

(a) symbols for break contacts (b) symbols for make contacts

Figure 1.1: Classical symbols for contacts

fig.1.1. This research, that can be found mainly in [63, 77, 99], becomes the classical method of logic minimization for both combinational and sequential systems.

In 1947, the age of semiconductors arrived with the development of the first junction transistors by William Shockley and his colleagues in Bell Labs of United States. The semiconductor transistors have great advantage over the traditional electromagnetic switching devices in size, speed, power dissipation and reliability etc. In 1958, the world's first integrated circuits (ICs) was developed by Jack Kilby in Texas Instruments where transistors, resistors and capacitors along with their interconnecting wiring were fabricated on a single piece of Germanium and glued to a glass slide. From then on, the integrated circuit technology has progressed tremendously. In 1961, commercial ICs became available from Fairchild Instruments. Four years later, Gordon Moore, co-founder of Intel, predicted that the number of transistors on a chip of integrated circuits could be doubled every 12 to 18 months. His statement can still be validated by Intel microprocessors as show in fig.1.2[101]. Unfortunately, most of the classical methods of logic minimization are only suitable for small Boolean functions. Therefore, these logic design methods must be improved to meet the rapid growth of the semiconductor technology.

Although there are only several theorems in Boolean algebra to simplify Boolean functions, the obtained results of minimization largely depend on how to select the orders of these theorems and to which terms to apply them[84]. For large Boolean functions, there are too many alternatives to be considered exhaustively. In 1976, the first general survey of Boolean function complexity was introduced in [129]. Following that there was immense research on the complexity of Boolean networks and their realizations that can be found in

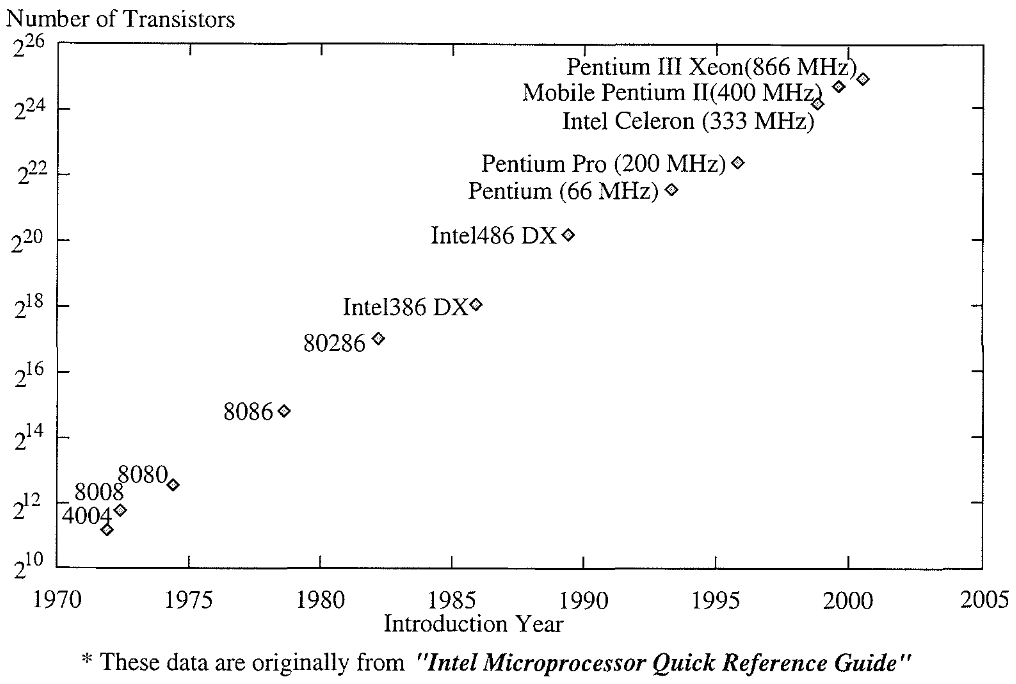


Figure 1.2: Moore's law — the growth of Intel microprocessors

[57]. It was realized that neither the traditional methods of switching theory nor manual design was feasible for large functions. With the development of automatic physical design methods for large Boolean systems[115, 123], several heuristic and efficient methods have been applied to obtain "good" solutions for large Boolean functions with the aid of computers. This process is typically called logic optimization or logic synthesis that was first commercially available in 1980's[131]. Recently the designs of *systems-on-a-chips* (SOCs) started to attract more and more organizations that lead to urgent demand for new generation logic synthesis tools[76]. In July of 1997, logic synthesis tools for designing Application Specific Integrated Circuits (ASICs) and SOCs became available[13]. Expert systems may be a new facility for logic synthesis[132]. Besides, Linux has been recommended to be a new common operating system in EDA communities because it is the only candidate offering a compatibility for both workstations and personal computers (PCs)[133].

On the other hand, there is a new challenge for *Symbolic Logic* proposed in Japan. In their opinion, the current prevailing theory of *Symbolic Logic* needs a thorough revision because of some crucial misunderstanding. Their newly developed theory is available in the electronic book of Internet, *Elements of the Reformed Theory of Logic*[136]. It will not be discussed here since this topic is beyond the coverage of this thesis. The above historic perspective can be shown in fig.1.3, where "logic synthesis" is the subject of this thesis.

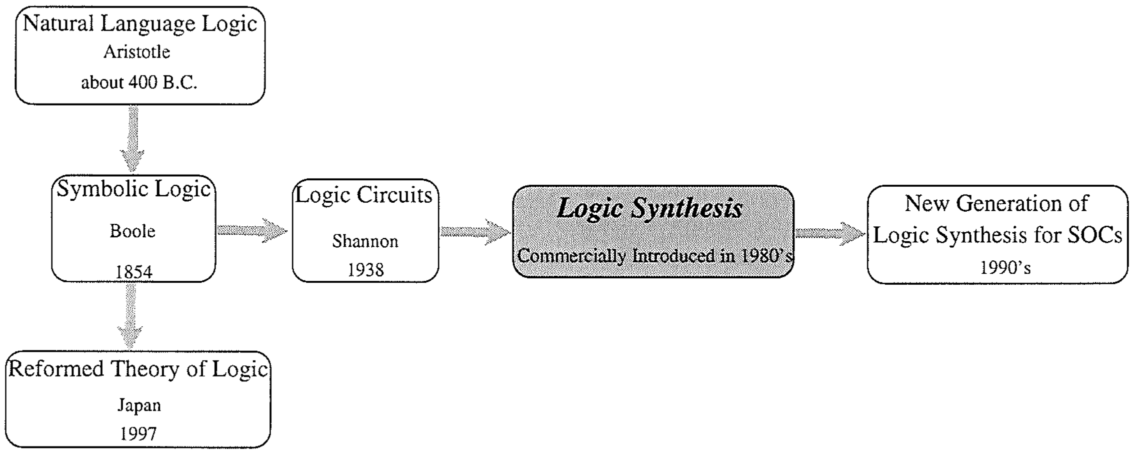


Figure 1.3: Historic perspective of logic design

1.2 VLSI chip design methodology based on logic synthesis

The wide range of computer-aided design (CAD) tools for digital integrated circuits can fall into four major categories based on the production of Very Large Scale Integration (VLSI) components[59]: memories, microprocessors, Application Specific Integrated Circuits (ASICs), and Programmable Logic Devices (PLDs)[29, 108, 124]. Numerous industrial applications can be implemented by either ASICs or field programmable gate arrays (FPGAs) and offer distinct advantages. Besides, countless commercial design synthesis tools are available and some of them are shown in table 1.1. Today's ASICs have a low cost-per-gate advantage as well as an inherent speed advantage. In contrast, FPGAs have been winning with their time-to-market and reprogrammability. This detail analysis and prediction of the competition between ASICs and FPGAs is addressed in [14]. Additionally, it can be seen that VHDL[113] and Verilog[140] are the most popular hardware description languages(HDLs). Furthermore, each company usually applies different design methodology. However, a unified model of design representation has been developed in [149]. It is proposed that a model of design representation can be described by three separate domains, namely behavioral, structural, and physical domains. Behavioral domain describes the basic functionality of the design while structural domain the abstract implementation, and the physical domain the physical implementation of the design. Each domain can further be divided into five abstract levels that are architectural, algorithmic, register & transfer, logic and circuit levels. This design model can be represented in fig.1.4 that is quite similar to Y-chart in [65].

Based on the model in fig.1.4, design synthesis is the process of translating a high abstract level in the behavioral domain to a low level in the physical domain through structural domain. Different design methodology may take different tracks on this model. Moreover, not all the levels in this three domains need to be fitted neatly. For example, silicon compilation can generate physical layout directly from behavioral description[15].

Organization	System	Description	Input	Environment
Bell Labs Design Automation	Synovation	High-level RTL Synthesis	VHDL & Verilog	SunOS
Cadence Design Systems	FPGA Designer ASIC Designer	FPGA, PIC ASIC/IC design packages	Schematic Verilog, VHDL	Unix
Compass Design Automation	ASIC Synthesizer	FPGA, ASIC ASSP designs	VHDL & Verilog	Unix
Mentor Graphics	AutoLogic II PLD Synthesis II	ASIC, PLD	VHDL, Verilog ABEL-4,JEDEC,	Unix
Synopsys	Design Compiler Family	ASIC, FPGA	VHDL Verilog	Unix
Synplicity	Synplify	FPGA/CPLD Synplify Editor	VHDL Verilog	Windows
Viewlogic Systems	ViewSynthesis	FPGA Design	VHDL	Unix Windows

Table 1.1: Some commercial design synthesis tools

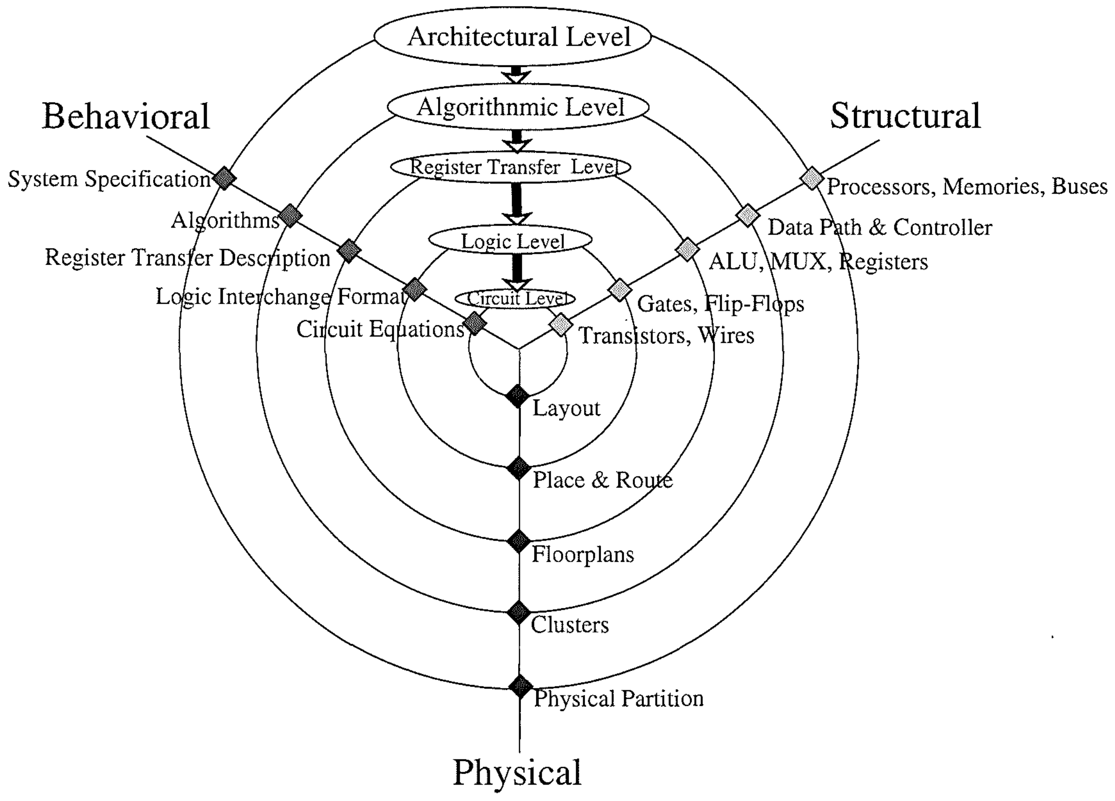


Figure 1.4: A VLSI design model

A typical ASIC chip design flow based on logic synthesis can be shown in fig.1.5.

It can be seen from fig.1.5 that three steps are involved in logic synthesis:

1. Convert the description from register transfer level to logic level consisting of AND/OR

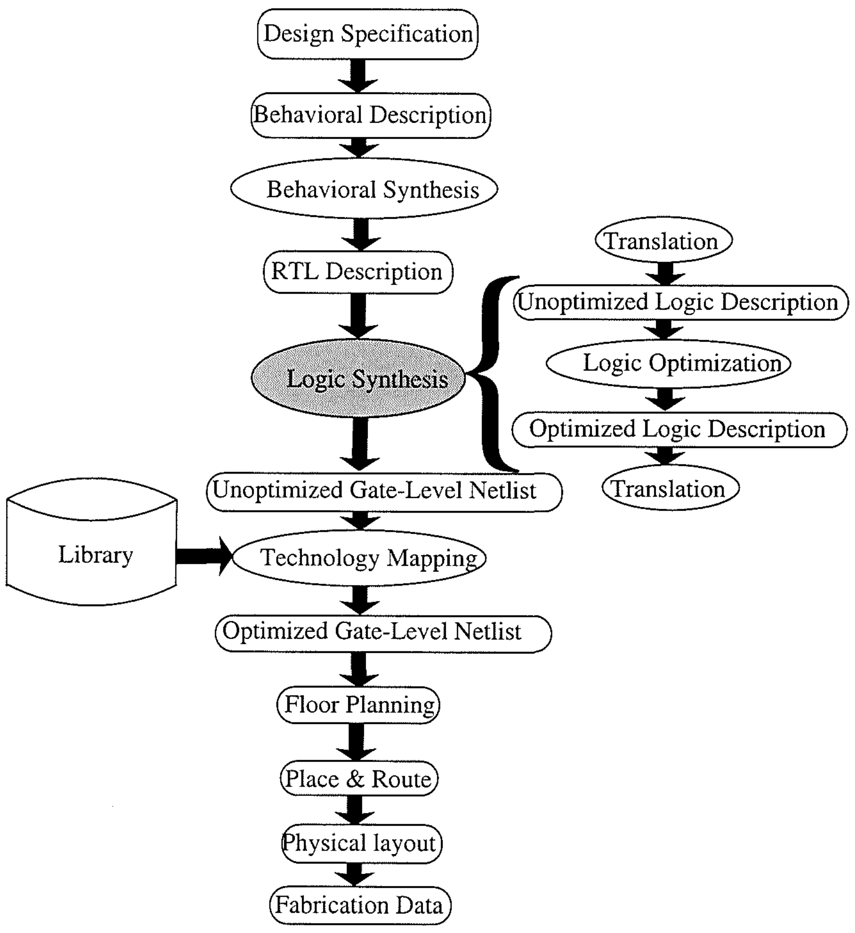


Figure 1.5: An ASIC chip design flow

gates, flip-flops, and latches. The description file in logic level may be in the formats of KISS(Keep Internal State Simple), BLIF (Berkeley Logic Interchange Format), SLIF (Stanford Logic Interchange Format), PLA(Programmable Logic Array), and equations. All these formats can be converted to each other by SIS[134].

2. Optimize the description through various available procedures by the criteria of area, speed, power dissipation[19], or testability. This important process is typically called logic optimization. Recently adiabatic circuits based on split-level charge recovery logic(SCRL) is a new topic for low power VLSI design[62].

3. Produce a gate level net-list, usually in electronic design interchange format (EDIF)[58].

Comparatively speaking, step 2 is the most difficult one. In the last two decades, a lot of heuristic techniques have been developed for logic minimization of large Boolean functions and circuits. A new multilevel logic optimization method, based on functional don't cares (DCs), will be proposed according to the criterion of area in chapter 3 and 4 of this thesis.

1.3 Two-Level versus multilevel logic synthesis

In the logic level of design synthesis, the two-level logic minimization is a mature and very popular approach especially for control logic. Quine[117] laid down the basic theory that was adapted later by McCluskey[96], known as Quine-McCluskey procedure. It basically consists of two steps:

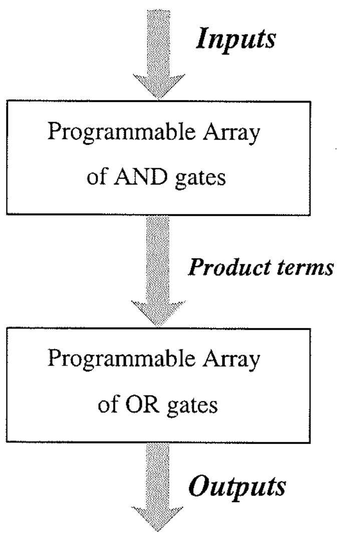
1. Generate all the prime implicants from on-set minterms of a Boolean function;
2. Select an optimal subset of these primes that cover all the on-set minterms of the function.

Even though the primes can be efficiently produced in [78], solving the covering problem is known as an NP-complete problem. Thus this technique becomes impractical for large Boolean functions. The next important contribution in this area is MINI[78] that was further developed into ESPRESSO[26], the most powerful and popular two-level minimizer up to date. It is possible for ESPRESSO to find "very good" solutions for incompletely specified functions with hundreds of inputs and outputs in a reasonable time. There is one main loop consisting of four main procedures in ESPRESSO, EXPAND, ESSENTIAL_PRIME, IRREDUNDANT_COVER, and REDUCE. EXPAND replaces the previous cubes by prime implicants and assures the cover is minimal with respect to single-cube containment. Then ESSENTIAL_PRIME extracts the essential primes and put them in the don't care set. Following that IRREDUNDANT_COVER find an optional minimum irredundant cover by deleting totally redundant cubes. Finally, REDUCE procedure reduce all the cubes to be smallest that cover only necessary on-set minterms. Although REDUCE produces a non-prime cover, it can facilitate improvement in the subsequent iterations over the local minimum result obtained by IRREDUNDANT_COVER.

Two-level logic and its Programmable Logic Array (PLA) implementations shown in fig.1.6(a) provide good solutions to a wide class of problems in logic design. However, there are situations, especially for large multiple output circuits, where multilevel design is desirable and more effective. It facilitates sharing and simplifies testing. For example, a simple logic circuit,

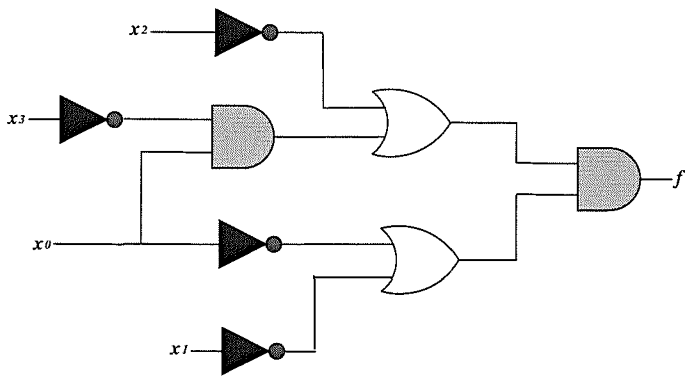
$$f = (x_0\bar{x}_3 + \bar{x}_2)(\bar{x}_0 + \bar{x}_1) \tag{1.1}$$

which can be shown in fig.1.6(b), has three levels. In the first level, the product term $x_0\bar{x}_3$ is generated, which is an AND level. In the second level, both $(x_0\bar{x}_3 + \bar{x}_2)$ and $(\bar{x}_0 + \bar{x}_1)$ are generated with an OR level. Finally, they are combined by an AND gate to produce the output for the function. Additionally, multilevel realization is useful for both control and data-flow logic[25]. However, multilevel logic circuits are much more difficult to be synthesized than two-level circuits. In 1964, Lawler proposed an approach for exact



Inputs
Programmable Array of AND gates
Product terms
Programmable Array of OR gates
Outputs

(a) Two-Level programmable logic array (PLA) structure



(b) A simple example for multilevel logic circuit

Figure 1.6: Comparison between two-level and multilevel structures

multilevel logic minimization[90]. All the multilevel prime implicants are first generated, then a minimal subset is found by solving a covering problem using any method for two-level minimization. As an exact optimization method, it is only suitable for small Boolean functions on account of high computational complexity. In the last two decades, many heuristic techniques have been developed that will be reviewed in chapter 2.

1.4 Reed-Muller logic based on AND/XOR operations

It was presented in the classic paper [135], published in 1938, that any n-variable Boolean function f can be expanded by Shannon expansion based on AND/OR operations as follows.

$$f(x_{n-1}x_{n-1}\cdots x_0) = \bar{x}_i f_{x_i=0} + x_i f_{x_i=1} \tag{1.2}$$

where $0 \leq i \leq n-1$, and $f_{x_i=0}$ and $f_{x_i=1}$ are called the cofactors of f with respect to x_i. Alternatively, any Boolean function can be represented based on AND/XOR operations, which is called Reed-Muller expansion[103, 119]. In contrast to equation (1.2), there are three basic expansions using AND/XOR operations, which are shown in equations (1.3) - (1.5).

$$f(x_{n-1}x_{n-1}\cdots x_0) = \bar{x}_i f_{x_i=0} \oplus x_i f_{x_i=1} \tag{1.3}$$

$$f(x_{n-1}x_{n-1}\cdots x_0) = f_{x_i=0} \oplus x_i(f_{x_i=0} \oplus f_{x_i=1}) \tag{1.4}$$

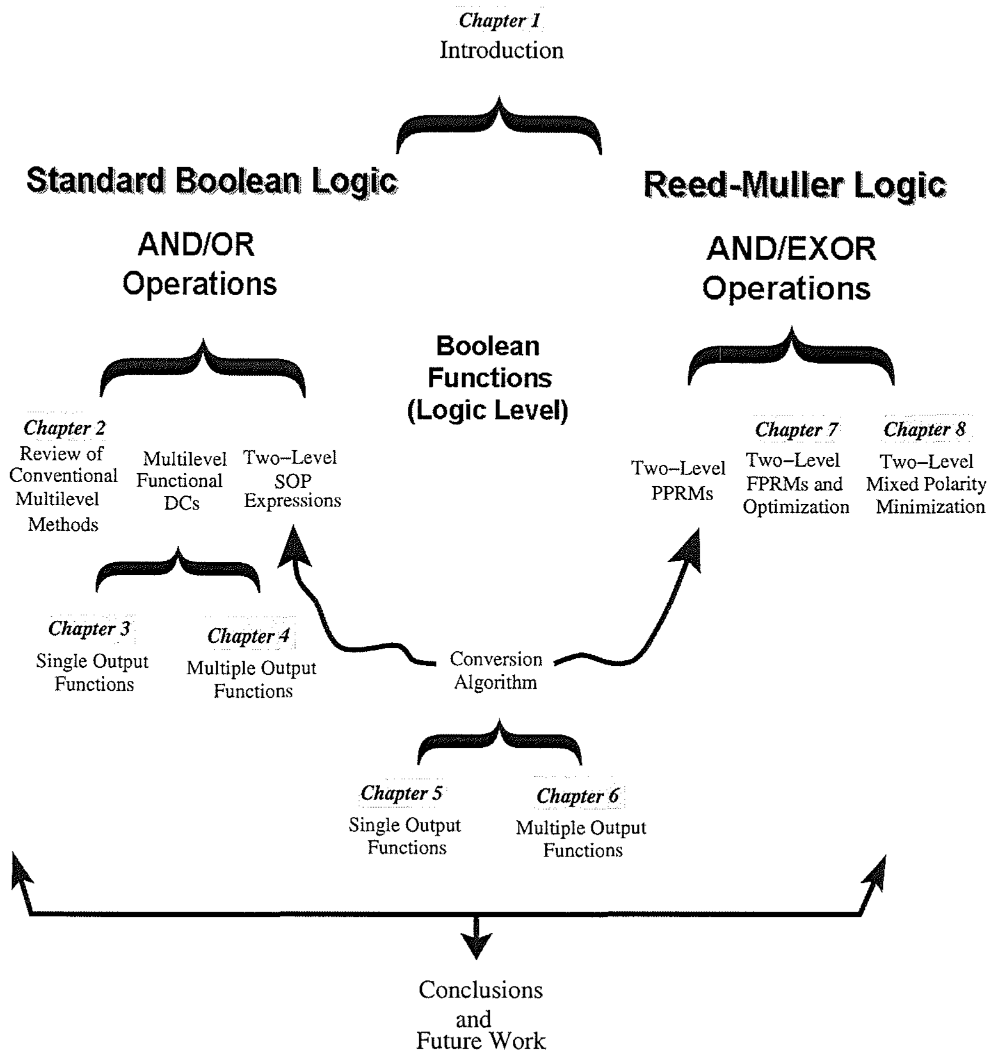
$$f(x_{n-1}x_{n-1}\cdots x_0) = f_{x_i=1} \oplus \bar{x}_i(f_{x_i=0} \oplus f_{x_i=1}) \tag{1.5}$$

In logic synthesis process, Reed-Muller logic methods are important alternatives to the traditional SOP approaches to implement Boolean functions. Currently, the widely used logic minimizers for SOP forms, such as ESPRESSO[26] and SIS[134] are based on the "unate paradigm", according to which most of the Boolean functions of practical interest are close to unate and nearly unate functions. While the category of unate and nearly unate functions covers many control and glue logic circuits, these minimizers perform quite poorly on other broad classes of logic[154]. For instance, the unateness principle does not work well for arithmetic circuits, digital signal processing operations, linear or nearly linear functions, and randomly generated Boolean logic functions[72, 126]. However, Reed-Muller realization is especially suitable for these functions[2, 46]. For example, to represent a parity function with n variables, $f = x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1}$, an SOP form needs 2^{n-1} product terms while only n terms are sufficient for an AND/XOR expression. Additionally, circuits based on AND/XOR operations have great advantage of easy testability [44, 92, 118]. Applications of Reed-Muller logic to function classification[143], Boolean matching[144], and symmetry detection[145] have also been achieved.

Due to the lack of an efficient Reed-Muller logic minimizer, applications of Reed-Muller implementations have not become popular despite these advantages. It is generally accepted that the optimization problem for Reed-Muller logic is much more difficult than the standard Boolean logic. One of the main obstacles is the polarity problem, including fixed polarity and mixed polarity, which does not exist in SOP forms for the standard Boolean logic. For a fixed polarity Reed-Muller (FPRM) form, the number of product terms largely depends on the polarity for the same function. Further, any Boolean function can be represented canonically by FPRM forms while it does not hold for mixed polarity expressions. Conventionally, Boolean functions are represented by AND/OR operations, instead of AND/XOR operations. Thus, the optimization of Reed-Muller logic consists of conversion algorithm between SOP and FPRM formats, fixed polarity and mixed polarity minimization. These will be discussed in chapters 5 to 8.

1.5 Structure of this thesis

Any Boolean function can be represented by two paradigms, which are based on AND/OR and AND/XOR operations respectively. Correspondingly, there are two main parts in this thesis as shown in fig.1.7. The first part is multilevel logic minimization based on AND/OR operations. The conventional methods for multilevel logic synthesis are reviewed in chapter 2 based on AND/OR operations. A new type of don't cares (DCs), functional DCs, are introduced comparing with satisfiability DCs and observability DCs. The usefulness of functional DCs is discussed for single output functions in chapter 3 and for multiple output functions in chapter 4 respectively. The second part deals with Reed-Muller logic which is based on AND/XOR operations. A mutual conversion algorithm is first proposed to convert a single output Boolean function between SOP and FPRM formats in chapter 5 and for very large multiple output functions in chapter 6 respectively. There are 2^n polarities for an n-variable function, and the number of on-set product terms largely depends on the polarity. Therefore a fast algorithm in presented in chapter 7 to find the best polarity for a function, which corresponds with the least number of on-set product terms. This FPRM form with the best polarity can be further simplified with respect to the number of product terms by combining the adjacent terms. Consequently, the result is in mixed polarity Reed-Muller forms, which is covered in chapter 8. Finally, the main improvements and contributions are summarized and some future work is suggested in the "conclusions and future work".


Chapter 1
Introduction
Standard Boolean Logic
AND/OR Operations
Reed-Muller Logic
AND/EXOR Operations
Boolean Functions (Logic Level)
Chapter 2
Review of Conventional Multilevel Methods
Multilevel Functional DCs
Two–Level SOP Expressions
Two–Level PPRMs
Chapter 7
Two–Level FPRMs and Optimization
Chapter 8
Two–Level Mixed Polarity Minimization
Chapter 3
Single Output Functions
Chapter 4
Multiple Output Functions
Conversion Algorithm
Chapter 5
Single Output Functions
Chapter 6
Multiple Output Functions
Conclusions and Future Work

PPRMs: positive polarity Reed-Muller expressions
FPRMs: fixed polarity Reed-Muller expressions

Figure 1.7: Structure of this thesis

Chapter 2

Conventional Multilevel Logic Synthesis

2.1 Algorithmic approach

The algorithmic approach to multilevel logic optimization consists of defining an algorithm for each transformation type, including elimination, decomposition, extraction, factoring, and substitution. These transformations are manipulated based on the concept of Boolean networks[74].

Definition 2.1. A Boolean network η for n-variable m-output functions is an interconnection of p Boolean functions defined by a five-tuple (f, y, I, O, d^X), consisting of:

1. $f = (f_0, f_1, \cdots, f_{p-1})$, a vector of completely specified logic functions. Each of them is a node in the network.

2. $y = (y_0, y_1, \cdots, y_{p-1})$, a vector of logic variables (signals of the network) where y_i has a one-to-one correspondence with f_i, $0 \leq i \leq p-1$. In other words, the output of a node can be an input for other nodes.

3. $I = (I_0, I_1, \cdots, I_{n-1})$, a vector of externally controllable signals as primary inputs.

4. $O = (O_0, O_1, \cdots, O_{m-1})$, a vector of externally observable signals as primary outputs.

5. $d^X = (d_0^X, d_1^X, \cdots, d_{m-1}^X)$, a vector of completely specified logic functions that specify the set of don't care minterms on the outputs of η where $X = y_I$.

For combinational logic functions, a Boolean network is usually equivalent to a Directed Acyclic Graph (DAG) as show in fig.2.1. Elimination, collapsing or flattening of an internal node of a Boolean network is its removal from the network, resulting in a network with one less nodes. Decomposition is the process of re-expressing a node as a number of sub-nodes so that these sub-nodes may be shared by other nodes in the network. Extraction, related to decomposition, is the process of creating a new common sub-node for several intermediate nodes in the network. Consequently, the structure of all these intermediate nodes are simplified. The optimization problem associated with the extraction transformation is to identify a set of common sub-nodes such that the resulting network has minimum area, delay, power dissipation, maximum testability or routability[41]. If the sum-of-products form of Boolean functions are considered as polynomial expressions, rather than Boolean expressions, then factoring the common polynomial terms will lead to a simpler structure of the corresponding network. The factoring problem is how to find a factored form with the minimum number of literals. Substitution or resubstitution, is the inverse transformation of elimination. It creates an arc in the Boolean network connecting the node of the substituting function to the node of the substituted function[50, 100].

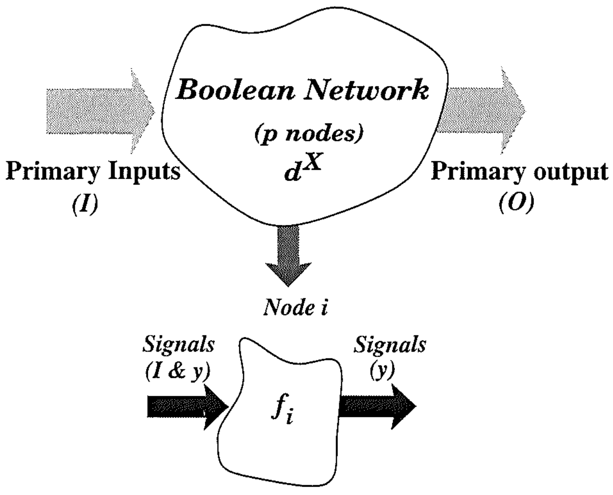


Figure 2.1: DAG representation of a combinational Boolean network

Among these transformations, division operation, the inverse process of product operation, plays a very important role to identify a common sub-expression. Given two expressions, F and P, find expressions Q and R, such that $F = P \bullet Q + R$, where Q and R are quotient and remainder respectively. If "$\bullet$" and "$+$" are taken as algebraic operations, then this is algebraic division, or weak division; otherwise, if they are taken as Boolean operations, then this is Boolean division. Comparatively speaking, algebraic division is faster but neglects other useful properties of Boolean algebra except distribute law. However, due to the lack of an efficient algorithm, Boolean division has rarely been applied[37]. A widespread concept of kernel is first proposed in 1982 based on algebraic division as follows[24].

Definition 2.2. The kernels of an expression F for a Boolean function f are the set of $K(F)$,

$$K(F) = \{g | g \in D(F) \text{ } and \text{ } g \text{ } is \text{ } cube \text{ } free\}$$

where

$$D(F) = \{F/c \mid c \text{ } is \text{ } a \text{ } cube\}$$

and g *is cube free* means that there is no cube to divide it evenly and "/" is algebraic division.

Example 2.1. A four-variable completely specified function $f(x_3x_2x_1x_0) = \Sigma\{0, 1, 2, 5, 8, 9, 10\}$ has a minimal two-level expression F_1 as follows,

$$F_1 = \bar{x}_2\bar{x}_1 + \bar{x}_2\bar{x}_0 + \bar{x}_3\bar{x}_1x_0 \tag{2.1}$$

In equation (2.1), $\bar{x}_0 + \bar{x}_1$, $\bar{x}_2 + \bar{x}_3x_0$ are the kernels[24] so that F_1 can be simplified to F_2 or F_3 by algebraic division in equations (2.2) or (2.3).

$$F_2 = \bar{x}_2(\bar{x}_0 + \bar{x}_1) + \bar{x}_3\bar{x}_1x_0 \tag{2.2}$$

and

$$F_3 = \bar{x}_1(\bar{x}_2 + \bar{x}_3x_0) + \bar{x}_2\bar{x}_0 \tag{2.3}$$

Both equations (2.2) and (2.3) are not the minimal multilevel forms because this function f can also be expressed by F_4 using Boolean division in equation (2.4).

$$F_4 = (\bar{x}_2 + \bar{x}_3x_0)(\bar{x}_1 + \bar{x}_0) \tag{2.4}$$

In addition to algebraic division technique, another facility that is extensively used is don't care method, including both satisfiability don't cares (SDC) and observability don't cares (ODC)[17, 130] in the logic level although there are more DCs in high level synthesis[20, 23]. These DCs are shown as d^X in fig.2.1. Generally, SDCs and ODCs are quite large and complex[38]. Hence filters are needed to reduce the size so that only the useful portions are retained[125].

Within MIS[27] that was further developed into SIS[134], there are many algorithms to compute the kernels and realize the previous five types of transformations. Additionally, some scripts are included in SIS to obtain good results for different kinds of functions. Extensive experimental results for SIS have been reported in [1].

2.2 Rule-Based approach

Rule-based systems are a class of expert systems that use a set of rules to determine the action to be taken to minimize a Boolean network[18, 47]. The network undergoes local transformations[48] that preserve its functionality by a stepwise refinement.

A rule-based system may consist of[100]:

- A rule database that contains two types of rules: replacement rules and meta-rules. The former abstract the local knowledge about subnetwork replacement and the latter the global heuristic knowledge about the convenience of using a particular strategy (i.e. applying a set of replacement rules).
- A system for entering and maintaining the database.
- A controlling mechanism that implements the inference engine.

A rule database contains a family of circuit patterns and the corresponding replacement for both replacement rules and meta-rules according to the overall goal, such as optimizing area, speed, power dissipation or testability. Several rules may match a pattern, and a priority scheme is used to choose the replacement. For example, a circuit pattern is shown in fig.2.2, when it is acknowledged by the rule-based system, it will be replaced by a smaller circuit.

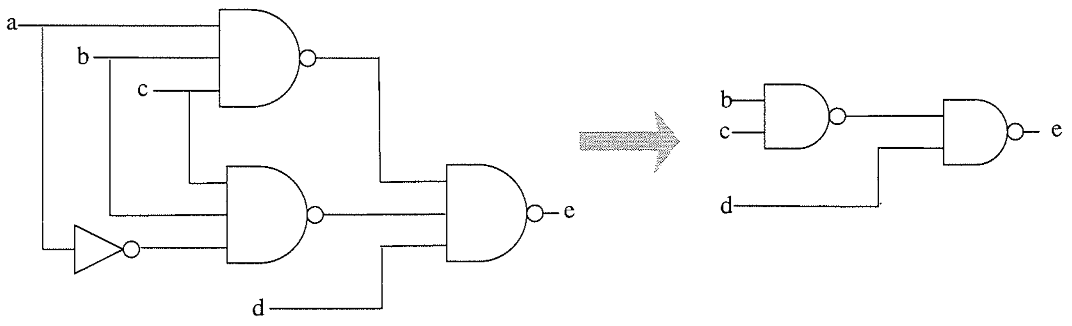


Figure 2.2: A circuit pattern and its replacement

A major advantage of this approach is that rules can be added to the database to cover all thinkable replacements and particular design styles. This feature plays a key role in the acceptance of optimization systems, because when a designer could outsmart the program, the new knowledge pattern could then be translated into a rule and incorporated into the database.

The major disadvantage in the rule-based system is the order in which rules should be applied and the possibility of look-ahead and backtracking. This is the task of the control algorithm, that implements the inference engine. Further discussion and experimental results can be found in [18, 47, 66].

2.3 BDD approach

Binary Decision Diagrams (BDDs) were first proposed as Binary-Decision programs (BDPs) in 1959[91], which are further generalized by Akers in 1978 and represented them as diagrams[12]. Early research results can be found in [102]. Any Boolean function can be expressed by a BDD based on the following definition[31].

Definition 2.3. A BDD is a rooted directed graph with vertex set V containing two kinds of vertices, terminal and nonterminal vertices. If a vertex is a terminal vertex, then it is associated a value of either 0 or 1 denoted as $value(v) \in \{0, 1\}$; otherwise, if it is a nonterminal vertex then it has as attribute an argument index, $index(v) \in \{0, 1, \cdots, n-1\}$ and two children, $low(v)$, $high(v) \in V$. An n-variable Boolean function f can be expressed by the value of the root vertex. Any vertex v corresponds with a Boolean function f_v defined recursively as:

1) If v is a terminal vertex:
   a) If $value(v)$ =1, then $f_v = 1$;
   b) If $value(v)$ =0, then $f_v = 0$.
2) If v is a nonterminal vertex with $index(v) = i$, then f_v is the function

$$f_v = \bar{x}_i f_{low(v)} + x_i f_{high(v)} \tag{2.5}$$

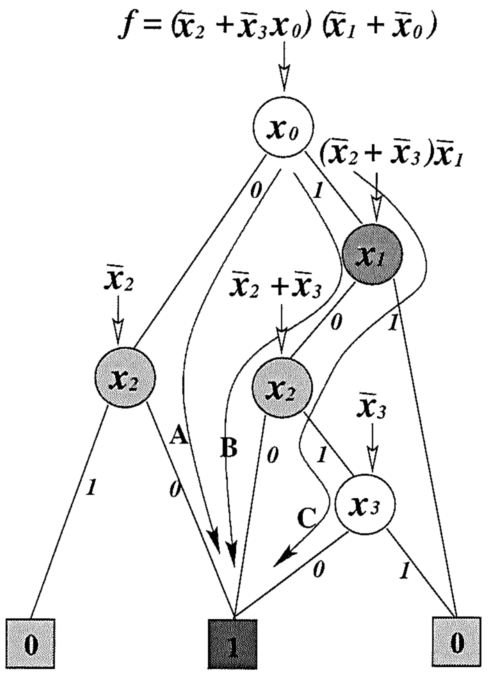
In example 2.1, a Boolean function $f(x_3x_2x_1x_0) = \Sigma\{0, 1, 2, 5, 8, 9, 10\}$ can also be expressed by a BDD as in fig.2.3(a) corresponding with equation (2.4). Each path from the root vertex to any terminal vertex corresponds to either an on-set or off-set product term. In fig.2.3(a), there are three paths A, B and C, from the root vertex to "1" terminal vertex. They correspond to three on-set products, $\bar{x}_2\bar{x}_0$, $\bar{x}_2\bar{x}_1x_0$, and $\bar{x}_3x_2\bar{x}_1x_0$ respectively. Putting these products together leads to another form F_5 of this function f.

$$F_5 = \bar{x}_2\bar{x}_0 + \bar{x}_2\bar{x}_1x_0 + \bar{x}_3x_2\bar{x}_1x_0 \tag{2.6}$$

Notice there are two common edges between path B and C. Thus another multilevel form can be obtained as follows.

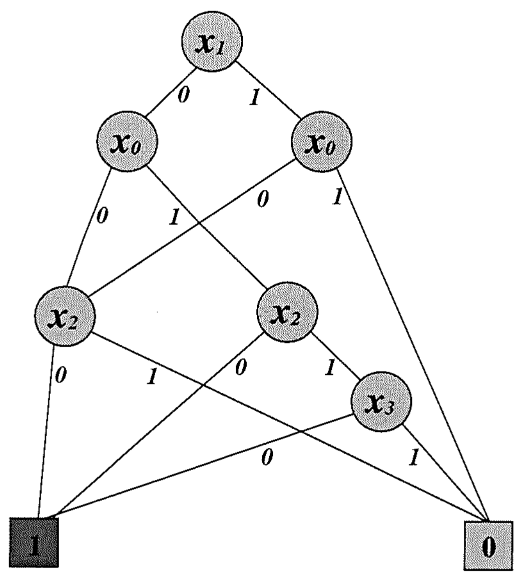
$$F_6 = \bar{x}_2\bar{x}_0 + \bar{x}_1x_0(\bar{x}_2 + \bar{x}_3x_2) \tag{2.7}$$

There are several operations directly manipulated on BDDs, RESTRICTION, COMPOSITION, SATISFY, and APPLY etc[31]. The most complex operation is APPLY through which



$f = (\bar{x}_2 + \bar{x}_3 x_0)(\bar{x}_1 + \bar{x}_0)$
$(\bar{x}_2 + \bar{x}_3)\bar{x}_1$
$\bar{x}_2$
$\bar{x}_2 + \bar{x}_3$
$\bar{x}_3$
A
B
C

(a) with variable order (x0,x1,x2,x3)



(b) with variable order (x1,x0,x2,x3)

Figure 2.3: BDD representation of the Boolean function in example 2.1

two BDDs, G_1 and G_2 are combined as one BDD, $G = G_1 <op> G_2$ where *op* is any Boolean operation.

A BDD can be simplified by the following two rules:

1. If $low(v) = high(v)$, then v can be deleted;

2. If the subgraphs rooted by v and v' are isomorphic, then v and v' can be merged to one vertex.

A reduced BDD that has been simplified by these two rules is unique for any Boolean function if the variable order is fixed[31]. In fig.2.3(b), another BDD is shown for the same function as in fig.2.3(a) but with different variable order. It can be seen that fig.2.3(b) has one more vertex than fig.2.3(a). The unique BDD corresponding with a fixed variable order is called reduced ordered BDD(ROBDD). Therefore, the equivalence of two Boolean functions can be checked through ROBDDs. This technique has been incorporated into SIS[134].

A logic circuit can be constructed from a BDD if each node is replaced by a multiplexor(MUX). The circuit corresponding with fig.2.3(b) is shown in fig.2.4.

A BDD with less node number corresponds with a simpler logic circuit realized with MUXes. Hence the minimization of the number of nodes of a BDD is quite important[109].

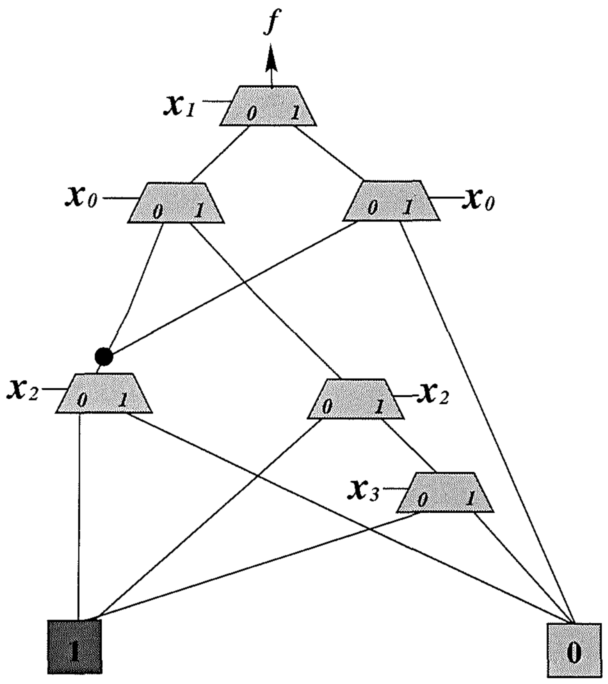


Figure 2.4: Logic circuit from fig.2.3(b)

Unfortunately, the size of a BDD, that is its node number, is very sensitive to the ordering of variables. There have been extensive research on the variable ordering of BDDs[7, 9-11, 16, 34, 55, 80, 110, 122, 159]. Furthermore, some functions do not produce efficient results when processed using BDDs. For example, the sizes of the BDDs for representing arithmetic functions such as multiplication are known to increase exponentially with the number of input variables[75]. Therefore, a lot of other formats of decision diagrams, such as reversed ROBDDs[32], hybrid DDs[40], output value array graphs(OVAG)[82], edge-valued BDDs[89] and partitioned ROBDDs [107], have been proposed for different types of Boolean functions that were surveyed in [30, 52]. The research on testability for BDDs can be found in [33].

2.4 FPGA approach

The first type of user-programmable chip that could implement Boolean functions was the Programmable Read-Only Memory(PROM) introduced at the beginning of 1970s. Simple logic functions can be created using PROMs as a look-up table which stores the truth table of the function. The function inputs are connected to the address lines and the function truth table is programmed into the memory array for each function output. Field-Programmable Logic Array(FPLA) or simply PLA, shown as in fig.1.6(a), was later developed specially for implementing large logic circuits by Signetics in 1975, where both AND and OR planes are programmable. In section 1.3, it is discussed that multilevel logic networks usually offer more general structures and better solutions for large complex circuits. Therefore, Field-Programmable Gate Array (FPGA) was first introduced by the Xilinx company in 1985[139] to meet the general multilevel structures. FPGAs have much more density on a chip than PLAs with the technological evolution while the reprogramming time and cost are drastically reduced comparing with mask programmable gate arrays(MPGAs). The FPGA market has expanded dramatically with many different competing designs, developed by companies including Actel, Advanced Micro Devices, Algotronix, Altera, AT&T, Cypress, Intel, Lattice, Motorola, Quick Logic, and Texas Instruments etc. A generic FPGA architecture, shown in fig.2.5, consists of an array of logic elements together with an interconnect network which can be configured by the user at the point of application. This kind architecture has a very good correspondence with the definition of Boolean network in fig.2.1. Each node in the Boolean network is mapped into a logic block while the interconnection among the nodes can be configured inside a FPGA. Hence, user programming specifies both the logic function of each block and the connections between the blocks. The programming technologies used in commercial FPGA products include floating gate transistors, anti-fuses, and Static Random Access Memory (SRAM) cells. A brief comparison among these programming techniques is shown in table2.1[83].

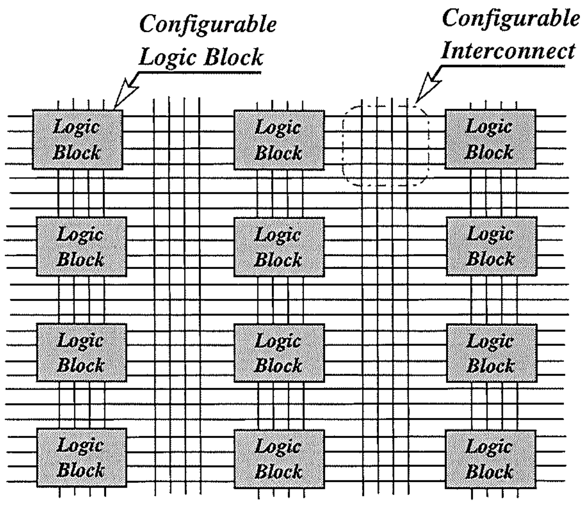


Figure 2.5: Typical FPGA architecture

Technique	Volatile Storage	Series Resistance(Ω)	Relative Capacitance	Relative Cell Area
EPROM	no	2K	10	1
EEPROM	no	2K	10	2
Anti_Fuse	no	50-500	1.2-5.0	1
SRAM	yes	1K	15	5

Table 2.1: Brief comparison of FPGA programming techniques

The design flow for FPGA chip is different from fig.1.5. Taking an example of Xilinx XACT design system, the description file after logic synthesis must first be translated into Xilinx Net-list Format(XNF), which is understood by Xilinx tools. Then in the technology mapping step, the XNF file is mapped into Xilinx Configurable Logic Blocks (CLBs). This step is very important because a Boolean network can be mapped into different CLBs based on different partitions. In fig.2.6, a Boolean network is partitioned into two different CLBs that lead different structure and cost in FPGA realizations. There has been a lot of research in this area, merging logic optimization with technology mapping[41]. In the next step, automatic placement assigns each CLB a physical location on the chip using simulated annealing algorithm. After the physical placement and routing (P&R) is completed, a BIT file is then created which contains the binary programming data. The final step is to download the BIT file to configure the SRAM bits of the target chip. Since all the logic blocks have been prefabricated on the chip, FPGA designs have been winning over ASICs with its time-to-market, low NRE(non-recurring engineering) fees, and reprogrammable features[14], especially for small volume of products.

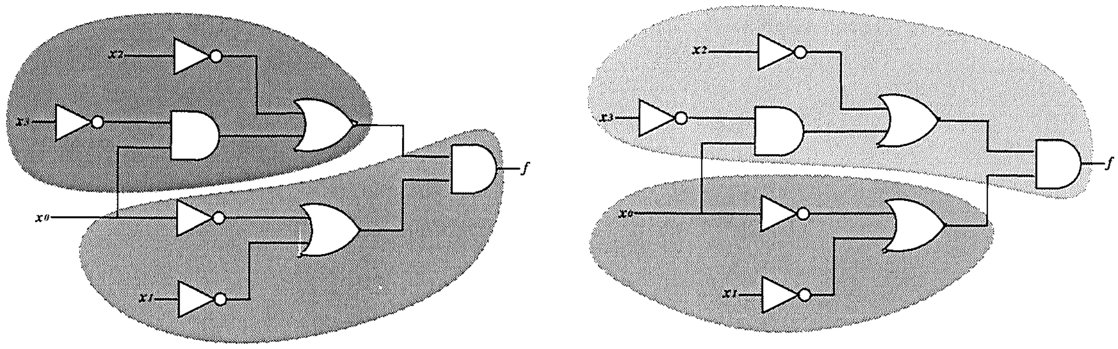


Figure 2.6: Different partitions for the same Boolean network

2.5 Several other approaches based on perturbation

There are some other methods for exploiting don't cares in Boolean networks. Muroga proposed transduction method, an acronym for transformation and reduction, based on the concept of permissible functions[105]. If replacing a node of function f in a Boolean network with another node of function g applying DCs of the network does not change the output, then g is called a permissible function for f. After the replacement, the network can be transformed either locally or globally so that some redundant part in the network can be removed. These transformations and reductions are repeated until no further improvement is possible without changing the network outputs. These ideas were employed in the design system, SYLON[106] for CMOS circuit design.

The idea of logic perturbation by rearranging the structure of the network without affecting its behavior, has been further applied for both combinational and sequential circuits[45]. With the help of efficient automatic test pattern generation (ATPG) techniques, redundancy addition and removal method was proposed based on perturbation in [39]. Some other results about perturbation can be found in [160, 161].

Chapter 3

Multilevel Logic Minimization Using Functional Don't Cares

3.1 Introduction

Multilevel logic synthesis is a known difficult problem although it can produce better results than two-level logic synthesis methods. In [90], multilevel prime implicants are first generated from a Boolean function, then a minimal form of these prime implicants is selected by solving a covering problem using any method for conventional two-level minimization. As an exact optimization method, it is only suitable for small functions due to the high computational complexity. In the last two decades, many heuristic methods have been proposed[24-25, 37, 50-51, 74, 100, 134, 160-161]. In most algorithmic methods, algebraic division plays an important role to decompose a function. Unfortunately, algebraic division applies the distribute law only, neglecting other useful properties of Boolean algebra. Besides, DCs can not be used[100]. As for its counterpart, Boolean division, there is no effective algorithm to find a good divisor. To compute the quotient for a given divisor, a large amount of implicit don't cares, SDCs(satisfiability don't cares) and ODCs(observability don't cares) should be generated and then a two-level minimizer[50] is used. This approach has rarely been used because of its complexity[37, 51]. Moreover, both of these kinds of divisions depend largely on the initial expressions[25]. Consequently, there are different standard scripts, as used in SIS[134], that can give quite different results for the same problem. Sometimes, further running a script may deteriorate the result. Other Boolean methods can be found in [25, 74, 87, 100, 160].

Traditionally, the systematic approach to multilevel logic synthesis is known as functional decomposition[43]. The main problem with this approach is to find the minimum column multiplicity for a bound set of variables based on simple disjunctive decomposition, multiple disjunctive decomposition and some more complicated decomposition methods [112]. A renewed interest in functional decomposition is caused by the introduction of

Look-Up table FPGAs recently[41]. There, a given switching function is broken down into a number of smaller subfunctions so that it can be implemented by the basic blocks of the FPGAs.

The DCs discussed in this chapter are based on the functionality while the implicit SDCs and ODCs are based on the topology of a Boolean network. In addition, these functional DCs are different from the DCs generated on variable-entered Karnaugh maps[2, 71]. A new efficient method to apply these functional DCs for multilevel logic synthesis is proposed according to a criterion based on the number of literals.

3.2 Multilevel Karnaugh map technique

Any Boolean function can be decomposed by Shannon expansion as follows,

$$\begin{aligned} f &= x_i f|_{x_i=1} + \bar{x}_i f|_{x_i=0} && (3.1)\\ &= (x_i + f|_{x_i=0})(\bar{x}_i + f|_{x_i=1}) && (3.2) \end{aligned}$$

where $f|_{x_i=0}$ and $f|_{x_i=1}$ are the cofactors of f with respect to x_i and $\bar{x}_i$. In the Karnaugh map, the effect of the decomposition of equation (3.1) is to split the map into two sub-maps of equal dimensions covered by cubes x_i and $\bar{x}_i$. The split is applied recursively until no off-set is covered based on the different orders of variables. All these cubes of the sub-maps are linked by the OR operation which leads to the expression of the function. This is the traditional two-level Karnaugh map technique where no DCs are generated after the selection of a cube[2]. It is similar to the decomposition of equation (3.2). The main purpose for two-level logic minimization is to find the least number of cubes and literals in each cube, that is equivalent to the best order of variables to decompose the function. For example, $f(x_3x_2x_1x_0) = \sum(2,3,5,7,11,13,15)$ is shown in fig.3.1(a), where the blank entries on the map are "0" outputs. Splitting this map recursively with respect to x_0 and x_1 leads to a cube x_1x_0, which covers no off-set but 4 on-set minterms. Similarly, splitting this map recursively with respect to x_0, x_2 and x_1, $\bar{x}_2$, $\bar{x}_3$ respectively leads to two cubes x_2x_0 and $\bar{x}_3\bar{x}_2x_1$, which covers no off-set but all the remaining on-set minterms. Therefore, this function can be expressed by linking these three cubes with the "OR" operation as in equation (3.3).

$$f(x_3x_2x_1x_0) = x_1x_0 + x_2x_0 + \bar{x}_3\bar{x}_2x_1 \tag{3.3}$$

In the previous two-level minimization, no DCs are generated. Actually, after the selection of a cube, all the on-sets covered by this cube can be used as DCs for the subsequent minimization. This idea will be generalized to multilevel cubes as will be shown in definition 3.1 and will consequently lead to multilevel forms. In [93], a concept of pseudocube is presented that has a more general form than the two-level cube since the XOR operation is incorporated. Any arbitrary Boolean function can then be expressed as a three-level, AND-XOR-OR form that has less literal number than the standard two-level form of sum of products(SOP) in general. A pseudocube enjoys some useful properties but still covers "1" entries only on a Karnaugh map. In this chapter, the concept of a cube on a Karnaugh map is generalized in the following definition.

Definition 3.1. A multilevel cube on a Karnaugh map is the same as a two-level cube except that it can cover both "0" and "1" entries, where "DC" is used to indicate a don't care minterm which can be either "0" or "1".

Definition 3.2. A Karnaugh map M will produce its complemented Karnaugh map M' by interchanging the entries of "0" and "1" while DCs remain the same.

Theorem 3.1. *Given a Karnaugh map of an incompletely specified single output n-variable function* $f(x_{n-1}, x_{n-2}, \cdots x_0)$, *and a multilevel cube* $c = \prod_{i=0}^{k} x_i = \dot{x}_k \dot{x}_{k-1} \cdots \dot{x}_0$, *that may cover the entries of "1", "0" and "DC",* $0 \leq k \leq n-1$, $\dot{x}_i \in \{x_i, \bar{x}_i\}$, $x_i \in \{x_{n-1}, x_{n-2}, \cdots x_0\}$, *then* f *can be decomposed as in equation (3.4) or (3.5).*

$$\begin{aligned} f &= cf_1 + f_2 \\ &= \dot{x}_k \cdots \dot{x}_1 \dot{x}_0 f_1 + f_2 \\ &= \dot{x}_k \cdots \dot{x}_1 \dot{x}_0 \bar{f}_3 + f_2 \end{aligned} \tag{3.4}$$

$$\begin{aligned} f &= (c + f_4) f_5 \\ &= (\dot{x}_k \cdots \dot{x}_1 \dot{x}_0 + f_4) f_5 \\ &= (\dot{x}_k \cdots \dot{x}_1 \dot{x}_0 + \bar{f}_6) f_5 \end{aligned} \tag{3.5}$$

In equation (3.4), f_1 *is a function of n-k-1 variables whose Karnaugh map is exactly the n-k-1 dimensional sub-map that is inside cube c; the Karnaugh map of* f_2 *is the same as* M *except that any entry of "1" covered by cube c can be taken as "DC"; and the Karnaugh map of* f_3 *is the complemented Karnaugh map of* f_1*. In equation (3.5),* f_4 *is the function whose Karnaugh map is exactly the sub-map covered by cube* $\bar{c}$*; the Karnaugh map of* f_5 *is the same as* M *except that any entry of "0" covered by cube* $\bar{c}$ *can be taken as "DC"; and the Karnaugh map of* f_6 *is the complemented Karnaugh map of* f_4*.*

Proof. We will proof equation (3.4) only since equation (3.5) is the dual form of equation (3.4). Moreover, in equation (3.4) only $f = \dot{x}_k \cdots \dot{x}_1 \dot{x}_0 f_1 + f_2$ needs proving on account of definition 3.2.

The entries of "1" on the Karnaugh map M can be divided into two parts:

Part A: entries that are inside cube c;

Part B: entries that are outside cube c.

If all these entries of "1" are covered, then f is realized. From the definitions of f_1 and f_2, all "1" entries in part A are covered by cf_1, all "1" entries in part B are covered by f_2. Now we need to prove that f_1 doesn't include any variables in c.

Because 0 AND $x = 0$, $x \in \{0, 1, DC\}$, all the entries outside of cube c on the map can be used as DCs while minimizing function f_1. Suppose there is a term, $\dot{x}_j\dot{F}$, in the two-level SOP expression of f_1, $\dot{x}_j$ is a literal of the variable x_j, $j \in \{0, 1, \cdots, k\}$, then there must be a cube $\bar{\dot{x}}_j\dot{F}$ that is outside cube $\dot{x}_j\dot{F}$, whose entries can be used as DCs on the Karnaugh map. Let all these DCs inside the cube $\bar{\dot{x}}_j\dot{F}$ be "1" so that we have $\bar{\dot{x}}_j\dot{F} + \dot{x}_j\dot{F} = \dot{F}$. Therefore, deleting $\dot{x}_j$ from $\dot{x}_j\dot{F}$ will not change the function cf_1. From this point, f_1 doesn't include any variable in c. In other words, the Karnaugh map of f_1 is exactly the *n-k-1* dimensional sub-map that is inside cube c.

When all the "1" entries inside cube c have been covered by cf_1, they can be considered to be DCs for minimizing f_2 since $1 + x = 1$, $x \in \{0, 1, DC\}$.

□

Example 3.1. Equation (3.3) is the minimal two-level expression without the aid of DCs for the function shown in fig.3.1(a). Now, apply theorem 3.1 to obtain the multilevel expressions utilizing DCs of functional decomposition. Suppose a multilevel cube x_0 is selected first that covers both "0" and "1" entries. Then f can be expressed as follows based on equation (3.4),

$$f = x_0 f_1 + f_2 \tag{3.6}$$

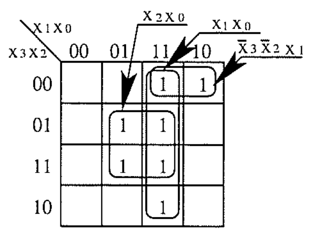
where f_1, as shown in fig.3.1(c), is a three variable function, independent of x_0. Furthermore, all the "1" entries covered by $x_0 f_1$ can be used as DCs that are entries of "$\times$" in fig.3.1(d) to minimize f_2. Selecting a multilevel cube $\bar{x}_2\bar{x}_1$ that covers "0" entries only leads to the expression $f_1 = \overline{\bar{x}_2\bar{x}_1} = x_2 + x_1$ as in fig.3.1(c). Additionally, cube $\bar{x}_3\bar{x}_2 x_1$ in fig.3.1(d) covers the only on-set minterm. Therefore, we have

$$f = x_0(x_2 + x_1) + \bar{x}_3\bar{x}_2 x_1 \tag{3.7}$$

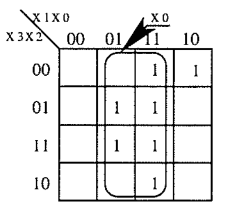
Alternatively, when the multilevel cube x_0 is selected in fig.3.1(b), equation (3.5) can be applied to split the map. Hence,

$$f = (x_0 + f_4) f_5$$

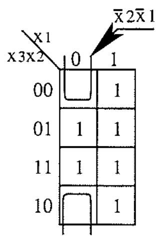
where f_4 is the function whose Karnaugh map is exactly the sub-map covered by cube $\bar{c}$



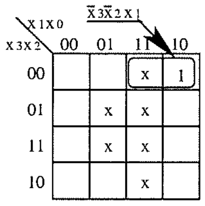
(a) two-level K-map method



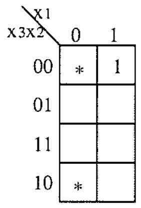
(b) select a multilevel cube



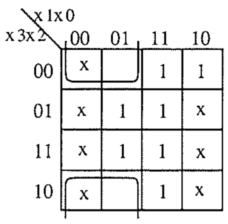
(c) K-map for f1



(d) K-map for f2



(e) K-map for f4



(f) K-map for f5

Figure 3.1: Comparison between two-level and multilevel K-map methods

and all the "0" entries covered by $\bar{c}$ can be used as DCs for f_5. Hence, the Karnaugh maps for f_4 and f_5 are shown in fig.3.1(e) and (f). Therefore we have another expression for f as follows.

$$\begin{aligned} f &= (x_0 + \bar{x}_3\bar{x}_2x_1)\overline{\bar{x}_2\bar{x}_1} \\ &= (x_0 + \bar{x}_3\bar{x}_2x_1)(x_2 + x_1) \end{aligned}$$

If the property of "containment" of cofactors, which will be discussed in the next section, is applied, two new "DC" entries for f_4 are generated. These are marked as "*" in fig.3.1(e). Consequently, this function can be further simplified as

$$f = (x_0 + \bar{x}_3\bar{x}_2)(x_2 + x_1)$$

The following are some properties that are peculiar to a multilevel cube on a Karnaugh map.

1. A multilevel cube can cover "1" , "DC", and "0" giving more degrees of freedom and a multilevel expression.

2. The different orders of multilevel cubes during manipulation on a Karnaugh map may lead to different expressions for the same function since each cube can generate different DCs. For example, the DCs on the Karnaugh map of f_2 and f_5 in fig.3.1(d) and (f) can only be generated after the cube x_0 has been selected.

3. For the same multilevel cube, expansions of equation (3.4) and (3.5) may lead to expressions of different literal numbers for the same function due to the different DCs generated. This will be further verified in the next section.

Theorem 3.2. *If all the "1" entries of a Karnaugh map M for an incompletely specified single output n-variable function $f(x_{n-1}x_{n-2}\cdots x_0)$ or of its complemented Karnaugh map M' can be covered by a traditional two-level prime implicant $p = \prod_{i=0}^{k} x_i = \dot{x}_k\dot{x}_{k-1}\cdots\dot{x}_0$, $\dot{x}_i \in \{x_i, \bar{x}_i\}$, $x_i \in \{x_{n-1}, x_{n-2}\cdots x_0\}$, $0 \le k \le n-1$, then $\dot{x}_k\dot{x}_{k-1}\cdots\dot{x}_0$ is the minimal form of f in the sense that there is no other multilevel expression of f whose literal number is less than $k+1$.*

Proof. We need only to proof the part for M since the counterpart for M' is obvious based on De Morgan law.

Suppose there is a multilevel cube method to find an expression for f whose literal number is less than $k+1$ by first selecting a multilevel cube c whose literal number is j, $1 \le j \le k$. Hence c covers p and at least one "0" entry; otherwise, c is a prime rather than p. Moreover, there are $k+1-j$ literals, $\ddot{x}_{k-j}, \ddot{x}_{k-j-1}, \cdots, \ddot{x}_0$, that exist in p rather than in c. Since c covers at least one "0" entry, equation (3.8) gives the only decomposition that will realize function f,

$$f = cf_1 \tag{3.8}$$

where the Karnaugh map of f_1 is the sub-map that is inside the cube c by theorem 3.1. The literal number of cf_1 is $j + |f_1|$, where $|f_1|$ is the literal number of any expression for f_1. From the previous supposition, it can be deduced that $|f_1| < k+1-j$. In other words, there is at least one literal among $\ddot{x}_{k-j}, \ddot{x}_{k-j-1}, \cdots, \ddot{x}_0$, that will not appear in f_1. Suppose this literal is $\dot{\tilde{x}}$, $\tilde{x} \in \{x_{n-1}, x_{n-2}, \cdots, x_0\}$. From equation (3.8), it can be concluded that f is independent of the variable $\tilde{x}$. Therefore, deleting the variable $\tilde{x}$ from p will not change its functionality. This conflicts with the proposition that p is a prime. □

Based on theorem 3.2, we have the following procedure to simplify a Boolean function in a multilevel form on a Karnaugh map.

Procedure 3.1. *An incompletely specified single output Boolean function f can be expressed by a multilevel form through the following recursive steps on a Karnaugh map.*

1. Let M be the initial Karnaugh map of f.

2. For a Karnaugh map M, if all the "1" (or "0") entries of M can be covered by a two-level prime p, then return p (or $\bar{p}$) as a minimal expression of M. Otherwise, select a suitable multilevel cube, c.

3. From theorem 3.1, M is split into two sub-maps by c, M_1 and M_2. Additionally, select equation (3.4) or (3.5) to decompose and generate new DCs on the sub-maps M_1 and M_2.

4. For both M_1 and M_2, go to step 2 to obtain their expressions f_{M_1} and f_{M_2} with the aid of new DCs.

Example 3.2. Find a multilevel expression for an incompletely specified 4-variable function f whose Karnaugh map is shown in fig.3.2(a) where each blank entry means "0".

1. In fig.3.2(a), M is the initial Karnaugh map.

2. According to step 2 in procedure 3.1, neither *1s* nor *0s* can be covered by a two-level prime, so a multilevel cube $\bar{x}_3$ is selected to cover the entries of "1".

3. After selecting a cube, $\bar{x}_3$ in fig.3.2 (b), M is split into two Karnaugh maps, M_1 and M_2 according to step 3 in procedure 3.1. Because cube $\bar{x}_3$ covers all the "1" entries, f_{M_2} is 0 while M_1 is shown in fig.3.2 (c). This lead to the expression $f = \bar{x}_3 f_{M_1}$ based on decomposition equation (3.4).

4. Now M is M_1 and go to step 2. In the same way, there is no prime to cover all the entries of either "1" or "0" in fig.3.2 (c). Thus a multilevel cube x_1 is selected according step 2. Consequently, the Karnaugh map is split into two sub-maps, M_{11} and M_{12}. Selecting equation (3.5) to decompose, we have $f_{M_1} = (x_1 + f_{M_{11}})f_{M_{12}}$ where the Karnaugh maps for $f_{M_{11}}$ and $f_{M_{12}}$ can be obtained in fig.3.2 (d) and (e) based on theorem 3.1. Notice that all the "0" entries covered by $\bar{x}_1$ can be used as DCs for $f_{M_{12}}$.

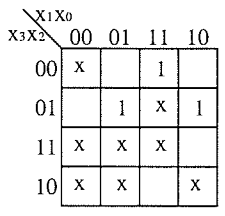
5. Now M is M_{11} as in fig.3.2 (d). There is a prime, x_2x_0, that covers all the entries of "1". Hence $f_{M_{11}} = x_2x_0$. In the same way, when M is M_{12}, there is also a prime, $\bar{x}_2\bar{x}_0$ to cover all the entries of "0", that gives the result, $f_{M_{12}} = \overline{\bar{x}_2\bar{x}_0} = x_2 + x_0$.

The previous steps give the following expression of f,

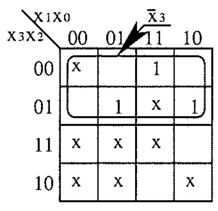
$$f = \bar{x}_3(x_1 + x_2x_0)(x_2 + x_0) \tag{3.9}$$

From example 3.2, procedure 3.1 gives a simple and convenient method to obtain the multilevel form of a Boolean function based on Karnaugh map. It offers a good opportunity to visualize the multilevel minimization process and to understand the functional decomposition technique. However, it has following disadvantages:

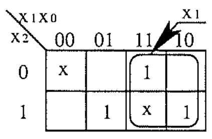
1. It is impractical for large functions;



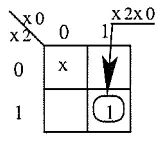
(a) original K-map



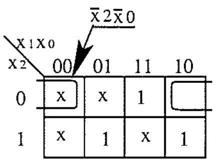
(b) select a multilevel cube



(c) select a multilevel cube on M1



(d) select a prime on M11



(e) select a prime on M12

Figure 3.2: Multilevel K-map method for an incompletely specified function

2. It still depends on personal experience to select a multilevel cube in step 2 of procedure 3.1 and to choose one decomposition equation from (3.4) and (3.5) to split in step 3. This is difficult for automation.

3. There are more implicit DCs of functional decomposition that have not been exploited as mentioned in example 3.1.

Hence, procedure 3.1 requires further refinement as discussed in section 3.3.

3.3 Multilevel logic synthesis for large Boolean functions

Definition 3.3. Any incompletely specified Boolean function $f(x_{n-1}, x_{n-2}, \cdots, x_0)$ can be defined uniquely as two minterm sets, f_{ON} and f_{DC}, representing the on-set and DC-set minterms of f respectively. For a literal $\dot{x}_i$ of a certain variable x_i, $0 \leq i \leq n-1$, if $[(f|_{\dot{x}_i=1})_{DC} \cup (f|_{\dot{x}_i=1})_{ON}] \supset (f|_{\dot{x}_i=0})_{ON}$, where "$\cup$" and "$\supset$" are set operations of "union" and "proper inclusion" respectively, then it is said that $f|_{\dot{x}_i=1}$ contains $f|_{\dot{x}_i=0}$ or there is containment of f upon the literal $\dot{x}_i$. If there is no containment upon x_i or $\bar{x}_i$, then it is said there is no containment upon variable x_i. If there is containment upon both x_i and $\bar{x}_i$, then it is said there is containment upon variable x_i. For convenience, this property can be called the containment of cofactors.

	X3X2X1	X0
2	**0 0 1**	**0**
3	0 0 1	1
5	0 1 0	1
7	0 1 1	1
11	1 0 1	1
13	1 1 0	1
15	1 1 1	1

f_{ON}

(a) minterm array of ex.3.1

	X3X2X1
1	0 0 1
2	0 1 0
3	0 1 1
5	1 0 1
6	1 1 0
7	1 1 1

$f|_{X0=1}$

	X3X2X1
1	0 0 1

$f|_{X0=0}$

(b) minterm arrays after splitting by x0 for ex.3.1

	X3X2X1	X0
3	0 0 1	1
5	0 1 0	1
6	**0 1 1**	**0**

f_{ON}

	X3X2X1	X0
0	**0 0 0**	**0**
7	0 1 1	1
8	**0 1 1**	**0**
9	1 0 0	1
10	**1 0 1**	**0**
12	**1 1 0**	**0**
13	1 1 0	1
15	1 1 1	1

f_{DC}

(c) minterm arrays of ex.3.2

	X3X2X1
1	0 0 1
2	0 1 0

$(f|_{X0=1})_{ON}$

	X3X2X1
3	0 1 1
4	1 0 0
6	1 1 0
7	1 1 1

$(f|_{X0=1})_{DC}$

	X3X2X1
3	0 1 1

$(f|_{X0=0})_{ON}$

	X3X2X1
0	0 0 0
3	0 1 1
5	1 0 1
6	1 1 0

$(f|_{X0=0})_{DC}$

(d) minterm arrays after splitting by x0 for ex.3.2

Figure 3.3: Two examples for definition 3.3

From definition 3.3, a Boolean function and its minterm sets will be used interchangeably hereafter. In example 3.1, $f_{ON} = \{2, 3, 5, 7, 11, 13, 15\}$ as shown in fig.3.3(a), $f_{DC} = \emptyset$. From fig.3.3(a), the value of x_0 in any minterm of $\{3, 5, 7, 11, 13, 15\}$ is "1" while the value of x_0 in the minterm "2" is zero. After deleting x_0 from all the minterms in fig.3.3(a), we obtain $f|_{x_0=1} = \{1, 2, 3, 5, 6, 7\}$, $f|_{x_0=0} = \{1\}$ as shown in fig.3.3(b). Hence $f|_{x_0=1}$ contains $f|_{x_0=0}$ or there is containment of f upon the literal x_0 because of $f|_{x_0=1} \supset f|_{x_0=0}$ based on definition 3.3. In example 3.2 whose minterm arrays are shown in fig.3.3(c), $f_{ON} = \{3, 5, 6\}$, $f_{DC} = \{0, 7, 8, 9, 10, 12, 13, 15\}$, in the same way, we can calculate the results as in fig.3.3(d), $(f|_{x_0=1})_{ON} = \{1, 2\}$, $(f|_{x_0=1})_{DC} = \{3, 4, 6, 7\}$, $(f|_{x_0=0})_{ON} = \{3\}$, $(f|_{x_0=0})_{DC} = \{0, 3, 5, 6\}$. Therefore, $f|_{x_0=1}$ contains $f|_{x_0=0}$ but there is no containment of f upon the literal $\bar{x}_0$ since $(f|_{\bar{x}_0=0})_{ON}$ is not properly included in the union set of $(f|_{\bar{x}_0=1})_{ON} \cup (f|_{\bar{x}_0=1})_{DC}$.

For comparison, there are two concepts of "containment" in the two-level minimization technique. One is the single-cube containment where a cube c contains another cube d of the cover[100]. So cube d is redundant and should be deleted to achieve the cover with single cube containment minimality. The other can be called multiple-cube containment where a cube c is contained by several relatively essential cubes. This cube c is called totally redundant cube and will be deleted from the cover[50]. These ideas have been used in the procedure "IRREDUNDANT_COVER" of ESPRESSO[26]. However, the containment in definition 3.3 has more general meaning because it is a relation of cofactors instead of one or several cubes only. In other words, the containment of cofactors is one of the properties of the structure of Boolean functions. What is more important is that a Boolean function with containment implies new DCs of functional decomposition as will be shown in theorem 3.3. Consequently these DCs offer more degrees of freedom for logic minimization rather than just deletion of the redundant cubes.

Theorem 3.3. *For an incompletely specified Boolean function $f(x_{n-1}, x_{n-2}, \cdots, x_0)$, if there is containment upon a literal $\dot{x}_i$, $0 \leq i \leq n-1$, then f can be decomposed as follows,*

$$\begin{aligned} f &= \dot{x}_i f|_{\dot{x}_i=1} + \check{f}|_{\dot{x}_i=0} &(3.10)\\ &= (\dot{x}_i + f|_{\dot{x}_i=0})\check{f}|_{\dot{x}_i=1} &(3.11)\end{aligned}$$

In equation (3.10), all the minterms in $(f|_{\dot{x}_i=0})_{ON}$ can be used as DCs for $f|_{\dot{x}_i=1}$, and $\check{f}|_{\dot{x}_i=0}$ is the same as $f|_{\dot{x}_i=0}$ except that all the minterms in $(f|_{\dot{x}_i=0})_{DC}$ which are not in $(f|_{\dot{x}_i=1})_{ON} \cup (f|_{\dot{x}_i=1})_{DC}$ are deleted. In equation (3.11), all the minterms that are not in $(f|_{\dot{x}_i=1})_{ON} \cup (f|_{\dot{x}_i=1})_{DC}$ can be used as DCs for $f|_{\dot{x}_i=0}$, and $\check{f}|_{\dot{x}_i=1}$ is the same as $f|_{\dot{x}_i=1}$ except that all the minterms in $(f|_{\dot{x}_i=1})_{DC}$ which are also in $(f|_{\dot{x}_i=0})_{ON}$ are moved to $(f|_{\dot{x}_i=1})_{ON}$.

Proof. If there is containment upon a literal $\dot{x}_i$, from definition 3.3, we have $[(f|_{\dot{x}_i=1})_{DC} \cup (f|_{\dot{x}_i=1})_{ON}] \supset (f|_{\dot{x}_i=0})_{ON}$. If all the minterms in $(f|_{\dot{x}_i=0})_{DC}$ which are not in $(f|_{\dot{x}_i=1})_{ON} \cup (f|_{\dot{x}_i=1})_{DC}$ are deleted as required in the theorem, then we have,

$$[(f|_{\dot{x}_i=0})_{DC} \cup (f|_{\dot{x}_i=0})_{ON}] \supseteq [(\check{f}|_{\dot{x}_i=0})_{DC} \cup (f|_{\dot{x}_i=0})_{ON}] \tag{3.12}$$

$$[(f|_{\dot{x}_i=1})_{DC} \cup (f|_{\dot{x}_i=1})_{ON}] \supset [(f|_{\dot{x}_i=0})_{DC} \cup (f|_{\dot{x}_i=0})_{ON}] \tag{3.13}$$

Because the on-set minterms of $f|_{\dot{x}_i=0}$ are not deleted from $\check{f}|_{\dot{x}_i=0}$ and DCs can be set to "0", we have equation (3.14) from equation (3.12).

$$\bar{\dot{x}}_i f|_{\dot{x}_i=0} = \bar{\dot{x}}_i \check{f}|_{\dot{x}_i=0} \tag{3.14}$$

From equation (3.13), we have,

$$\dot{x}_i f|_{\dot{x}_i=1} = \dot{x}_i f|_{\dot{x}_i=1} + \dot{x}_i \check{f}|_{\dot{x}_i=0} \tag{3.15}$$

Therefore, equation (3.1) can be rewritten as below based on equations (3.14) and (3.15),

$$\begin{aligned} f &= \dot{x}_i f|_{\dot{x}_i=1} + \bar{\dot{x}}_i f|_{\dot{x}_i=0} && (3.16)\\ &= \dot{x}_i f|_{\dot{x}_i=1} + \dot{x}_i \check{f}|_{\dot{x}_i=0} + \bar{\dot{x}}_i \check{f}|_{\dot{x}_i=0} && (3.17)\\ &= \dot{x}_i f|_{\dot{x}_i=1} + \check{f}|_{\dot{x}_i=0} && (3.18)\end{aligned}$$

Therefore equation (3.10) is proved. From equation (3.10), all the minterms in $(f|_{\dot{x}_i=0})_{ON}$ are covered by $\check{f}|_{\dot{x}_i=0}$. Since $(f|_{\dot{x}_i=0})_{ON}$ covers both $\dot{x}_i(f|_{\dot{x}_i=0})_{ON}$ and $\bar{\dot{x}}(f|_{\dot{x}_i=0})_{ON}$, all the minterms in $(f|_{\dot{x}_i=1})_{ON}$ that are covered by $(f|_{\dot{x}_i=0})_{ON}$ can be used as DCs for $f|_{\dot{x}_i=1}$ because of $1 + x = 1$, $x \in \{0, 1, DC\}$. This point can also be seen in equations (3.16)-(3.18).

Similarly, if there is containment upon a literal $\dot{x}_i$, from definition 3.3, we have

$$[(f|_{x_i=1})_{DC} \cup (f|_{x_i=1})_{ON}] \supset (f|_{x_i=0})_{ON}$$

If all the minterms in $(f|_{\dot{x}_i=1})_{DC}$ that are in $(f|_{\dot{x}_i=0})_{ON}$ are moved to $(f|_{\dot{x}_i=1})_{ON}$ as required in the theorem, then we have,

$$(\check{f}|_{\dot{x}_i=1})_{ON} \supseteq (f|_{\dot{x}_i=1})_{ON} \tag{3.19}$$

$$(\check{f}|_{\dot{x}_i=1})_{ON} \supset (f|_{\dot{x}_i=0})_{ON} \tag{3.20}$$

Because DCs can be set to "1", we have the following equation from (3.19),

$$\bar{\dot{x}}_i + f|_{\dot{x}_i=1} = \bar{\dot{x}}_i + \check{f}|_{\dot{x}_i=1} \tag{3.21}$$

Besides, all the DCs in $f|_{\dot{x}_i=0}$ can be set to "0", equation (3.22) can be obtained from equation (3.20).

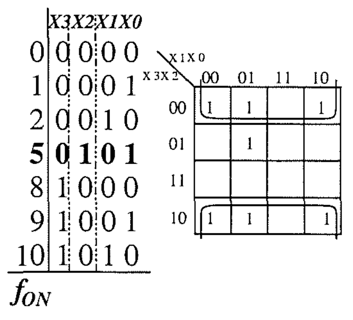
$$\dot{x}_i + f|_{\dot{x}_i=0} = (\dot{x}_i + f|_{\dot{x}_i=0})(\dot{x}_i + \check{f}|_{\dot{x}_i=1}) \tag{3.22}$$

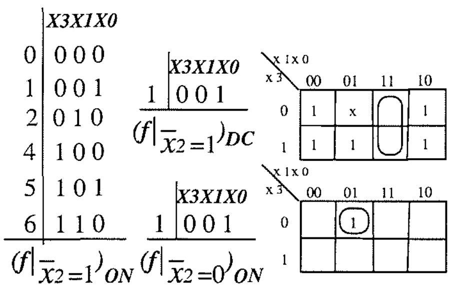
Therefore, equation (3.2) can be rewritten as below based on equations (3.21) and (3.22),

$$\begin{aligned} f &= (\dot{x}_i + f|_{\dot{x}_i=0})(\bar{\dot{x}}_i + f|_{\dot{x}_i=1}) && (3.23)\\ &= (\dot{x}_i + f|_{\dot{x}_i=0})(\dot{x}_i + \check{f}|_{\dot{x}_i=1})(\bar{\dot{x}}_i + \check{f}|_{\dot{x}_i=1}) && (3.24)\\ &= (\dot{x}_i + f|_{\dot{x}_i=0})\check{f}|_{\dot{x}_i=1} && (3.25) \end{aligned}$$

Hence equation (3.11) is proved. From equation (3.11), all the off-set minterms of $\check{f}|_{\dot{x}_i=1}$ can be used as DCs for $f|_{\dot{x}_i=0}$ because $0 \text{ AND } x = 0$, $x \in \{0, 1, DC\}$. Therefore, all the minterms that are not in $(f|_{\dot{x}_i=1})_{ON} \cup (f|_{\dot{x}_i=1})_{DC}$ can be used as DCs for $f|_{\dot{x}_i=0}$. This point can also be seen from equations (3.23)-(3.25).

This completes the proof of theorem 3.3. □


(a) minterm array and K-map of ex.3


(b) select equation (14) to split by x2'

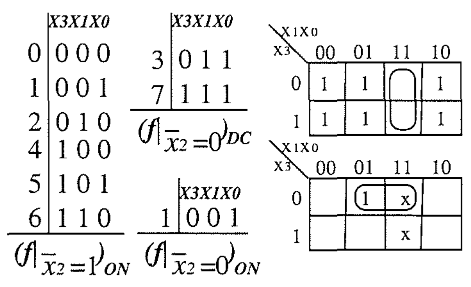

(c) select equation (15) to split by x2'

Figure 3.4: Examples of theorem 3.3

Example 3.3. A completely specified Boolean function $f(x_3x_2x_1x_0) = \Sigma(0,1,2,5,8,9,10)$ is shown in fig.3.4(a) both in minterm array and Karnaugh map forms for better understanding. It can be calculated that $(f|_{\bar{x}_2=1})_{ON} = \{0,1,2,4,5,6\}$; $(f|_{\bar{x}_2=0})_{ON} = \{1\}$; $(f|_{\bar{x}_2=1})_{DC} = (f|_{\bar{x}_2=0})_{DC} = \emptyset$ from either the minterm array or the Karnaugh map. So there is containment upon the literal $\bar{x}_2$ according to definition 3.3. Then f can be split by $\bar{x}_2$ based on equation (3.10) or (3.11), that means to select a multilevel cube $\bar{x}_2$ on the Karnaugh map. First select equation (3.10) to split so that $f = \bar{x}_2 f|_{\bar{x}_2=1} + \check{f}|_{\bar{x}_2=0}$. Because $(f|_{\bar{x}_2=0})_{DC} = \emptyset$, no minterm is deleted or $\check{f}|_{\bar{x}_2=0} = f|_{\bar{x}_2=0}$. However, all the minterms in $(f|_{\bar{x}_2=0})_{ON}$ can be used as DCs for $f|_{\bar{x}_2=1}$, that is, $(f|_{\bar{x}_2=1})_{DC} = \{1\}$;$(f|_{\bar{x}_2=1})_{ON} = \{0,2,4,5,6\}$. This result is shown in fig.3.4(b) from which it can be seen that both $f|_{\bar{x}_2=1}$ and $f|_{\bar{x}_2=0}$ can be covered by primes x_1x_0 and $\bar{x}_3\bar{x}_1x_0$ respectively. From theorem 3.2, we obtain $f|_{\bar{x}_2=1} = \overline{x_1x_0}$ and $\check{f}|_{\bar{x}_2=0} = \bar{x}_3\bar{x}_1x_0$ that gives an expression for f as indicated in equation (3.26).

$$f = \bar{x}_2(\overline{x_1x_0}) + \bar{x}_3\bar{x}_1x_0 \tag{3.26}$$

Now equation (3.11) is selected to split f so that $f = (\bar{x}_2 + f|_{\bar{x}_2=0})\check{f}|_{\bar{x}_2=1}$. Because $(f|_{\bar{x}_2=1})_{DC} = \emptyset$, no minterm is moved or $\check{f}|_{\bar{x}_2=1} = f|_{\bar{x}_2=1}$. However, all the minterms that are not in $(f|_{\bar{x}_2=1})_{ON} \cup (f|_{\bar{x}_2=1})_{DC}$ can be used as DCs for $f|_{\bar{x}_2=0}$, that is, $\{3,7\}$ can be used as DCs for $f|_{\bar{x}_2=0}$. So we have, $(f|_{\bar{x}_2=1})_{ON} = \{0,1,2,4,5,6\}$;$(f|_{\bar{x}_2=0})_{ON} = \{1\}$; $(f|_{\bar{x}_2=0})_{DC} = \{3,7\}$ as shown in fig.3.4(c) from which we know that both $\check{f}|_{\bar{x}_2=1}$ and $f|_{\bar{x}_2=0}$ can be covered by primes x_1x_0 and $\bar{x}_3x_0$ respectively. In the same way, another expression for f can be obtained as follows,

$$f = (\bar{x}_2 + \bar{x}_3x_0)\overline{x_1x_0} \tag{3.27}$$

Comparing these two expressions for f, equation (3.27) has one less literal number than equation (3.26). The reason is that different DCs may be generated depending on whether equations (3.10) or (3.11) is selected. However, there are two special cases of containment where it is simple to determine the splitting equations.

Lemma 3.1. *For an incompletely specified Boolean function $f(x_{n-1}, x_{n-2}, \cdots, x_0)$, if all the entries of $\dot{x}_i$ are filled with on-set minterms, then there is containment upon $\dot{x}_i$ and this function can be split by equation (3.28) based on equation (3.10) since $f|_{\dot{x}_i=1} = 1$.*

$$f = \dot{x}_i + f|_{\dot{x}_i=0} \tag{3.28}$$

In the same way, if all the on-set minterms can be covered completely by a literal, $\dot{x}_i$,

$0 \le i \le n-1$, then there is containment upon $\dot{x}_i$ and the function can be split by equation (3.29) based on equation (3.11) since $f|_{\dot{x}_i=0} = 0$. Note that these two conditions will not be met simultaneously in general; otherwise, $f = \dot{x}_i$.

$$f = \dot{x}_i f|_{\dot{x}_i=1} \tag{3.29}$$

For the general case of containment, we have the following heuristics to decide the decomposition equation.

Observation 3.1. *Rewrite equations (3.10) and (3.11) as follows.*

$$f = \dot{x}_i f|_{\dot{x}_i=1} + \check{f}|_{\dot{x}_i=0} \tag{3.30}$$

$$f = (\dot{x}_i + f|_{\dot{x}_i=0})\check{f}|_{\dot{x}_i=1} \tag{3.31}$$

In equation (3.30), all the minterms in $(f|_{\dot{x}_i=0})_{ON}$ can be used as DCs for $f|_{\dot{x}_i=1}$, and $\check{f}|_{\dot{x}_i=0}$ is the same as $f|_{\dot{x}_i=0}$ except that all the minterms in $(f|_{\dot{x}_i=0})_{DC}$ which are not in $(f|_{\dot{x}_i=1})_{ON} \cup (f|_{\dot{x}_i=1})_{DC}$ are deleted. In equation (3.31), all the minterms that are not in $(f|_{\dot{x}_i=1})_{ON} \cup (f|_{\dot{x}_i=1})_{DC}$ can be used as DCs for $f|_{\dot{x}_i=0}$, and $\check{f}|_{\dot{x}_i=1}$ is the same as $f|_{\dot{x}_i=1}$ except that all the minterms in $(f|_{\dot{x}_i=1})_{DC}$ which are also in $(f|_{\dot{x}_i=0})_{ON}$ are moved to $(f|_{\dot{x}_i=1})_{ON}$. Suppose we do not use any of the new DCs to minimize the function f, then the literal number of equation (3.30) is $|(\dot{x}_i f|_{\dot{x}_i=1} + f|_{\dot{x}_i=0})| = 1 + |f|_{\dot{x}_i=1}| + |f|_{\dot{x}_i=0}|$ where $|f|_{\dot{x}_i=1}|$ and $|f|_{\dot{x}_i=0}|$ are the literal numbers of $f|_{\dot{x}_i=1}$ and $f|_{\dot{x}_i=0}$ respectively. Similarly, if we do not use any of the new DCs to minimize the function f, then the literal number of equation (3.31) is $|(\dot{x}_i + f|_{\dot{x}_i=0})f|_{\dot{x}_i=1}| = 1 + |f|_{\dot{x}_i=1}| + |f|_{\dot{x}_i=0}|$. Hence the literal number in equation (3.30) would be the same as in (3.31) if we do not use any of the new DCs. Furthermore, the main DCs with respect to literal number in equation (3.30) is the one for $f|_{\dot{x}_i=1}$ while it is the one for $f|_{\dot{x}_i=0}$ in equation (3.31) from theorem 3.3. So we need only to compare the usefulness of these two main DCs for the equations. Intuitively, if a "DC" minterm is adjacent to an on-set minterm, then we say it is useful for the minimization; otherwise, we say it is not useful. In equation (3.30), the on-set minterms of $f|_{\dot{x}_i=1}$ may be used as DCs while in equation (3.31) the off-set minterms of $f|_{\dot{x}_i=0}$ may be used as DCs. Let α be the number of new generated "DC" minterms that are not adjacent to the on-set minterms of $f|_{\dot{x}_i=1}$; and β be the number of new generated "DC" minterms that are adjacent to the on-set minterms of $f|_{\dot{x}_i=0}$. If $\alpha < \beta$, then the new DCs in equation (3.30) are not so effective as in equation (3.31). Consequently it is better to select equation (3.31) to decompose the function. If $\alpha > \beta$, then the new DCs in equation (3.31) are not so *effective as equation (3.30). Consequently it is better to select equation (3.30) to decompose the function. Otherwise, if $\alpha = \beta$, then select either of them to decompose the function.*

The preceding observation can be justified by the function in example 3.3, where there is a containment upon $\bar{x}_2$. If equation (3.30) is selected to decompose as shown in fig.3.4(b), then there is one generated "DC" minterm but it is adjacent to two on-set minterms in $f|_{\bar{x}_2=1}$. Therefore, α is "0". On the other hand, if equation (3.31) is selected to decompose as shown in fig.3.4(c), then there are two generated "DC" minterms and one of them is adjacent to the on-set minterm in $f|_{\bar{x}_2=0}$. Therefore, β is "1". From observation 3.1, equation (3.31) should be selected to decompose the function that will lead to a smaller literal number.

From theorems 3.2 and theorem 3.3, it is conjectured that a multilevel cube which is nearly prime or covers more on-set minterms has priority to be selected. This idea is expressed in the following conjecture that has already been utilized in the previous examples.

Conjecture 3.4. *For an incompletely specified Boolean function $f(x_{n-1}, x_{n-2}, \cdots, x_0)$, there are 2 literals for each variable x_i, $0 \leq i \leq n-1$. Any of these literals is a n-1 dimensional multilevel cube. Among these 2n literals, a multilevel cube can be selected that covers the most on-set minterms.*

Now procedure 3.2 is obtained to minimize a Boolean function in a multilevel form based on procedure 3.1.

Procedure 3.2. *An incompletely specified Boolean function $f(x_{n-1}, x_{n-2}, \cdots, x_0)$ can be simplified by the following recursive steps on the minterm sets f_{ON} and f_{DC}.*

1. If f is a constant function, then return 0 or 1.

2. If either of the two special cases in lemma 3.1 is satisfied, then return equation (3.28) or (3.29) accordingly and go to step 1 for the subsequent function.

3. Select a literal $\dot{x}_i$ according to conjecture 3.4 to split. This is the important but difficult step. If there are several literals that cover the same number of on-set minterms, then a literal is selected based on the following conditions:

(a). Select a literal that has containment so that new DCs can be generated.

(b). If there are several literals that have containment, then detect whether they are symmetric variables[86]. If they are symmetric, just select the first one. Otherwise, go to the next condition.

(c). If there are still several literals that satisfy the previous conditions, then select the one that has more "DC" minterms; Otherwise, just select the first literal since no other efficient method has been found at present.

4. If there is containment upon $\dot{x}_i$, then select a splitting equation (3.10) or (3.11). This is also the other key step of the procedure. At present observation 3.1 is used to select *the splitting equation. Otherwise, if there is no containment, decompose the function as equation (3.1) or (3.2). For each of the two subsequent functions, go to step 1.*

Example 3.4. The minterm array for a completely specified function $f(x_4x_3x_2x_1x_0) = \sum(0,1,2,4,8,9,10,12,24,25,28)$ is shown in fig.3.5(a). It can be simplified by procedure 3.2 as follows.

1. f is not a constant function. Besides, neither of the special cases in lemma 3.1 is satisfied. So go to step 3 of procedure 3.2.

2. In fig.3.5(a), the on-set minterm numbers covered by the literals x_4, x_3, x_2, x_1, x_0 are "3", "7", "3", "2", "3", shown at the bottom of the corresponding column respectively. According to conjecture 3.4, literal $\bar{x}_1$ is selected because it covers the most on-set minterms.

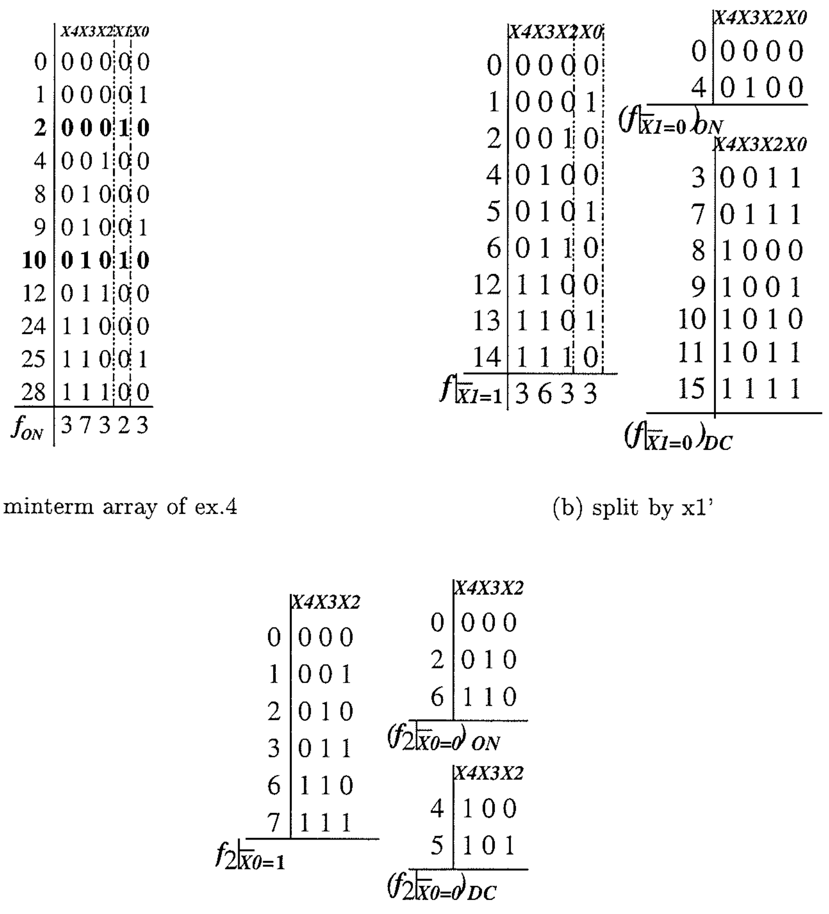
3. After selecting a multilevel cube $\bar{x}_1$, the minterm array is split into two sub-arrays as in fig.3.5(b), where $(f|_{\bar{x}_1=1})_{ON} = \{0,1,2,4,5,6,12,13,14\}$; $(f|_{\bar{x}_1=1})_{DC} = \emptyset$; $(f|_{\bar{x}_1=0})_{ON} = \{0,4\}$; $(f|_{\bar{x}_1=0})_{DC} = \emptyset$. It can be seen that there is containment upon $\bar{x}_1$ based on definition 3.3. If equation (3.10) is selected to split, then there will be two on-set minterms in $f|_{\bar{x}=1}$ that can be used as DCs. But both of them are adjacent to other on-sets. Hence α is "0" based on observation 3.1. On the other hand, if equation (3.11) is selected to split, then there will be 7 off-set minterms in $f|_{\bar{x}=0}$ that can be used as DCs shown in fig.3.5(b). One of them, "8" is adjacent to a on-set minterm "0" in $(f|_{\bar{x}=0})_{ON}$. Hence β is "1". Based on observation 3.1, equation (3.11) is selected to split the function because of $\alpha < \beta$. Thus we have $f = (\bar{x}_1 + f_1)f_2$, where f_1 is $f|_{\bar{x}_1=0}$ including on-set and DC minterms; $f_2 = f|_{\bar{x}_1=1}$ as shown in fig.3.5(b).

4. For $f_1 = f|_{\bar{x}_1=0}$, it is not a constant function, but all the on-set minterms are covered by cubes $\bar{x}_0$, $\bar{x}_2$, and $\bar{x}_4$ that are the special cases in equation (3.29). Hence $f_1 = \bar{x}_0\bar{x}_2\bar{x}_4$.

5. For $f_2 = f|_{\bar{x}_1=1}$ in fig.3.5(b), the on-set minterm numbers covered by the literals $\bar{x}_0$, $\bar{x}_2$, x_3, $\bar{x}_4$ are the same in step 3 of the procedure. Moreover, all of them have containment. So we use a symmetry detection method and find that they are symmetric. From step 3(b) of procedure 3.2, the first literal, $\bar{x}_0$ is selected to split as shown in fig.3.5(c).

6. From step 4 of procedure 3.2, there is containment upon $\bar{x}_0$ in fig.3.5(c). In the same way, it can be calculated that $\alpha = 0$, $\beta = 1$. Based on observation 3.1, selecting equation (3.11) to split gives $f_2 = (\bar{x}_0 + f_2|_{\bar{x}_0=0})f_2|_{\bar{x}_0=1}$. For simplicity, it can be seen that both $f_2|_{\bar{x}_0=0}$ and $f_2|_{\bar{x}_0=1}$ can be covered by two-level primes. Thus, $f_2|_{\bar{x}_0=0} = \bar{x}_2$; $f_2|_{\bar{x}_0=1} = \overline{\bar{x}_3x_4} = x_3 + \bar{x}_4$. Alternatively, applying procedure 3.2 for both $f_2|_{\bar{x}_0=0}$ and $f_2|_{\bar{x}_0=1}$ will produce the same results without the help of the primes.

From the above steps, we achieve the expression $f = (\bar{x}_1 + \bar{x}_0\bar{x}_2\bar{x}_4)(\bar{x}_0 + \bar{x}_2)(x_3 + \bar{x}_4)$.



(a) minterm array of ex.4

(b) split by x1'

(c) split by x0'

Figure 3.5: An example of procedure 3.2

3.4 Experimental results

Procedure 3.2 has been implemented in C language. Given the on-set and DC-set for a single output function, the program will produce a multilevel expression using the equation format defined in [134]. We test all the single output functions from both IWLS'93 benchmark and SIS testcases on a PC with Cyrix6x86-166 CPU and 32M RAM under Linux operating system. For all the single output functions found in the common benchmarks, there are 1299 splits with containment property among total 1813 splits. In other words, more than 70% of splits have the containment property of definition 3.3. The comparison with SIS is shown in table 3.1, where "var#", "min#", and "lit#" are the numbers of variables, minterms, and literals respectively. Besides, the unit of "time" is second and "/" means not available due to memory and time limitation. The program is first compared with the algebraic method and results are shown in column "gcx" (short for "*gcx*; *resub -a*") of table 3.1. There are two testcases, "max46.pla" and "col4.pla", for which our program produces worse results in the term of literal numbers. On the other hand, we want to know if it is possible for our program mainly based on theorem 3.3 to produce very good results for some functions. Hence we compare with "script.rugged" and the results are shown in column "script.rugged" of table 3.1. Among 15 testcases, there are 6 cases that our program produces worse results. But our program is much faster in these cases. Generally speaking, it can still produce the results very quickly although the minterm input files are much larger than .blif or .pla files. For the testcase of parity.pla, where each cube is actually a minterm, both "gcx" and "script.rugged" would require excessive CPU time and memory.

From these experimental results, it can be concluded that the DCs of function decomposition are very useful to minimize the Boolean functions in multilevel forms. For the testcases that our program produces worse results than "gcx" and "script.rugged", there are several reasons as follows.

1. The input files of our program are in minterm format, that have the most number of literals comparing with .pla and .blif formats.

2. There are two heuristic methods in procedure 3.2. One is the selection of variable orders in step 3. The other is observation 3.1, which makes the decision of decomposition equations. Currently the order of variables is decided according to the number of minterms, which is not suitable for some functions. Furthermore, if two cubes cover the same numbers of on-sets and DC-sets respectively with the containment of cofactors, then there is no efficient way to determine the order. As for the decomposition equations, there is some inefficiency in evaluating the usefulness of new DCs based on the adjacency only.

			Functional DCs		gcx		script.rugged	
	var#	min#	lit#	time(s)	lit#	time(s)	lit#	time(s)
t481.blif*	16	42016	928	5.92	3400	79.4	881	240.5
9sym.blif	9	420	239	0.04	604	0.5	275	33.2
9symml.blif	9	420	239	0.04	364	0.4	241	19.2
xor5.blif	5	16	16	~0	97	~0	16	0.5
cm152a.blif	11	1024	26	0.24	54	~0	22	0.2
majority.blif	5	21	10	~0	34	~0	10	~0
parity.blif	16	32768	60	9.29	136	~0	60	0.3
parity.pla	16	32768	60	9.29	/	/	/	/
max46.pla	9	62	215	0.03	189	0.2	174	4.1
newill.pla	8	142	23	~0	28	~0	24	~0
newtag.pla	8	234	11	~0	13	~0	11	~0
ryy6.pla	16	19710	17	14.6	93	0.2	21	1.2
co14.pla	14	14	118	~0	77	~0	68	0.5
life.pla	9	140	216	0.05	238	0.6	79	2.7
sym10.pla	10	837	391	0.09	545	17.4	205	143.2

*This testcase is from LGSynth91 benchmark.

Table 3.1: Comparison for single output functions run on the same PC

3.5 Summary

In this chapter, don't cares of functional decomposition are utilized to simplify the Boolean functions based on the concepts of multilevel cube. An important and common property of containment of cofactors is exploited so as to produce useful DCs to minimize the function without the help of SDCs or ODCs. The experimental results verify the correctness and effectiveness of our method. The work is being generalized to multiple output functions which will be presented in the next chapter.

Chapter 4

Multilevel Minimization for Multiple Output Logic Functions

4.1 Introduction

Multilevel logic minimization plays a very important role in achieving high quality digital circuits. The key point of multilevel logic simplification is to extract don't cares (DCs) from a given Boolean network[88]. There are two kinds of internal DCs, known as satisfiability don't cares (SDCs) and observability don't cares (ODCs) which are based on the topology or structure of a Boolean network[17]. Usually these DCs are very large and consequently only a small subset of them can be used efficiently as in the transduction[105, 106] and global flow methods[21]. A new kind of DCs, namely functional DCs, have been proposed in chapter 3[152]. These are based on functional or logic information, instead of network or circuit connections. It is proved to be very effective for multilevel minimization. An algorithm for single output functions has been developed in chapter 3, which can produce better results than script.rugged of SIS[134] for some testcases even with the most expensive minterm input format. In this chapter, the algorithm proposed in chapter 3 is improved for multilevel logic optimization and then generalized to multiple output functions. The literal count is used as the cost criterion.

4.2 Review of Functional Don't Cares

The concept of functional DCs is based on the property of containment, which has been defined in definition 3.3. For convenience, the definition is reproduced below.

Definition 4.1. For an incompletely specified Boolean function $f(x_{n-1}, x_{n-2}, \cdots, x_0)$, there are two cofactors $f_{\dot{x}_i}$ and $f_{\bar{\dot{x}}_i}$ with respect to a literal $\dot{x}_i$, $\dot{x}_i \in \{x_i, \bar{x}_i\}$, $0 \leq i \leq n-1$. There is containment upon $\dot{x}_i$ if and only if

$$[(f_{\dot{x}_i})_{DC} \cup (f_{\dot{x}_i})_{ON}] \supset (f_{\bar{\dot{x}}_i})_{ON} \tag{4.1}$$

Alternatively, one can say that $f_{\dot{x}_i}$ contains $f_{\bar{\dot{x}}_i}$. For completely specified function, equation (4.1) can be simplified as in equation (4.2).

$$f_{\dot{x}_i} \supset f_{\bar{\dot{x}}_i} \tag{4.2}$$

As in previous equations, f_{DC} and f_{ON} are the dc-set and on-set of the function f; "$\cup$" and "$\supset$" are set operations of "union" and "proper inclusion" respectively.

The concept of containment is similar to but different from unateness due to the existence of DCs. If f is unate upon variable x_i, then there is containment upon literal $\dot{x}_i$. However, if there is containment upon literal $\dot{x}_i$, then the unateness is not necessarily guaranteed upon variable x_i. For instance, it is possible to have containment upon both x_i and $\bar{x}_i$ based on equation (4.1)[152] but impossible to be both monotone decreasing and monotone increasing upon variable x_i[26]. This fact is due to the existence of DCs. Hence containment is a more general concept than unateness, which leads to the generation of the functional DCs.

Theorem 4.1. *Given a single output incompletely specified Boolean function $f(x_{n-1}, x_{n-2}, \cdots, x_0)$, if there is no containment upon literal $\dot{x}_i$, $0 \leq i \leq n-1$, then there are at least two literals in any of the multilevel expression for the function.*

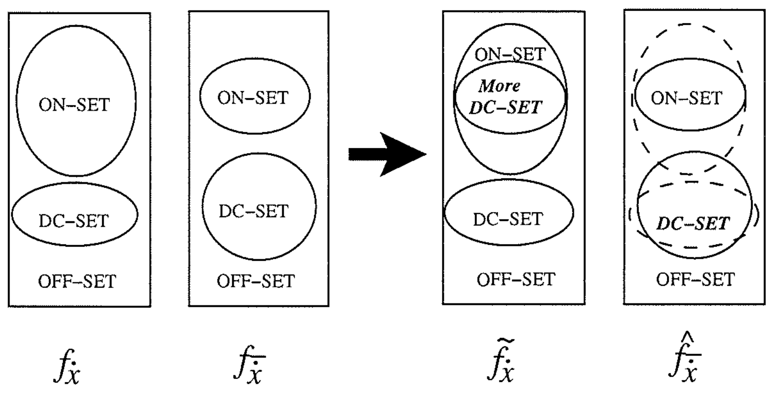
Proof. Suppose there is only one literal $\dot{x}_i$, $\dot{x}_i \in \{x_i, \bar{x}_i\}$, in an expression of f. Factor out all the multilevel terms in order to obtain a two-level sum-of-products (SOP) expression F. It is straightforward that either x_i or $\bar{x}_i$ but not both appear throughout the expression. Hence cofactor $f_{\dot{x}_i}$ covers more cubes than cofactor $f_{\bar{\dot{x}}_i}$. Consequently $f_{\dot{x}_i}$ contains $f_{\bar{\dot{x}}_i}$ which conflicts with the condition that there is no containment upon literal $\dot{x}_i$ for function f. □

If there is containment upon $\dot{x}_i$, then functional DCs are generated and the function can be decomposed by either of the following equations.

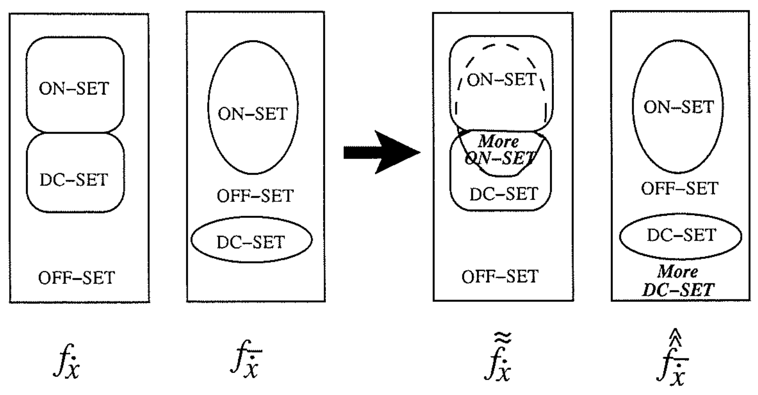
$$f = \dot{x}_i \tilde{f}_{\dot{x}_i} + \hat{f}_{\bar{\dot{x}}_i} \tag{4.3}$$

$$f = (\dot{x}_i + \hat{\hat{f}}_{\bar{\dot{x}}_i})\tilde{\tilde{f}}_{\dot{x}_i} \tag{4.4}$$

In equation (4.3), $\tilde{f}_{\dot{x}_i}$ is the same as $f_{\dot{x}_i}$ except that all the cubes covered by $(f_{\bar{\dot{x}}_i})_{ON}$ can be used as DCs; $\hat{f}_{\bar{\dot{x}}_i}$ is the same as $f_{\bar{\dot{x}}_i}$ except that $(f_{\bar{\dot{x}}_i})_{DC}$ is changed to $(f_{\bar{\dot{x}}_i})_{ON} \cap [(f_{\dot{x}_i})_{ON} \cup (f_{\dot{x}_i})_{DC}]$. Similarly, in equation (4.4), $\hat{\hat{f}}_{\bar{\dot{x}}_i}$ is the same as $f_{\bar{\dot{x}}_i}$ except that all the cubes which are not covered by $(f_{\dot{x}_i})_{ON} \cup (f_{\dot{x}_i})_{DC}$ can be used as DCs; $\tilde{\tilde{f}}_{\dot{x}_i}$ is the same as $f_{\dot{x}_i}$ except that all the cubes covered by $(f_{\dot{x}_i})_{DC} \cap (f_{\bar{\dot{x}}_i})_{ON}$ become new on-set. Both equations (4.3) and (4.4) have only one literal for variable x_i which is consistent with theorem 4.1. The generation process of functional DCs is illustrated by Venn diagrams in fig.4.1 and example 4.1. The detail proof can be found in chapter 3 or [152].



(a) Venn diagrams for equation (4.3)



(b) Venn diagrams for equation (4.4)

Figure 4.1: Explanation of the functional DCs

Example 4.1. A Boolean function $f(x_3x_2x_1x_0) = \Sigma(0,1,2,5,8,9,10)$ is shown in fig.4.2 by a Karnaugh map. It can be seen that $f_{\dot{x}_2} = \{0,1,2,4,5,6\}$ contains $f_{\bar{\dot{x}}_2} = \{1\}$ where $\dot{x}_2 = \bar{x}_2$ according to definition 4.1. After selection of the multilevel cube $\bar{x}_2$ on the Karnaugh map[152], the original function can be decomposed into two subfunctions. Without loss the generality, equation (4.4) is selected to split as shown in fig.4.2. As a result, two minterms $\{3, 7\}$ which are not covered by $(f_{\dot{x}_2})_{ON} \cup (f_{\dot{x}_2})_{DC}$ are functional DCs for $\tilde{\tilde{f}}_{\dot{x}_2}$. From equation (4.4) we have,

$$\begin{aligned} f &= (\bar{x}_2 + \hat{\hat{f}}_{\bar{x}_2})\tilde{\tilde{f}}_{\dot{x}_2} && (4.5)\\ &= (\bar{x}_2 + \bar{x}_3 x_0)\overline{x_1 x_0} && (4.6)\\ &= (\bar{x}_2 + \bar{x}_3 x_0)(\bar{x}_1 + \bar{x}_0) && (4.7) \end{aligned}$$

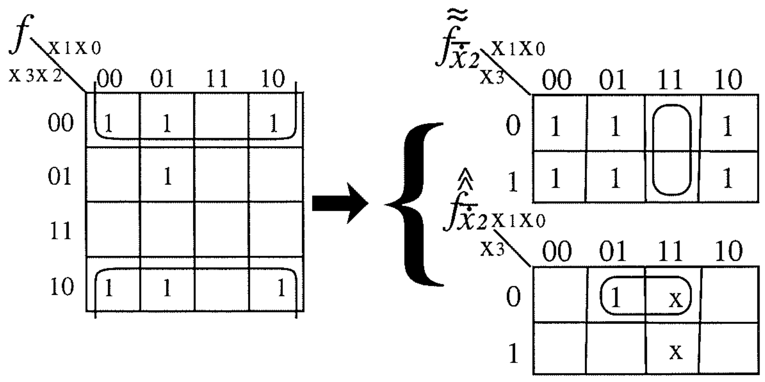


Figure 4.2: Example of functional DCs

In this chapter, the following improvements will be added over chapter 3 on the application of functional DCs.

1. Find a better solution to select a literal when there are more than one literals with containment. This is similar to the variable order problem of Binary Decision Diagrams (BDDs).
2. One more splitting equation is introduced to further utilize the property of containment.
3. Generalize the idea for multiple output Boolean functions.

4.3 Simplification for single output functions

4.3.1 A better solution to the variable order problem

Variable order is a very common problem in logic synthesis. In ESPRESSO[26], a two-level logic minimizer based on the property of unate functions, the most binate variable is selected to decompose a function so that the subsequent subfunctions will be unate after the minimum number of splittings. The most binate variable is the variable having the most number of *"0"* and *"1"* in all the cubes. The same method is used for BDDs in [64] for multilevel methods as the initial representations since both of them are based on AND/OR operations. Although this is very simple heuristic, it is proven to be very powerful judging by the experimental results in [64]. For functional decision diagrams (FDDs) based on AND/XOR operations, the most unate variable is selected in [54] to solve the variable order problem. Ideally the most "containable" literal should be selected to apply functional DCs which is based on containment. However it is computationally extensive. Therefore, the most unate literal is selected instead. Based on our experiments for incompletely specified functions, which consist of both on-sets and DCs, the most unate literal decided by on-set cubes only usually leads to better results compared to a decision based on both on-set and dc-set cubes which is used in [152].

4.3.2 One more splitting equation

Although equations (4.3) and (4.4) are complete to express functions that have containment, they are not sufficient to lead to the minimal results even with the generation of functional DCs as shown in example 4.2.

Example 4.2. A 5-variable single output Boolean function f can be expressed by two equations,

$$\begin{cases} f = x_0 g + x_1 \bar{g} \\ g = \bar{x}_2 + x_3 x_4 \end{cases}$$

The total literal number of f is 7. If equations (4.3) and (4.4) are used to decompose the function with functional DCs[152], then another expression is obtained for f, $h = x_0(\bar{x}_2 + x_3 x_4) + x_1 x_2(\bar{x}_3 + \bar{x}_4)$ with 8 literals where the common subfunction $\bar{x}_2 + x_3 x_4$ cannot be extracted and consequently shared.

Theorem 4.2. *If two cofactors $f_{\dot{x}_i}$ and $f_{\bar{\dot{x}}_i}$ of a single output n-variable Boolean function f are $\dot{x}_j + g$ and $\dot{x}_j \bar{g}$ respectively, $\dot{x}_{i,j} \in \{x_{i,j}, \bar{x}_{i,j}\}$, then there are containments upon both $\dot{x}_i$ and $\dot{x}_j$, where $0 \le i, j \le n-1$, $i \ne j$, g is a n-2 variable function independent of x_i and x_j. Furthermore, f can be decomposed as in equation (4.8).*

$$f = \dot{x}_i g + \dot{x}_j \bar{g} \tag{4.8}$$

Proof. From Shannon expansion, we have,

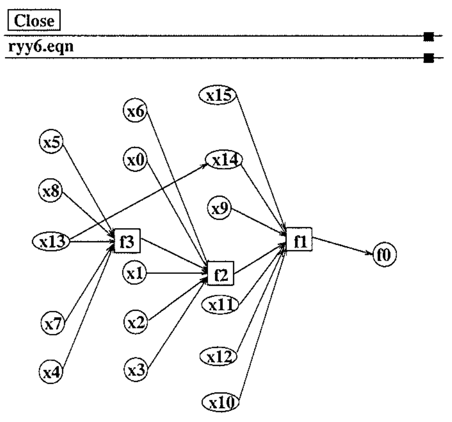
$$\begin{aligned} f &= \dot{x}_i f_{\dot{x}_i} + \bar{\dot{x}}_i f_{\bar{\dot{x}}_i} && (4.9) \\ &= \dot{x}_i \dot{x}_j \bar{g} + \bar{\dot{x}}_i (\dot{x}_j + g) && (4.10) \\ &= \dot{x}_i g + \dot{x}_j \bar{g} && (4.11) \end{aligned}$$

Hence equation (4.8) holds. Based on definition 4.1, it can easily be seen that there is containment upon $\dot{x}_i$. From equation (4.11), $f_{\dot{x}_j} = \bar{g} + \dot{x}_j g$ and $f_{\bar{\dot{x}}_j} = \dot{x}_j g$. Therefore there are containments upon both $\dot{x}_i$ and $\dot{x}_j$. $\square$

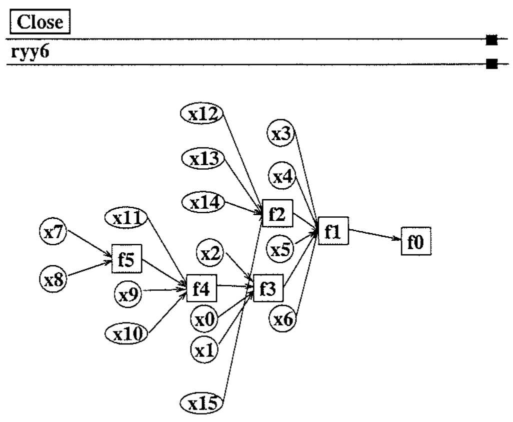
From theorem 4.2, if there are containments upon two literals $\dot{x}_i$ and $\dot{x}_j$, then check whether the following two conditions can be satisfied:

1. One cofactor $f_{\dot{x}_i}$ covers the *n-1* dimensional cube $\dot{x}_j$ while $f_{\bar{\dot{x}}_i}$ is covered by the same cube $\dot{x}_j$.
2. The function composed of the cubes covered by $f_{\dot{x}_i} - \dot{x}_j$ and the function composed of the cubes covered by $\dot{x}_j - f_{\bar{\dot{x}}_i}$ can be each other's complement, where "-" is the set operation of "difference".

If the conditions are satisfied, then decompose the function by equation (4.8) to reduce the literal numbers. Fig.4.3 shows a simple example for a single output function *ryy6* with 16 inputs. The output file of our program is in equation format which can be read by SIS. Command *plot_ blif* is used to obtain the diagrams in fig.4.3. The algorithm based on functional DCs produces better result than "script.rugged" both in area and speed.



(a) functional DCs (17 literals)



(b) script.rugged (21 literals)

Figure 4.3: Results of *ryy6* with 16 inputs

4.4 Multilevel minimization for multiple output functions

4.4.1 Multiple output functions and Boolean relations

It is emphasized in [28] that multiple output functions are fundamentally different from single output functions with respect to don't care conditions. For multiple output functions, the concept of DCs should be generalized to the concept of *vertex equivalence classes* in order to formulate the functions as Boolean relations. In this way, a Boolean relation can be considered as a specification while multiple Boolean functions are only one of the various implementations. One main disadvantage of Boolean relations is that they are very computation expensive to simplify. However, if we use $\lceil \log_2^m \rceil$ extra binary variables to merge m output functions into a single output function, then the gap between single output functions and multiple output functions is reduced and thus it provides a new way to solve the problem of Boolean relations. This idea can be illustrated by example 6 of [28].

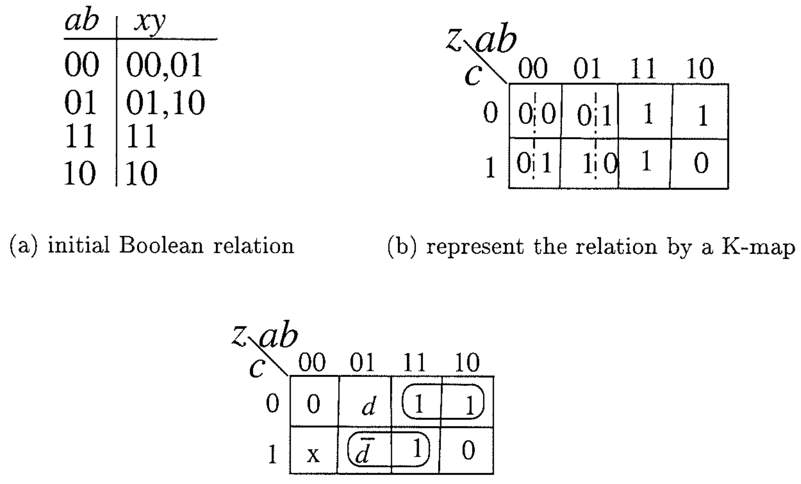
Example 4.3. Given a Boolean relation $\mathbb{R} \subseteq \mathrm{B}^2 \times \mathrm{B}^2$ with two inputs, a and b; two outputs, x and y as shown in fig.4.4(a). When input vector $(ab) = (00)$, the output vector (xy) can be either (00) or (01). Similarly, when input vector $(ab) = (01)$, the output vector (xy) can be either (01) or (10). For either of the individual function, there is no DC conditions. Thus the useful information cannot be represented as DCs for individual function but use *vertex equivalence classes* instead. In [28], four Karnaugh maps are used with two outputs on each map to calculate the best solution, $x = a$; $y = b$. If we add an extra variable c so that two output functions can be merged into one 3-variable function z; $x = z|_{c=0}$ and $y = z|_{c=1}$, then fig.4.4(b) can be obtained. In fig.4.4(b), when input vector $(abc) = (000)$, z is a constant 0. As a result z can be replaced by "x" (DC) for input vector $(abc) = (001)$. When input vector $(ab) = (01)$, one more extra variable d can be used to represent the relation between the outputs x and y. Therefore, a variable-entered map[2, 71] is shown in fig.4.4(c). From that variable-entered map, d is set to 0 and the result of z is $z = \bar{c}a + cb$. Finally splitting z with respect to variable c leads to the same result as in [28], $x = z|_{c=0} = a$ and $y = z|_{c=1} = b$.

The general algorithm to solve Boolean relations will not be further discussed here. We will use the same strategy to implement multilevel logic minimization for multiple output functions. This method has also been used in the minimization of BDDs for multiple output functions where the extra variables are called *output selection variables*[81].

4.4.2 Encoding problem for multiple output functions

In the previous section, multiple output functions are merged into one single output function by adding extra variables. Then that function is simplified and split to produce the results for individual functions. From the view of multilevel network, all the nodes that are independent of extra variables are shared by all the individual functions. That will greatly reduce the literal count.

Example 4.4. Given two Boolean functions, $f_0(x_3, x_2, x_1, x_0) = \{0, 1, 2, 8, 9, 10, 12, 13, 14\}$ and $f_1(x_3, x_2, x_1, x_0) = \{0, 8\}$. One extra variable x_4 is required to merge them into a 5-variable function f. Let $f_0 = f|_{x_4=0}$ and $f_1 = f|_{x_4=1}$. Thus $f(x_4, x_3, x_2, x_1, x_0) =$



(a) initial Boolean relation (b) represent the relation by a K-map

(c) minimization of the relation

Figure 4.4: Simplification of Boolean relation

$\{0, 1, 2, 8, 9, 10, 12, 13, 14, 16, 20\}$. Then call the program for single output function using functional DCs, whose result is shown in example 3.4 in chapter 3, $f = (\bar{x}_4 + \bar{x}_0\bar{x}_3\bar{x}_1)(\bar{x}_0 + \bar{x}_3)(\bar{x}_2 + \bar{x}_1)$. Therefore, $f_0 = f|_{x_4=0} = (\bar{x}_0 + \bar{x}_3)(\bar{x}_2 + \bar{x}_1)$ and $f_1 = f|_{x_4=1} = \bar{x}_0\bar{x}_3\bar{x}_1(\bar{x}_0 + \bar{x}_3)(\bar{x}_2 + \bar{x}_1) = \bar{x}_0\bar{x}_3\bar{x}_1 f_0$. It can be seen that f_0 and f_1 share a common node $(\bar{x}_0 + \bar{x}_3)(\bar{x}_2 + \bar{x}_1)$ since it is independent of variable x_4.

For more than two outputs, an obvious problem is about how to encode these outputs. The cost varies with different encoding schemes. For example, testcase *table5* has 17 input variables and 15 outputs. Hence we need $\lceil \log_2^{15} \rceil = 4$ binary variables to encode these outputs and there are totally $16! \doteq 2.1 \times 10^{13}$ different encodings. In table 4.1, we select 16 encodings among them where all the encoding numbers are in increasing order. For example, in the No.0 encoding scheme of table 4.1, (0001) is assigned to the first subfunction, (0010) is assigned to the second subfunction and so on. Finally (1111) is assigned to the last subfunction. Besides, the unused code (0000) can be considered as DCs. In table 4.1, lit#1 and lit#2 are the literal numbers with and without the extra variables respectively, the unit for time is second. It can be seen that the literal number excluding extra variables is 1064 with encoding number 12, which is less than the result of script.rugged of SIS[134], whose literal number is 1074 and the CPU time is 27.7 seconds respectively run on the same computer.

Although encoding is a traditional problem for functional decomposition[104] and other fields of logic synthesis[147], there are several differences.

1. Encoding problem here is based on containment. In other words, the function after the encoding should have as many containments as possible.

No.	lit#1/lit#2/time*(s)*	No.	lit#1/lit#2/time*(s)*
0	2021/1650/4.41	8	2212/1711/3.95
1	2021/1650/4.40	9	2106/1584/4.36
2	2064/1661/12.43	10	2157/1663/3.85
3	2054/1630/12.64	11	2120/1587/5.52
4	2764/2453/7.50	12	1772/1064/7.73
5	2772/2454/7.40	13	1824/1094/7.63
6	2334/1968/4.04	14	2797/2180/5.30
7	2041/1561/4.00	15	1800/1092/7.75

Table 4.1: Literal numbers with different encodings for *table5* (*17 inputs and 15 outputs*)

2. The extra variables will not appear in the final implementation after splitting. So the literal number of extra variables does not matter.

4.5 Experimental results

Based on the C program developed in chapter 3, we incorporate the ideas from section 4.3 about the variable order and splitting expansions and apply it for multiple output functions. An input PLA format[26] is first converted to minterm format of multiple m output functions, $m \geq 1$. Then merge them into a single output function using $\lceil \log_2^m \rceil$ extra variables and call the improved program to simplify the single output function. The reason for using minterm format is just to know the effectiveness of functional DCs without any other techniques. For the practical point, the idea of functional DCs should be incorporated to other minimizer such as SIS to further improve the results.

	i/o	*Fun. DCs*	*script.algebraic*	*script.rugged*
		lit#/time	*lit#/time*	*lit#/time*
alu4	14/8	2501/0.21	3530/68.1	-
b12	15/9	80/1.51	629/4.7	141/8.1
cu	14/11	66/0.26	66/0.2	58/0.3
decode	5/16	37/0.01	52/0.2	52/0.4
misex1	8/7	47/0.01	66/0.2	60/0.4
misex3	14/14	2686/0.57	4379/97.1	-
pm1	16/13	53/2.71	52/0.4	50/0.4
rd84	8/4	409/0.02	756/4.4	348/21.8
sao2	10/4	180/0.01	248/1.1	188/7.6
sqrt8ml	8/4	70/0.01	148/0.5	94/0.8
table5	17/15	1092/7.75	1236/8.1	1074/27.7
x2	10/7	45/0.06	54/0.2	48/0.3

Table 4.2: Results for multiple output functions

The developed program is tested using MCNC and IWLS'93 benchmarks on a PC with PII-266 CPU and 64M RAM under Linux operating system. Due to lack of efficient encoding method for the problem proposed in section 4.4.2, a simple encoding scheme is applied, that is to encode a binary code of i to ith subfunction. For example, *rd84* has 4 outputs, so (00), (01), (10), (11) are assigned to the first, second, third and fourth subfunctions respectively. For the codes that have not been used, they are taken as external DCs. In table 4.2, all the time is in seconds and "-" means not available due to memory and time limitation. The results of "script.algebraic" and "script.rugged" are also shown with BLIF input format[134]. Although there is no efficient method for encoding, our program can still produce very good results even with the minterm input format.

4.6 Summary

The concept of containment is more general than unateness[26] and leads to the generation of functional DCs. The algorithm of multilevel logic simplification based on functional DCs is first reviewed and improved in the aspects of variable order and splitting equations for single output Boolean functions. The idea is then generalized to multiple output function through encoding method. This encoding strategy provides a new approach to simplify Boolean relations which are known to be computationally extensive[28]. The improved algorithm has been implemented in C language and tested using common benchmarks. Experimental results show that functional DCs are also very effective for multiple output function. This work can be further developed as follows.

1. Find an efficient method to solve the encoding problem described in section 4.4.2, which is different from the traditional one for the functional decomposition[104].
2. Incorporate the idea in other logic minimizers such as SIS to improve the performance.

Chapter 5

Polarity Conversion for Single Output Boolean Functions

5.1 Introduction

In the logic synthesis process, AND/XOR (Reed-Muller) design has shown several advantages, such as high testability, low cost for arithmetic and symmetric functions. Therefore, the conversion between the standard SOP and Reed-Muller forms becomes necessary and there has been extensive research on it.

In [155], any function in SOP form can be converted by three algorithmic rules of b_j coefficient maps, similar to Karnaugh maps. Then the best polarity that corresponds with the maximum number of zero-valued b_j coefficients can be determined by folding technique. Tabular technique is applied in [4] and [6] for conversion between the canonical SOP and Reed-Muller forms. The computer experiments show that it will take much CPU time for large Boolean functions even with the parallel process realization [138]. Recently, the relationship between on-set coefficients of SOP forms and the corresponding Reed-Muller coefficients is published in [85]. The algorithm in [85] requires less computer memory since it computes from only on-set coefficients, but it takes excessive CPU time when $n \geq 15$. On the other hand, a hardware realisation for conversion with any fixed polarity is proposed in [3]. Although the conversion speed is very fast, it is only suitable for small functions because of the limitation of the number of chip pins. Other relevant results can also be found in [5] and [97].

5.2 Basic definitions and terminology

Any n-variable Boolean function can be expressed canonically by the SOP form in equation (5.1).

$$f(x_{n-1}x_{n-2}\cdots x_0) = \sum_{i=0}^{2^n-1} a_i m_i \tag{5.1}$$

where the subscript i can also be expressed in a binary form as $i = (i_{n-1}i_{n-2}\cdots i_0)$, "$\sum$" is the OR operator, the minterm m_i can be expressed as $m_i = \dot{x}_{n-1}\dot{x}_{n-2}\cdots\dot{x}_0$,

$$\dot{x}_j = \begin{cases} \bar{x}_j,\ i_j = 0 \\ x_j,\ i_j = 1 \end{cases} \tag{5.2}$$

Or it can be expressed by the exclusive sum-of-products, loosely known as the positive polarity Reed-Muller (PPRM) form as follows.

$$f(x_{n-1}x_{n-2}\cdots x_0) = \bigoplus_{i=0}^{2^n-1} b_i p_i \tag{5.3}$$

where "$\bigoplus$" is the XOR operator, $p_i = \dot{x}_{n-1}\dot{x}_{n-2}\cdots\dot{x}_0$,

$$\dot{x}_j = \begin{cases} 1,\ i_j = 0 \\ x_j,\ i_j = 1 \end{cases} \tag{5.4}$$

In equations (5.2) and (5.4), j is from 0 to $n-1$. We will refer to the coefficients of SOP form and the coefficients of Reed-Muller form as a and b respectively for simplicity.

For example, when n is 2, $f(x_1x_0)$ can be expanded by the SOP form as follows.

$$\begin{aligned} f(x_1x_0) &= a_{00}\bar{x}_1\bar{x}_0 + a_{01}\bar{x}_1x_0 + a_{10}x_1\bar{x}_0 + a_{11}x_1x_0 \\ &= a_0\bar{x}_1\bar{x}_0 + a_1\bar{x}_1x_0 + a_2x_1\bar{x}_0 + a_3x_1x_0 \\ &= \sum_{i=0}^{3} a_i m_i \end{aligned}$$

Alternatively, it can be expanded by the positive polarity Reed-Muller form as follows.

$$\begin{aligned} f(x_1x_0) &= b_{00} \oplus b_{01}x_0 \oplus b_{10}x_1 \oplus b_{11}x_1x_0 \\ &= b_0 \oplus b_1x_0 \oplus b_2x_1 \oplus b_3x_1x_0 \\ &= \bigoplus_{i=0}^{3} b_i p_i \end{aligned}$$

In a fixed polarity Reed-Muller (FPRM) expansion with any fixed polarity p, $p = (p_{n-1}p_{n-2}\cdots p_0)$, every variable can only be either true or complemented, but not both. If an entry of p, p_j is 0 (or 1) then the corresponding variable is in the true (or complemented) form. Therefore, there are 2^n polarities for a n-variable function, and the positive polarity is equivalent to zero polarity.

5.3 Conversion of the coefficients with zero polarity

Since the minterms are mutually exclusive, equation (5.1) can be rewritten as follows.

$$f(x_{n-1}x_{n-2}\cdots x_0) = \sum_{i=0}^{2^n-1} a_i m_i \tag{5.5}$$

Because $\overline{x} = 1 \oplus x$, replace $\overline{x}$ with $1 \oplus x$ in each m_i of equation (5.5) and simplify to convert from a to b. For any Reed-Muller coefficient, b_i, where $i = (i_{n-1}i_{n-2}\cdots i_0)$, if i_j is 1, then there is x_j in p_i according to equation (5.4). Because x_j can be created by both x_j and $\bar{x}_j$ in m_i, i_j can be both 1 and 0 in a according to equation (5.2). Otherwise, if i_j is 0, then there is a constant "1" instead of x_j in p_i according to equation (5.4). Because "1" can only be created by $\bar{x}_j$ in m_i, i_j can only be 0 in a according to equation (5.2). Alternatively, the above transformation can be written as equation (5.6).

$$b_i = b_{i_{n-1}i_{n-2}\cdots i_0} = \sum_k a_{k_{n-1}k_{n-2}\cdots k_0} \tag{5.6}$$

where $k = (k_{n-1}k_{n-2}\cdots k_0)$,

$$k_j = \begin{cases} \times, & i_j = 1 \\ 0, & i_j = 0 \end{cases} \tag{5.7}$$

In equation (5.7) "$\times$" is the notation for both 0 and 1, $j \in \{0, 1, \cdots, n-1\}$. For example, if n is two, then according to equation (5.6) we have,

$$\begin{aligned} b_2 &= b_{10} \\ &= \sum_k a_{\times 0} \\ &= a_{00} \oplus a_{10} \\ &= a_0 \oplus a_2 \end{aligned}$$

In the same way, we have,

$$\begin{aligned} b_3 &= b_{11} \\ &= \sum_k a_{\times\times} \\ &= a_{00} \oplus a_{01} \oplus a_{10} \oplus a_{11} \\ &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \end{aligned}$$

Similarly, because $1 = x \oplus \overline{x}$, replace "1" with $x \oplus \overline{x}$ in each p_i of equation (5.3) and simplify to convert from b to a. Equations (5.8) and (5.9) can be obtained in the same way.

$$a_i = a_{i_{n-1}i_{n-2}\cdots i_0} = \sum_k b_{k_{n-1}k_{n-2}\cdots k_0} \tag{5.8}$$

where $k = (k_{n-1}k_{n-2}\cdots k_0)$,

$$k_j = \begin{cases} \times, & i_j = 1 \\ 0, & i_j = 0 \end{cases} \tag{5.9}$$

Comparing equations (5.6), (5.7) with (5.8), (5.9), it can be seen that the only difference between them is the entries of a and b. The conversion methods from a to b and from b to a are identical with zero polarity. This leads to observation 5.1.

Observation 5.1. *Any Reed-Muller coefficient b_i can be computed from SOP coefficients a_k according to equations (5.6) and (5.7) with zero polarity. Similarly, any SOP coefficient a_i can be computed from Reed-Muller coefficients b_k according to equations (5.8) and (5.9) with zero polarity.*

Alternatively, equations (5.7) and (5.9) can be expressed using a truth table, table 5.1, to show the bitwise relationship between the subscripts of a and b, where i_j or k_j is any bit of $i_0, i_1, \cdots, i_{n-1}$ or $k_0, k_1, \cdots, k_{n-1}$. Here we define a criterion function $g = (g_{n-1}g_{n-2}\cdots g_0)$, where g_j is shown in table 5.1. If the value of the function $g = (g_{n-1}g_{n-2}\cdots g_0)$ is 1, that is, all of its binary bits, $g_{n-1}, g_{n-2}, \cdots, g_0$ are 1, then equations (5.7) and (5.9) are satisfied. The coefficient can be computed using equation (5.6) or (5.8) with all the coefficients whose subscript satisfies equation (5.7) or (5.9). It can be seen from equations (5.7) and (5.9) that the number of satisfied coefficients, both on-set and off-set, is 2^t, where t is the number of 1-valued bits of i. For example, to compute a Reed-Muller coefficient b_i from SOP coefficients, if all the binary bits of g are 1 for the subscripts i and k, then the SOP coefficient a_k should be included in equation (5.6); otherwise, it should not be included. This process is shown in example 5.1.

$i_j \backslash k_j$	0	1
0	1	0
1	1	1

Table 5.1: Truth table of the criterion function g_j

Furthermore, table 5.1 can be expressed by a Boolean function (5.10) or (5.11), where " | " and "&" are the bitwise OR and AND operators respectively. Using either equation (5.10) or (5.11), it is possible to decide if a particular coefficient should be included for conversion. Besides, only the on-set coefficients need to be calculated since $0 \oplus x = x$. If the number of the included on-set coefficients is odd, then the converted coefficient is 1; otherwise, it is 0. The above methods are exemplified in example 5.1.

$$g_j = \bar{k}_j \,|\, i_j \tag{5.10}$$

$$\bar{g}_j = k_j \,\&\, \bar{i}_j \tag{5.11}$$

Example 5.1. Compute Reed-Muller coefficient b_6 with zero polarity for a 4-variable function $f(x_3x_2x_1x_0) = \sum(1, 2, 5, 6, 7, 8, 10, 11)$.

According to equations (5.6) and (5.7), b_6 =b_{0110}, which means $i_2 = i_1 = 1$. So k_2 and k_1 can be both 0 and 1, that is,

$$\begin{aligned} b_{0110} &= \sum_k a_{0\times\times 0} \\ &= a_{0000} \oplus a_{0010} \oplus a_{0100} \oplus a_{0110} \\ &= a_0 \oplus a_2 \oplus a_4 \oplus a_6 \\ &= 0 \oplus 1 \oplus 0 \oplus 1 \\ &= 0 \end{aligned}$$

Alternatively, b_6 can be calculated by equation (5.10). For the first on-set coefficient "1", we have

$$
\begin{aligned}
g &= \bar{1}\,|\,6 \\
&= \bar{0}\bar{0}\bar{0}\bar{1}\,|\,0110 \\
&= 1110\,|\,0110 \\
&= 1110
\end{aligned}
$$

Because g_0 is 0 instead of 1, this coefficient should not be included in the computation of b_6. Then switch to the second on-set coefficient "2". Similarly we have

$$
\begin{aligned}
g &= \bar{2}\,|\,6 \\
&= \bar{0}\bar{0}\bar{1}\bar{0}\,|\,0110 \\
&= 1101\,|\,0110 \\
&= 1111
\end{aligned}
$$

Because all of g_j are 1, this coefficient should be included. Then switch to the third on-set coefficient and so on. After finishing all 8 on-set coefficients, only two coefficients "2" and "6" are included. Hence b_6 is 0 since two is even. All these procedures can be done easily by a computer.

Now observation 5.1 is updated by the following observation.

Observation 5.2. *Any zero polarity Reed-Muller coefficient b_i can be computed from the on-set SOP coefficients using equation (5.10) or (5.11). If all g_j are 1 then this coefficient is included. If the number of included coefficients is odd, then b_i is 1; otherwise, b_i is 0. In the same way, any SOP coefficient a_i can be computed from the on-set Reed-Muller coefficients with zero polarity.*

5.4 Conversion of the coefficients with a fixed polarity

In the previous section, a zero polarity conversion method is introduced. In this section a polarity for SOP forms is proposed to extend the conversion method from zero polarity to any fixed polarity. Then the bidirectional conversion method between SOP and FPRM forms is presented.

5.4.1 Polarity for SOP expansions of Boolean functions

For any n-variable Boolean function, there are 2^n FPRM expansions and expansions with different polarities have different on-set coefficient sets. Here we define a polarity for SOP expansions of Boolean functions.

Definition 5.1. Any Boolean function $f(x_{n-1}x_{n-2}\cdots x_0)$ can be expressed canonically as in equation (5.1). That expansion is defined as the zero polarity. Any variable $\ddot{x}_j$, $j \in \{0,1,\cdots,n-1\}$ in every minterm with a polarity $p = (p_{n-1}p_{n-2}\cdots p_0)$ of the same SOP function $f(\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots\ddot{x}_0)$ is defined as in equation (5.12).

$$\ddot{x}_j = \begin{cases} \bar{x}_j, & if\ p_j = 1 \\ x_j, & if\ p_j = 0 \end{cases} \tag{5.12}$$

According to equation (5.12), if any entry of p, p_j is 0 (or 1), then the corresponding variable is in the true (or complemented) form. This is the same as the polarity for FPRM forms. Now equation (5.1) is extended accordingly as follows.

$$f(\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots\ddot{x}_0) = \sum_{i=0}^{2^n-1} a_i m_i \tag{5.13}$$

where $m_i = \breve{x}_{n-1}\breve{x}_{n-2}\cdots\breve{x}_0$,

$$\breve{x}_j = \begin{cases} \bar{\ddot{x}}_j,\ i_j = 0 \\ \ddot{x}_j,\ i_j = 1 \end{cases} \tag{5.14}$$

In equation (5.14), $\bar{\ddot{x}}$ is the complemented form of $\ddot{x}$. Besides, equation (5.14) is an extension of equation (5.2). Therefore, the corresponding subscript in equation (5.13), $i = (i_{n-1}i_{n-2}\cdots i_0)$ can be obtained from equation (5.14) as follows.

$$i_j = \begin{cases} 0,\ \breve{x}_j = \bar{\ddot{x}}_j \\ 1,\ \breve{x}_j = \ddot{x}_j \end{cases} \tag{5.15}$$

Example 5.2. A 3-variable function $f(x_2x_1x_0) = \sum(1,2,5)$ has zero polarity. Since $\bar{\bar{x}} = x$, we have

$$\begin{aligned} f(x_2x_1x_0) &= \textstyle\sum(1,2,5) \\ &= \textstyle\sum\{(001),(010),(101)\} \\ &= \bar{x}_2\bar{x}_1x_0 + \bar{x}_2x_1\bar{x}_0 + x_2\bar{x}_1x_0 \\ &= \bar{x}_2\bar{x}_1\bar{\bar{x}}_0 + \bar{x}_2x_1\bar{x}_0 + x_2\bar{x}_1\bar{\bar{x}}_0 \end{aligned}$$

If the polarity p is 1, then $\ddot{x}_2 = x_2$, $\ddot{x}_1 = x_1$, and $\ddot{x}_0 = \bar{x}_0$ according to equation (5.12).

Hence, from equations (5.13) and (5.15), this function can also be expressed as $f(\check{x}_2\check{x}_1\check{x}_0)$ with polarity 1 as follows,

$$\begin{aligned} f(\check{x}_2\check{x}_1\check{x}_0) &= f(x_2x_1x_0) \\ &= \bar{x}_2\bar{x}_1\bar{\bar{x}}_0 + \bar{x}_2x_1\bar{x}_0 + x_2\bar{x}_1\bar{\bar{x}}_0 \\ &= \sum\{(000),(011),(100)\} \\ &= \sum(0,3,4) \end{aligned}$$

So this function is $\sum(1,2,5)$ with zero polarity, while it is $\sum(0,3,4)$ with polarity 1. Therefore, any n-variable Boolean function can be expanded canonically as equation (5.13) with polarity p according to definition 5.1. There are 2^n polarities for a Boolean functions. To convert any SOP expansion from polarity p' to polarity p, every subscript i, $0 \leq i < 2^n$ should be converted using equation (5.1),

$$i \Leftarrow i \wedge p \wedge p' \tag{5.16}$$

where "$\wedge$" and "$\Leftarrow$" are bitwise XOR and assignment operators respectively. Since p' is conventionally zero as in equation (5.1), equation (5.16) can be simplified as follows.

$$i \Leftarrow i \wedge p \tag{5.17}$$

A theorem can be formulated about the polarity of SOP expansions.

Theorem 5.1. *If there are M on-set minterms of a n-variable function with polarity p, then there are always M on-set minterms with any other polarities.*

Proof. For any coefficient i of the function with polarity p, it is converted to i' with any other polarity p', $i' = i \wedge p \wedge p'$, according to definition 5.1 and equation (5.16). So there is a one-to-one correspondence between any minterm with polarity p and its corresponding minterm with polarity p'. Thus, the number of on-set minterms is the same with any polarity. □

Unlike the FPRM expansions, the number of on-set coefficients for any SOP expression is fixed for all polarities. The only effect of the polarity is the order of the on-set minterms.

5.4.2 Conversion from a to b with a fixed polarity

Procedure 5.1. *Any Boolean function can be converted from canonical SOP expansion to canonical FPRM expansion with a fixed polarity p through the following steps.*

1. Convert the function to SOP expansion with polarity p;

2. Convert the expansion after step 1 to Reed-Muller expansion using any zero polarity conversion method.

Suppose p and a n-variable Boolean function are expressed by $p = (p_{n-1}p_{n-2}\cdots p_0)$ and $f(x_{n-1}x_{n-2}\cdots x_0)$ respectively. If this function is converted to a canonical Reed-Muller expansion as equation (5.3) with zero polarity then every variable is in the true form. It follows that the polarity of FPRM forms after zero polarity conversion is the same polarity of SOP forms before the conversion. In short, zero polarity conversion does not change the polarity. In other words, equation (5.3) can be extended as follows.

$$f(\hat{x}_{n-1}\hat{x}_{n-2}\cdots\hat{x}_0) = \bigoplus_{i=0}^{2^n-1} b_i p_i \tag{5.18}$$

where $\hat{x}_j$ may be either the true or the complemented form of x_j, and every variable $x_{j'}$ in any term p_i is in the same form as $\hat{x}_{j'}$, $0 \le j, j' \le n-1$. Now, in step 1, this function is first converted to another SOP expansion $f(\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots\ddot{x}_0)$ with polarity p according to equation (5.17). If p_j is 1, $\ddot{x}_j$ is the complemented form of x_j; otherwise, $\ddot{x}_j$ is the true form of x_j. In step 2, $f(\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots\ddot{x}_0)$ is converted to Reed-Muller expansion $\bigoplus_{i=0}^{2^n-1} b_i\ddot{p}_i$ with a zero polarity conversion method. According to equation (5.18), every variable x_j in the term $\ddot{p}_i$ is the same form as in the expansion $f(\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots\ddot{x}_0)$ after step 2. Because of the canonicality of FPRM expansions, $\bigoplus_{i=0}^{2^n-1} b_i\ddot{p}_i$ is the FPRM expansion of the function $f(x_{n-1}x_{n-2}\cdots x_0)$ with polarity p .

Example 5.3. Convert a 4-variable function $f(x_3x_2x_1x_0) = \sum(1,2,5,6,7,8,10,11)$ from the SOP form to the FPRM form with polarity 1.

In step 1, this function is converted to the SOP expansion with polarity 1. From equation (5.17), we have,

$$f(x_3x_2x_1\bar{x}_0) = \sum(0,3,4,7,6,9,11,10)$$

In step 2, $\sum(0,3,4,7,6,9,11,10)$ is converted to Reed-Muller expansion using any zero polarity conversion method. We use the method based on observation 5.2. For example, to compute b_{11} , all the on-set minterms should be decided as in example 5.1. For the first SOP on-set coefficient "0", we have

$$\begin{aligned} g &= \overline{0} \,|\, 11 \\ &= \overline{0000} \,|\, 1011 \\ &= 1111 \,|\, 1011 \\ &= 1111 \end{aligned}$$

This minterm should be included. Similarly, 3, 9, 11, 10 should be included to compute b_{11}. The number of these included on-set coefficients is an odd number 5, so b_{11} is 1. In the same way, the final FPRM expansion can be obtained as $\sum(0,1,2,6,7,8,11,13)$ with polarity 1.

From procedure 5.1, if we have any zero polarity conversion algorithm between SOP and Reed-Muller forms, then it can be extended to any fixed polarity.

5.4.3 Conversion from b to a with a fixed polarity

From equations (5.6), (5.7) and (5.8), (5.9), it can be seen that the zero polarity conversion methods are exactly the same from a to b and from b to a. If a coefficient set C of a SOP form is converted to a Reed-Muller coefficient set with zero polarity, then the result is a set C'. Likewise, if a coefficient set C' of a SOP form is converted to a Reed-Muller coefficient set with zero polarity, then the result is a set C. In example 5.3, the SOP on-set coefficient set $\{0,3,4,7,6,9,11,10\}$ is converted to a Reed-Muller on-set coefficient set $\{0,1,2,6,7,8,11,13\}$. Similarly, the SOP on-set coefficient set $\{0,1,2,6,7,8,11,13\}$ will be converted to the Reed-Muller on-set coefficient set $\{0,3,4,7,6,9,11,10\}$. From this point, procedure 5.1 can be extended to procedure 5.2.

Procedure 5.2. *Any Boolean function can be converted from canonical FPRM expansion with a fixed polarity p to canonical SOP expansion through the following steps.*

$1'$. *Convert the function to SOP expansion using any zero polarity conversion method;*

$2'$. *Convert the SOP expansion after step $1'$ from polarity p to zero polarity using equation (5.17).*

Suppose we have converted the coefficient set C of the SOP form to the FPRM coefficient set C' by the two steps in procedure 5.1. If we want to convert C' to C, then it is an inverse procedure. In step 1 of procedure 5.1, C is first converted to C'' , then C'' is converted to C' in step 2. Now, C'' can be obtained in step $1'$ because of the zero polarity conversion method. In step $2'$, C can be calculated from C'' since $i \wedge p \wedge p = i$ for every coefficient in C''.

The bidirectional conversion is illustrated in fig.5.1 where d is a binary variable for the direction. If d is 0, then the conversion is from a to b; otherwise, it is from b to a. Besides, a or b in fig.5.1 is any element of the set A or B, and "$*$" is for multiplication.

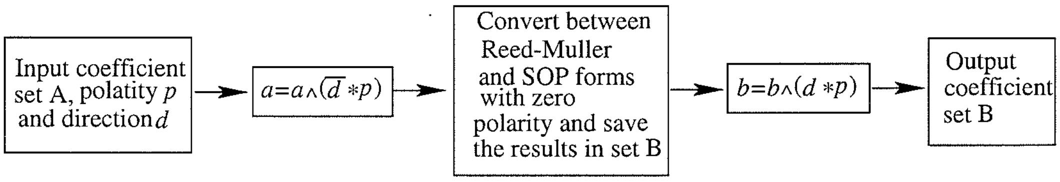

Input coefficient set A, polatity p and direction d
$a=a\wedge(\overline{d}*p)$
Convert between Reed-Muller and SOP forms with zero polarity and save the results in set B
$b=b\wedge(d*p)$
Output coefficient set B

Figure 5.1: Bidirectional conversion between SOP and FPRM forms

5.5 Conversion algorithm for large Boolean functions

In section 5.4, the method for bidirectional conversion is discussed with any fixed polarity. This method has been implemented in C language. For a 15-variable function with 10,000 on-set coefficients, the CPU time for conversion with any fixed polarity is about 13 seconds on a personal computer with Cyrix6x86-166 CPU and 32M RAM. Most of the CPU time, however, is spent on the zero polarity conversion because other parts of the algorithm are simple to compute. In this section, multiple segment and multiple pointer techniques are introduced to improve the speed for zero polarity conversion.

5.5.1 Multiple segment technique

In observation 5.2, to compute a Reed-Muller coefficient, all the SOP on-set coefficients should be accessed once. If we order these on-set coefficients in advance, then the speed will be improved. Thus the multiple segment technique is introduced, defining a segment as a suitable subset of the on-set coefficients.

Definition 5.2. For an n-variable Boolean function $f(x_{n-1}x_{n-2}\cdots x_0)$, any of its on-set coefficients c can be ordered into one of s segments defined below,

$$c \in \textit{the jth segment, iff } j * 2^l \leq c < (j+1) * 2^l \tag{5.19}$$

where $s = 2^{n-l}$, $0 \leq l \leq n$, $0 \leq j < s$, $0 \leq c \leq 2^n - 1$.

Example 5.4. All of the on-set coefficients of a 4-variable function $f(x_3x_2x_1x_0) = \sum(1,2,5,6,7,8,10,11)$ can be ordered into 4 segments, $s0, s1, s2, s3$, that is, $s = 4$, $l = 2$. According to definition 5.2, $s0 = \{1,2\}$, $s1 = \{5,6,7\}$,$s2 = \{8,10,11\}$, $s3 = \emptyset$. Alternatively, these on-set coefficients can be reordered into 8 segments, $s0', s1', s2', s3', s4', s5', s6', s7'$, that is, $s = 8$, $l = 1$. Then, $s0' = \{1\}$, $s1' = \{2\}$, $s2' = \{5\}$, $s3' = \{6,7\}$, $s4' = \{8\}$, $s5' = \{10,11\}$, $s6' = s7' = \emptyset$.

Definition 5.3. For two integers, i, k, which can be represented by binary m-tuples, $i = (i_{m-1}i_{m-2}\cdots i_0)$, $k = (k_{m-1}k_{m-2}\cdots k_0)$, if $i_j \geq k_j$ for all j, $0 \leq j < m$, then i covers k or k is covered by i.

According to definition 5.3, the relation of cover can be expressed exactly as equation (5.10) or (5.11). So any coefficient b_i can be computed from the number of all on-set coefficients a_k, where k is covered by i. If this number is odd, then b_i is 1; otherwise, b_i is 0.

Theorem 5.2. *Suppose there are an integer i and an integer set K, $i = (i_{m-1}i_{m-2}\cdots i_0)$, $k = (k_{m-1}k_{m-2}\cdots k_0)$, where k is any element of K. If $i_j = k_j$ for all j that is from*

l to $m-1$, $0 \leq l < m$, and the number of elements in K covered by i is α, then, after subtracting $i' * 2^l$ from both i and any element of the set K, $i' = (i_{m-1}i_{m-2}\cdots i_l)$, the number of elements in K covered by i is still α.

Proof. For i and any element k in K, since $i_j = k_j$ for all j that is from l to $m-1$, $0 \leq l < m$, all the most significant l bits of k in set K are the same, then i' covers k', $k' = (k_{m-1}k_{m-2}\cdots k_l)$ by definition 5.3. Or the criterion function $g = \overline{k'} \,|\, i'$ is always 1 when k' and i' are the same number by equation (5.10). After subtracting $i' * 2^l$ from both i and any element of K, $i' = (i_{m-1}i_{m-2}\cdots i_l)$, all these $m-l$ bits are set to be zero. Therefore, the criterion function $g = \overline{k'} \,|\, i'$ is still 1 when both k' and i' are zero. So the number of elements that are covered by i in K is the same as before subtracting. □

Example 5.5. Let i, m and l be 109, 7 and 3 respectively, and $K = \{111, 110, 105, 104\}$.

$$\begin{aligned} 109 &= (1101101) \\ 111 &= (1101111) \\ 110 &= (1101110) \\ 105 &= (1101001) \\ 104 &= (1101000) \end{aligned}$$

All of these numbers begin with "1101", that is the number 13. Then subtract $13 * 2^3 = 104$ from these numbers. Now $i = 5$, $K = \{7, 6, 1, 0\}$, the number of elements in K covered by i is the same as before subtracting, that is 2.

Procedure 5.3. *Conversion between a and b with zero polarity can be implemented as follows.*

$1''$. *Any on-set coefficient of an n-variable Boolean function $f(x_{n-1}x_{n-2}\cdots x_0)$ can be ordered into one of s segments by definition 5.2 where $s = 2^{n-l}$, $0 \leq l \leq n$.*

$2''$. *Set $u = 2^l$ and every coefficient in the j'th segment is reduced by $j' * u$, $0 \leq j' < s$.*

$3''$. *For any integer i', $0 \leq i' < u$, count the numbers of on-set coefficients, k', that are covered by i' in each segment. Save these s numbers in an integer array $M[s]$.*

$4''$. *For any integer i'' , $0 \leq i'' < s$, if $\sum_{k''} M[k'']$ is odd, where k'' is covered by i'', $0 \leq k'' < s$, and "$\sum$" stands for the arithmetic addition operator instead of logic operator OR here, then the coefficient $i' + i'' * u$ should be saved as output. Otherwise, if $\sum_{k''} M[k'']$ is even, then the coefficient $i' + i'' * u$ should not be saved as output.*

$5''$. *The saved coefficients in step $4''$ are the result .*

Without losing generality, we suppose that the conversion is from a to b since it is zero polarity conversion.

For any Reed-Muller coefficient i, $0 \leq i < 2^n$, $i = (i_{n-1}i_{n-2}\cdots i_0)$, all its n bits can be divided into two groups, group 1: $i_{n-1}i_{n-2}\cdots i_l$ and group 2: $i_{l-1}i_{l-2}\cdots i_0$. So i can be expressed by $i = i' + i'' * u$, where $i' = i_{l-1}i_{l-2}\cdots i_0$, and $i'' = i_{n-1}i_{n-2}\cdots i_l$. From definition 5.3, the value of b_i is decided by the number of on-set SOP coefficients covered by i. If this number is odd, then b_i is 1; otherwise, b_i is 0. In step $1''$, all the on-set SOP coefficients are ordered into s segments. So the number of on-set SOP coefficients covered by i is the sum of the numbers of coefficients covered by i' of all the segments that are covered by i''. After reducing by $j' * u$ in step $2''$, all the most significant $n - l$ bits are set to zero. Therefore, the number of SOP coefficients covered by the least significant l bits of i is the same as before subtracting according to theorem 5.2. These numbers for summation are obtained in step $3''$ in each segment. Finally the sum is calculated in step $4''$. Therefore, the numbers saved in step $4''$ are the Reed-Muller on-set coefficients.

Example 5.6. Compute Reed-Muller form of the 4-variable function $f(x_3x_2x_1x_0) = \sum(1, 2, 5, 6, 7, 8, 10, 11)$ with zero polarity by procedure 5.3 if s is 4.

In step $1''$, all the SOP coefficients are divided into 4 segments, $s0, s1, s2, s3$. From example 5.4, it can be seen that $s0 = \{1, 2\}$, $s1 = \{5, 6, 7\}$,$s2 = \{8, 10, 11\}$, $s3 = \emptyset$.

In step $2''$, set $u = 4$, and every coefficient in the $j'th$ segment is reduced by $j' * u$, $0 \leq j' < s$. Therefore, $s0 = \{1, 2\}$, $s1 = \{1, 2, 3\}$,$s2 = \{0, 2, 3\}$, $s3 = \emptyset$.

In step $3''$, when i' is 0, the numbers of on-set coefficients covered by i' in each segment are, $M[0] = 0$, $M[1] = 0$, $M[2] = 1$, $M[3] = 0$ because 0 can only cover 0.

In step $4''$:

when i'' is 0, $\sum_{k''} M[k''] = M[0] = 0$, because 0 only covers 0. So i, that is, $i' + i'' * u = 0$, should not be saved.

when i'' is 1, $\sum_{k''} M[k''] = M[0] + M[1] = 0$, because 1 covers both 0 and 1. So i, that is, $i' + i'' * u = 4$, should not be saved.

when i'' is 2, $\sum_{k''} M[k''] = M[0] + M[2] = 1$, because 2 covers both 0 and 2. So i, that is, $i' + i'' * u = 8$, should be saved.

when i'' is 3, $\sum_{k''} M[k''] = M[0] + M[1] + M[2] + M[3] = 1$, because 3 covers 0, 1, 2, 3. So i, that is, $i' + i'' * u = 12$, should be saved.

Then, switch to $i' = 1$ in step $3''$ after all the values of i'', $0 \leq i'' < s$ have been counted in step $4''$. In step $3''$, when i' is 1, the numbers of on-set coefficients covered by i' in each segment are, $M[0] = 1$, $M[1] = 1$, $M[2] = 1$, $M[3] = 0$ because 1 can only cover 0 and 1.

In step $4''$:

when i'' is 0, $\sum_{k''} M[k''] = M[0] = 1$, because 0 only covers 0. So i, that is, $i' + i'' * u = 1$, should be saved.

when i'' is 1, $\sum_{k''} M[k''] = M[0] + M[1] = 2$, because 1 covers both 0 and 1. So i, that is, $i' + i'' * u = 5$, should not be saved.

when i'' is 2, $\sum_{k''} M[k''] = M[0] + M[2] = 0$, because 2 covers both 0 and 2. So i, that is, $i' + i'' * u = 9$, should not be saved.

when i'' is 3, $\sum_{k''} M[k''] = M[0] + M[1] + M[2] + M[3] = 3$, because 3 covers 0, 1, 2, 3. So i, that is, $i' + i'' * u = 13$, should be saved.

Then switch to $i' = 2$ in step $3''$ and repeat the above procedure. Finally, switch to $i' = 3$ in step $3''$ and repeat. These data are shown in table 5.2. According to procedure 5.3, the coefficients saved in step $4''$ are the Reed-Muller coefficients. So the Reed-Muller form is $\circledS(8, 12, 1, 13, 2, 10, 7, 11)$.

i'	0				1				2				3			
$M[0]$	0				1				1				2			
$M[1]$	0				1				1				3			
$M[2]$	1				1				2				3			
$M[3]$	0				0				0				0			
i''	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
$\sum M[k'']$	0	0	1	1	1	2	2	3	1	2	3	4	2	5	5	8
$i' + i'' * u$	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15
saved(Y/N)	N	N	Y	Y	Y	N	N	Y	Y	N	Y	N	N	Y	Y	N

Table 5.2: Example of multiple segment technique

5.5.2 Multiple pointer technique

For large Boolean functions, there are still many coefficients in each segment although they have been divided into s segments by multiple segment technique. Here the multiple pointer technique is introduced to operate on every segment.

In the realization of the multiple segment technique, every coefficient should be accessed by a pointer in a segment to decide the values of $M[s]$ in step $3''$. If there are q pointers to access a segment at the same time, then the speed will be improved. But the number of all coefficients should be the multiples of q, otherwise, some coefficients may be accessed more than once. Some extra coefficients should be appended to any segment so that all the coefficients are accessed exactly once in this segment. The values in $M[s]$ must be the same as before appending of extra coefficients.

In the multiple segment technique of procedure 5.3, the coefficients in the $j'th$ segment are reduced by $j' * u$ in step $2''$ so that the most significant $n - l$ bits of any coefficient are zero. Because zero can only cover zero by definition 5.3, any number whose most significant bit is 1 can be appended to all segments and keep the values in $M[s]$ same. In our experiments, this extra coefficient e is set so that all of its bits are 1.

The conversion algorithm using multiple segment and multiple pointer techniques is given in fig.5.2.

```
$convert\_with\_zero\_polarity(u, s, d, p, q, MMs[\,])$

    /* data in array $MMs[\,]$ are the numbers of coefficients of all
    segments after appending the extra coefficients */

     {
     $for(i = 0; i < u; i++)$
      {
    loop1: $for(j = 0; j < s; j++)$
              $covered\_number[j] \Leftarrow number\_covered\_by\_i(MMs[j])$;
    loop2: $for(j = 0; j < s; j++)$
             {
              $for(k = 0; k < s; k++)$
                {
                 $g \Leftarrow k \& \bar{j}$;
                 $if(g = 1) temp \Leftarrow temp + covered\_number[k]$;
                }
              $if(temp\ is\ odd) save(output\_file, (i + j * u) \wedge (d * p))$;
             }
      }
     }
    /* this routine is based on observation 2 and
                              procedure 5.3 */
```

Figure 5.2: Conversion algorithm using multiple segment technique

5.6 Algorithm and experimental results

From section 5.5, the values of s and q, that are the number of segments and the number of pointers, should be set first. Both s and q can not be too big or too small; otherwise, either an extra burden, in time and memory, is incurred or no significant improvement is achieved. Although the number of pointers q can be set dynamically using an array, we find that the speed will suffer because of the array. In our experiments, we found, by experiment, it is efficient to set q to be a constant 32 in most cases.

As for s, it can be seen from procedure 5.3 that most of the CPU time is spent on steps $3''$ and $4''$. These two steps are labeled as the two loops, loop1 and loop2 in the routine of *convert_with_zero_polarity*( ) in fig.5.2. Suppose there are altogether M on-set coefficients to convert, then these coefficients are divided into s segments. Besides, the number of coefficients of all segments are multiples of q. Then the average number of coefficients in a segment is $\frac{M}{s} + \frac{q}{2}$. In loop1, all the s segments should be accessed once so the CPU time is proportional to $s * (\frac{M}{s} + \frac{q}{2})$. Similarly, the CPU time is proportional to s^2 in the second loop. We determine the value of s by setting these two parts of CPU time to be equal.

$$s * (\frac{M}{s} + \frac{q}{2})\beta = s^2 \tag{5.20}$$

where β is a modifying factor decided by the hardware specifications of the computer and the capability of the compiler. After experimentation, β was to set to 4. From equation (5.20) the value of s can be set to $q + \sqrt{q^2 + 4M}$. Besides, s must be a power of 2 by definition 5.2. So s is set to be the power of 2 that is close to $q + \sqrt{q^2 + 4M}$.

From the above discussion, the conversion algorithm for very large functions between SOP and Reed-Muller forms with any fixed polarity is shown in fig.5.3 using multiple segment and multiple pointer techniques.

It can be seen from the algorithm that the CPU time is made up of three processes as follows.

1. Initialize and read the data from the file to the memory.
2. Preprocess the data with any fixed polarity and convert them with zero polarity according to fig.5.1.
3. Save the outputs to the file.

Further, most CPU time is spent on process 2 for large Boolean functions. Because the conversion process for any fixed polarity is the same, the CPU time in process 2 is independent of both the polarity and the direction. In other words, when n and M are fixed, the combined CPU time of both process 1 and process 2 is identical for any polarity in either direction. Since the number of on-set coefficients after conversion varies with the polarity and the direction, the CPU time of the algorithm is mainly dependent on n and M.

The algorithm is implemented in C language and the program is compiled by the GNU C compiler. Then it is tested on a personal computer with Cyrix6x86-166 CPU and 32M RAM under Linux operating system. Some random coefficient sets are generated to test the effectiveness of the algorithm. The results confirm that the CPU time is mainly dependent on n and M only. For any n-variable Boolean function with $n \leq 12$, the CPU time is almost zero using multiple segment and multiple pointer techniques with any fixed polarity in either direction. These results, both with and without multiple segment and multiple pointer techniques, are shown in table 5.3 and fig.5.4 when $n \geq 15$.

When n is greater than 20, the number of on-set coefficients is comparatively small. To convert these sparse functions, multiple pointer technique may not be necessary. We find from the experiments that the conversion will be slightly faster without multiple pointer technique for the sparse functions.

Furthermore, to compare with the results from reference [85], we also test the parity functions defined as follows.

```
$convert(n, M, p, d, input\_file, output\_file)$
    {
        /* $n$, $M$, $p$, and $d$ are the numbers of variables, the number
    of on-set coefficients, the polarity, and the direction
    respectively. If $d$ is zero, then the conversion is from SOP
    coefficients to Reed-Muller coefficients. Otherwise, if $d$ is one,
    the conversion is from Reed-Muller coefficients to SOP
    coefficients. All the on-set coefficients are read from the
    $input\_file$. All the on-set coefficients after conversion are saved
    in the $output\_file$. */

      $CPU\_time = clock()$;
      $q \Leftarrow 32$;
      $s = 2^n$;
    /* $s$ and $q$ are the number of segments and the number of pointers
    respectively */
      $if(s > (q + \sqrt{q^2 + 4M}))s \Leftarrow s/2$;
      $u = 2^n/s$;
      $for(i = 0; i < M; i++)$
      {
      $temp \Leftarrow read\_a\_coefficient(input\_file) \wedge (\bar{d} * p)$;
      $if(j * u \leq temp < (j+1) * u)Ms[j]++$;
      }        /* count the numbers of coefficients for the $jth$ segment,
    $0 \leq j < s$, save them in the array $Ms[j]$ */
      $for(i = 0; i < s; i++)$
      {
      $if(Ms[i]\, is\, not\, a\, multiple\, of\, q)MMs[i] \Leftarrow Ms[i] + (q - Ms[i]\%q)$;
      $else\, MMs[i] \Leftarrow Ms[i]$;
      $make\_memory(MMs[i])$;
      }       /* make the numbers of coefficients in all segments be
    multiples of $q$, and make memory for all the coefficients */
      $for(i = 0; i < M; i++)$
      {
      $temp \Leftarrow read\_a\_coefficient(input\_file)$;
      $temp \Leftarrow temp \wedge (\bar{d} * p)$;
      $if(j * u \leq temp < (j+1) * u)temp \Leftarrow temp - j * u$;
      }      /* read all the coefficients to the memory and subtract
    them according to theorem 5.2 */
      $for(i = 0; i < s; i++)add\_extra\_numbers(Ms[i], MMs[i])$;
      $convert\_with\_zero\_polarity(u, s, d, p, q, MMs[\,])$;
      $CPU\_time \Leftarrow clock() - CPU\_time$;
      $display(CPU\_time\ and\ the\ number\ of\ on\text{–}set\ coefficients)$;
    }
```

Figure 5.3: Bidirectional conversion algorithm for large Boolean functions

n	number of on-set coefficients before conversion	number of on-set coefficients after conversion	polarity	direction	CPU times with multiple techniques(s)	CPU times without multiple techniques(s)
15	10,000	16,696	990	1	0.74	13.28
15	10,000	20,827	10,000	0	0.77	13.22
15	15,000	16,398	990	0	1.14	20.58
15	15,000	16,896	990	1	1.11	20.06
16	15,000	33,202	12,345	0	1.63	39.74
16	15,000	33,792	12,345	1	1.61	41.16
16	30,000	32,853	23,456	0	2.53	89.57
16	30,000	33,440	23,456	1	2.45	89.78
17	30,000	66,080	34,567	0	3.73	180.41
17	30,000	66,880	34,567	1	3.68	180.33
17	60,000	65,372	56,789	0	7.28	371.21
17	60,000	66,248	56,789	1	6.93	374.99
18	60,000	121,658	123,456	0	10.49	739.87
18	60,000	132,496	123,456	1	10.31	749.30
18	120,000	130,471	234,567	0	16.97	1,522.36
18	120,000	131,558	234,567	1	16.03	1,511.68
19	120,000	129,200	345,678	0	27.06	2,977.76
19	120,000	263,116	345,678	1	24.07	3,053.22
19	240,000	259,392	456,789	0	40.83	6,252.67
19	240,000	261,356	456,789	1	39.03	6,242.29
20	240,000	260,055	678,901	0	68.48	12,415.58
20	240,000	522,712	678,901	1	62.75	12,430.17
20	480,000	519,743	789,012	0	131.28	26,174.18
20	480,000	521,974	789,012	1	125.31	26,165.60
21	480,000	1,041,166	890,123	0	176.58	50,560.50
21	480,000	1,043,948	890,123	1	172.68	50,499.83
21	500,000	1,042,416	10,000	0	184.36	52,720.70

Table 5.3: Conversion results of some random coefficient sets

$$f(x_{n-1}x_{n-2}\cdots x_0) = \sum_{i=0}^{n-1} x_i$$

Both the polarity and the direction are set to be zero in our algorithm while converting the parity functions. It should be noted that the tests in reference [85] are performed using Borland C++ on a PC with Pentium_90MHz CPU and 16M RAM under WIN32 platform. These results are shown in table 5.4 and fig.5.5.

Finally, we compare our method with the results from a recently published paper [138] using fast tabular technique. In [138], the CPU time for conversion depends on the polarity as well as n and M. To calculate the time in [138], the FPRM expansions for all possible

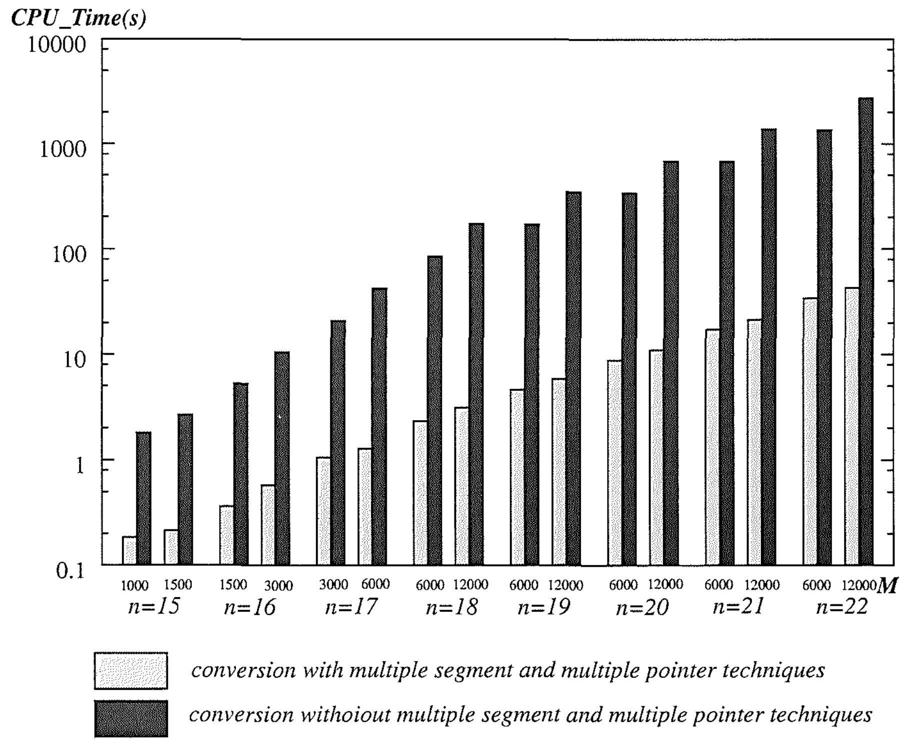


Figure 5.4: CPU Time versus the number of on-set coefficients for conversion

n	number of on-set coefficients before conversion	number of on-set coefficients after conversion	CPU time with multiple techniques(s)	CPU time without multiple techniques(s)	CPU time from reference [85]* (s)
15	16,384	15	1.48	22.14	140.39
16	32,768	16	2.63	99.09	560.47
17	65,536	17	7.90	401.94	2,254.69
18	131,072	18	17.42	1,658.11	9,129.83
19	262,144	19	53.43	6,780.51	36,757.92

* The results are performed on a PC with Pentium_90MHz CPU and 16M RAM.

Table 5.4: Test results for conversion of parity functions

polarities (from 0 to $2^n - 1$) were found, and their time average was taken for a given n and M. In our method, the computation time depends on n and M only and is independent of the polarity and direction of conversion. From fig.3 of [138], the average conversion times for the logic functions with 80% of on-set minterms are about 35, 85, 220 seconds when n is 8, 9, 10 respectively. The conversion was implemented in MATLAB on a PC. Applying our method with multiple techniques for the same functions, our conversion times are about 0, 0.01, 0.03 seconds respectively.

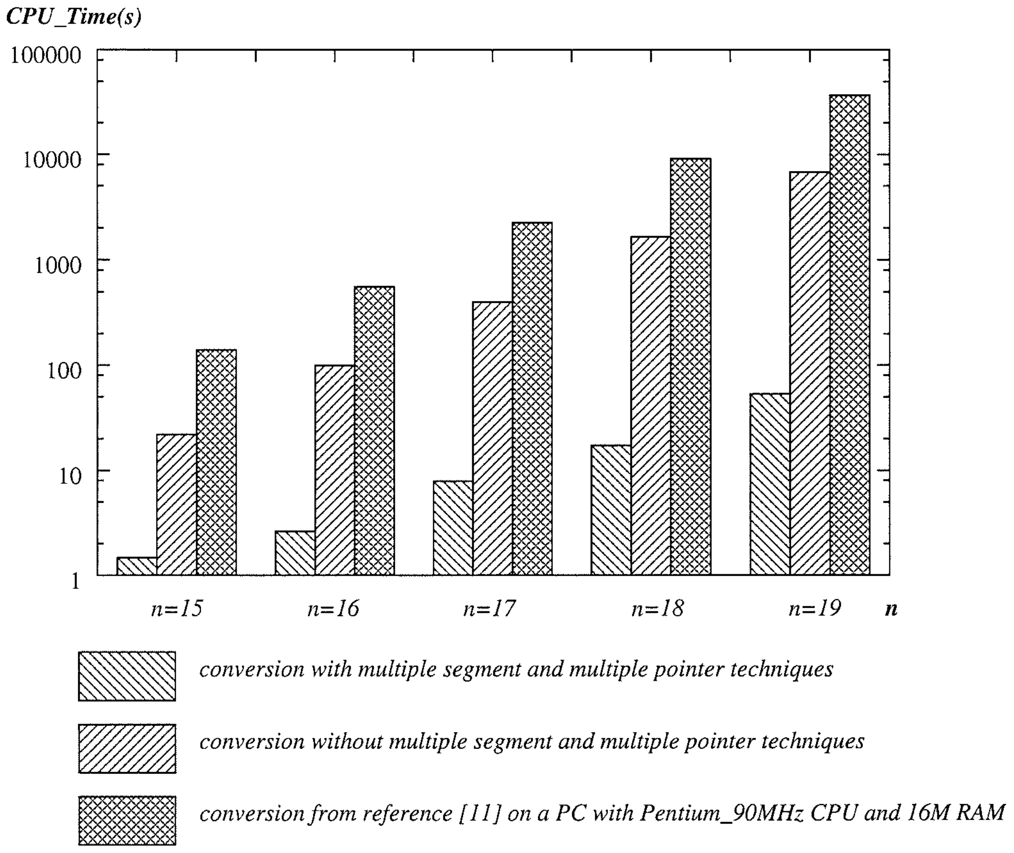


Figure 5.5: CPU time for parity function conversions

5.7 Summary

We propose the polarity for SOP form of any Boolean function through which the bidirectional conversion between SOP and Reed-Muller forms is direct. The CPU time for conversion is nearly independent of the polarity and the direction for a given number of variables and on-set coefficients. The speed of the algorithm is much faster than existing algorithms when multiple segment and multiple pointer techniques are used. From fig.5.2, the time complexity of our algorithm is $O(2^{1.5n})$, while it is $O(4^n)$ as reported in [85]. The space complexity is $O(2^n)$, same as in [85] since our conversion is also manipulated on on-set coefficients only. The algorithm is tested for randomly generated functions of up to 30 variable and 500,000 on-set coefficients. In the absence of a method for predicting the best polarity, short of exhaustive search, this method makes the search for a "good" polarity a practical reality.

Chapter 6

Conversion Algorithm for Very Large Multiple Output Functions

6.1 Introduction

Any n-variable Boolean function can be expressed canonically by the sum-of-products forms (SOPs) as follows.

$$f(x_{n-1}x_{n-2}\cdots x_0) = \sum_{i=0}^{2^n-1} a_i \dot{x}_{n-1}\dot{x}_{n-2}\cdots\dot{x}_0 \tag{6.1}$$

where each product is a minterm, "$\sum$" is the OR operator, $a_i \in \{0,1\}$, $\dot{x}_j \in \{x_j, \bar{x}_j\}$, $0 \leq j \leq n-1$. Alternatively, the function can be expressed by the fixed polarity Reed-Muller (FPRM) expressions as follows.

$$f(x_{n-1}x_{n-2}\cdots x_0) = \bigoplus_{i=0}^{2^n-1} b_i \dot{\dot{x}}_{n-1}\dot{\dot{x}}_{n-2}\cdots\dot{\dot{x}}_0 \tag{6.2}$$

where "$\bigoplus$" is the XOR operator, $b_i \in \{0,1\}$, $\dot{\dot{x}}_j \in \{1, \dot{x}_j\}$, $0 \leq j \leq n-1$. Furthermore, $\dot{x}_j$ in all the cubes can only be either true or complemented, but not both, which corresponds with a polarity p, $0 \leq p \leq 2^n - 1$. For very large Boolean functions with $n \geq 25$, it is impractical to use millions of on-set minterms as in equation (6.1) to express a function. Consequently, cube set expressions (commonly known as PLA description) and binary decision diagrams(BDDs) are the common representations based on AND/OR operations, corresponding with two-level and multilevel forms respectively. Accordingly, mixed polarity Reed-Muller expressions and functional decision diagrams (FDDs) are the common representations for very large Boolean functions based on AND/XOR operations. A cube set expression can be first converted to disjoint cube format in the standard Boolean domain[61], which is also called operational domain[68]. Then a mixed polarity two-level Reed-Muller expression is straightforward to obtain by replacing "OR" with "XOR" operation. However, the input irredundancy cannot be guaranteed in a mixed polarity Reed-Muller expression since it is not canonical[114]. Consequently, the mixed polarity Reed-Muller expression should be transformed to FPRM expressions or other procedures are employed to remove the redundant inputs such as the application of "linking rules" in [114]. For instance, a 4-variable Boolean function can be expressed by a mixed polarity Reed-Muller form as follows.

$$f(x_3x_2x_1x_0) = \bar{x}_3\bar{x}_1x_0 \oplus \bar{x}_3\bar{x}_2x_1 \oplus x_3x_0 \oplus x_3x_1\bar{x}_0 \oplus \bar{x}_3x_2x_1 \tag{6.3}$$

If this function is expressed by an FPRM with zero polarity, then we have equation (6.4).

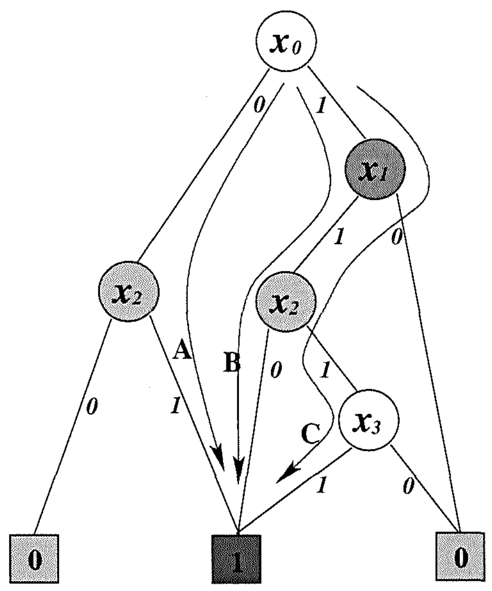
$$f(x_3x_2x_1x_0) = x_0 \oplus x_1 \oplus x_1x_0 \tag{6.4}$$

It can be easily seen from the above equation that x_2 and x_3 are redundant variables, while it is very difficult to draw the same conclusion from equation (6.3). However, there is no efficient program available to convert directly from SOPs to canonical FPRM expressions for very large functions because the space and time complexity increases exponentially with the number of variables as described in chapter 5.

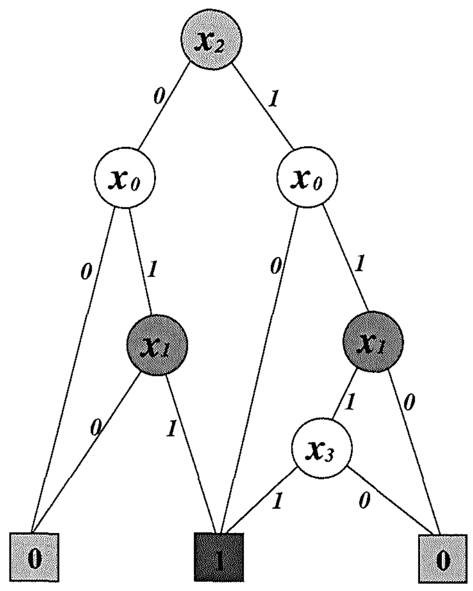
An alternative for obtaining FPRM expressions is by constructing functional decision diagrams (FDDs)[53]. A two-level FPRM form can be obtained from the corresponding ordered FDDs (OFDDs) where each 1-path defines a subset of the variables that uniquely corresponds to a cube in the FPRM form. Thereby, the number of 1-paths of a OFDD with fixed polarity variables is also the number of on-set cubes of the two-level FPRM for the same function [53]. Hence, a function that has compact OFDD structure can usually be represented by a two-level FPRM expression effectively. For example, an FDD for a 4-variable function is shown in fig.6.1(a) where each node is decomposed by Davio expansion as in equation (6.5).

$$f = f_0 \oplus xf_1 \tag{6.5}$$

In equation (6.5), if f is an n-variable function, then f_0 and f_1 are n-1 variable functions, independent of variable x. There are three on-set paths in fig.6.1(a) from the root node x_0 to the terminal node "1", namely, $x_0 \rightarrow x_2 \rightarrow 1$, $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow 1$, and $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow 1$. These paths are marked with "A", "B" and "C" respectively in fig.6.1(a) and the corresponding cubes are x_2, x_1x_0, and $x_3x_2x_1x_0$. Hence this function can be expressed as



(a) with variable order (x0, x1, x2, x3)



(b) with variable order (x2, x0, x1, x3)

Figure 6.1: Functional decision diagrams for $f = x_2 \oplus x_1x_0 \oplus x_3x_2x_1x_0$

an FPRM form in equation (6.6).

$$f = x_2 \oplus x_1x_0 \oplus x_3x_2x_1x_0 \tag{6.6}$$

The main disadvantage of FDDs is that they are generally not canonical except OFDD where the order of variables is fixed. Additionally, the size of FDDs is sensitive to the order of variables and the problem of finding the optimal OFDD is NP-complete as discussed for OBDDs in section 2.3. The same function in fig.6.1(a) can be represented by another FDD in fig.6.1(b) with a different variable order and number of nodes. However in this case, the same expression is obtained as in equation (6.6) from fig.6.1(b).

Although it is proposed in [53] that the size of OFDDs is a lower bound for the number of literals in FPRM expansions, in practice the size of FPRM is usually measured by the number of terms instead of literals. The main reason for that conclusion is that FPRMs are two-level representation while OFDDs are multilevel allowing many common literals to be factored out and shared. Moreover, two-level FPRM expressions are canonical and easily testable. Therefore, the conversion from SOPs to FPRM forms is desirable especially for very large functions without utilizing FDDs.

Due to the fact that the computational complexity usually increases exponentially with the number of input variables, an efficient approach can be achieved by possibly reducing the variable number. In this chapter, a fast algorithm is proposed to convert directly from SOPs to FPRM forms without generating mixed polarity Reed-Muller expressions or OFDDs. The algorithm takes advantage of the inherent redundancy commonly encountered in very large multiple output Boolean functions.

6.2 Algorithm

Observation 6.1. *In the PLA description of a Boolean function, if the entries on column i of all on-set cubes corresponding with a variable x_i are "-"s, then x_i is a redundant variable. Consequently, the entries on column i of all FPRM on-set cubes of this function are "0"s for the redundant variables with any polarity.*

Number of Dependent Variables	Number of Outputs	Number of Dependent Variables	Number of Outputs
2	3	10	3
3	1	11	5
4	2	12	17
5	51	13	1
6	3	19	1
7	3	20	3
8	2	24	1
9	3	***Total: 135 inputs, 99 outputs***	

Table 6.1: Distribution of dependent variables

For very large multiple output Boolean functions, each individual function usually does not depend on all the input variables. In other words, there are a number of redundant variables. For example, testcase "apex6", which is available in IWLS'93 and MCNC benchmarks, has 135 input variables and 99 outputs. More than half of the outputs are actually dependent on 5 variables only. The distribution of the number of dependent variables is shown in table 6.1. Consequently, a fast algorithm can be proposed as below based on observation 6.1.

Algorithm 6.1. *Given an n-variable m-output function in traditional PLA format, the following steps will produce a FPRM format with any polarity p, $0 \leq p \leq 2^n - 1$.*

1. Delete all the redundant variables for any individual function f_i, $0 \leq i \leq m-1$, where the entries are "-"s for all the on-set cubes in the PLA description. Suppose the number of redundant variables is r, then the number of dependent variables is $(n-r)$ for f_i.

2. Use the algorithm proposed in chapter 5 *to convert this single-output function f_i of $(n-r)$ variables from SOP to Reed-Muller form with polarity p and return a cube set C containing all the on-set Reed-Muller cubes.*

3. Add "0" to the columns of all the cubes in C for redundant variables from observation 6.1.

4. Repeat steps 1 to 3, and append the new Reed-Muller cubes to C for all other individual functions. Merge all the common Reed-Muller cubes in C to obtain the final FPRM format.

$x_2x_1x_0$	f_1f_0
–0 1	1 0
–1 0	1 0
0 0 –	0 1
1 0 –	0 1

(a) traditional PLA format

$x_2x_1x_0$	f_1f_0
0 0 0	0 1
0 0 1	1 0
0 1 0	1 1

(b) FPRM format with zero polarity

Figure 6.2: An example for algorithm 6.1

Example 6.1. A three-variable two-output function is shown as PLA format in fig.6.2(a), $f_0 = \bar{x}_2\bar{x}_1 + x_2\bar{x}_1$, and $f_1 = \bar{x}_1x_0 + x_1\bar{x}_0$. From observation 6.1, it can be seen that x_0 and x_2 are redundant for the functions f_0 and f_1 respectively. Convert these two functions of less variables from SOPs to FPRM format with default zero polarity using the method in chapter 5 and we have,

$$f_0 = 1 \oplus x_1 \tag{6.7}$$

and

$$f_1 = x_0 \oplus x_1 \tag{6.8}$$

Note the absence of the redundant variables in equations (6.7) and (6.8). Thus two cube sets can be obtained, $C_0 = \{00, 01\}$ for f_0 and $C_1 = \{01, 10\}$ for f_1. Then add "0" to the cubes of C_0 and C_1 for the redundant variables x_0 and x_2 respectively based on observation 6.1. Hence $C_0 = \{000, 010\}$ and $C_1 = \{001, 010\}$, which are shown in fig.6.2(b) after merging the common cube $\{010\}$. Furthermore, it can be seen that x_2 is actually redundant for both f_0 and f_1 from fig.6.2(b) while it is very difficult to draw this conclusion from fig.6.2(a).

6.3 Experimental Results

Algorithm 6.1 is implemented in C language and tested with IWLS'93 benchmark on a personal computer with Pentium II-266 CPU and 64M RAM. For most of the very large multiple output functions, an individual function usually does not depend on all the input variables. The results are shown in table 6.2, where the number of input variables(*variable number*), number of outputs (*output number*), cube number of two-level FPRM description with default zero polarity after merging the common cubes (*RM cube number with 0 polarity*), and the time for conversion(*time for conversion*) are also presented. The input file of the program is the PLA description of a multiple output Boolean function, and the output file is in FPRM format with any polarity p, $0 \leq p \leq 2^n - 1$, and n is the number of variables.

testcase	*variable number*	*output number*	*RM cube number with 0 polarity*	*time for conversion(s)*
apex6	135	99	11615	1090.62
b9	41	21	706	0.94
c8	28	18	460	0.30
cht	47	36	178	0.04
count	35	16	131137	156.34
example2	85	66	1076	0.81
i6	138	67	341	0.10
i7	199	67	330	0.11
i8	133	81	41874	158.26
lal	26	19	745	0.48
misex2	25	18	1100	0.16
pcler8	27	17	104	0.44
term1	34	10	9081	122.72
unreg	36	16	132	0.03
x3	135	99	11615	1151.96
x4	94	71	3174	2.10

Table 6.2: Experimental results of very large functions from IWLS'93 benchmark

From the results shown in table 6.2, it can be seen that the time for conversion does not depend on the number of input variables as in all the previous algorithms[5, 6, 68, 85, 155, 138], but depends on the structure of the functions. If a function has high redundancy, then it is still very fast even for very large function with hundreds of inputs. For example, "i7" testcase has 199 inputs and 67 outputs and most of input variables are redundant for any individual function. So the total conversion time is only 0.11 second while it takes much longer time for "term1" with much less inputs and outputs. For comparison, we tested the "misex2" and "lal" functions without applying the redundancy removal and found that it took more than ten hours to finish the conversion of either of them by the algorithm in chapter 5. No other comparison is available since all the previous published methods are not suitable for very large multiple output functions due to the excessive complexity.

6.4 Summary

A very fast algorithm is proposed in this chapter to convert very large multiple output Boolean functions directly from two-level PLA format to two-level FPRM format with any polarity using the property of input redundancy. This facilitates the canonical representation of very large multiple output Boolean functions ($n \geq 25$) by two-level FPRM forms. The space and time complexity does not depend on the number of variables but the structure of the functions. It lays an important basis for further research on Reed-Muller logic and logic synthesis and optimization generally.

Chapter 7

Exact Minimization of Fixed Polarity Reed-Muller Expressions

7.1 Introduction

Since the publication of the classic paper by Shannon in 1938[135], there has been great progress in the minimization of two-level Boolean functions[26, 42]. Additionally, any Boolean function can be expressed canonically based on AND and XOR operators using what is commonly known as Reed-Muller(RM) expansions. Besides, Reed-Muller realizations have several attractive advantages especially for functions that do not produce efficient solutions using SOP techniques[2]. Unfortunately, the techniques for synthesis and minimization of combinational logic using Reed-Muller forms are more difficult than those based on SOP expressions. Although there have been extensive research on Reed-Muller methods [3-9, 67-70, 155], they are still in their early stage of development. With recent improvement in layout technology and increased use of FPGAs where the XOR gate is already manufactured as a basic cell component[124], research on Reed-Muller logic received even more attention.

In fixed polarity Reed-Muller (FPRM) expressions, each variable can only be either true or complemented, but not both. There are two major steps in the minimization of FPRM expressions. The first one is to convert between SOP and FPRM expressions. The second one is to find the best polarity expression that has the least number of terms. Various minimization methods for these two steps can be classified into two categories[127]:

1. Gray code: Search all 2^n polarities sequentially and find the best one. Space and time complexities are $O(2^n)$ and $O(4^n)$ respectively;

2. Extended truth vector: Obtain the costs of 2^n expansions simultaneously by an extended truth vector and a weight vector. Both space and time complexities are $O(3^n)$.

It is generally accepted that the exact minimization of FPRM expressions is suitable for Boolean functions with less than 15 variables[56, 127]. A number of heuristic methods have been proposed recently using functional decision diagrams (FDDs) and genetic algorithm (GA) techniques[157]. The disadvantage of FDDs is that determination of the optimal variable order is impractical for large functions[8]. Besides, we do not know how good a polarity is by heuristic methods. In this chapter, an exact method to find the best polarity for FPRM forms is proposed based on the on-set coefficients by gray code with space complexity $O(M)$ and time complexity $O(2^n M)$, where M is the average number of on-set coefficients.

7.2 Background

For convenience, some definitions which are available in chapter 5 are reproduced here.

Any n-variable Boolean function can be expressed canonically by the SOP form in equation (7.1).

$$f(x_{n-1}x_{n-2}\cdots x_0) = \sum_{i=0}^{2^n-1} a_i m_i \tag{7.1}$$

where the subscript i can also be written as a binary n-tuple $i = (i_{n-1}i_{n-2}\cdots i_0)$, "$\sum$" is the OR operator, the minterm m_i can be represented as $m_i = \dot{x}_{n-1}\dot{x}_{n-2}\cdots\dot{x}_0$,

$$\dot{x}_j = \begin{cases} \bar{x}_j,\ i_j = 0 \\ x_j,\ i_j = 1 \end{cases} \tag{7.2}$$

Alternatively, it can be expressed by the positive polarity Reed-Muller (PPRM) form as follows[68].

$$f(x_{n-1}x_{n-2}\cdots x_0) = \bigoplus_{i=0}^{2^n-1} b_i \pi_i \tag{7.3}$$

where "$\bigoplus$" is the XOR operator, $\pi_i = \dot{x}_{n-1}\dot{x}_{n-2}\cdots\dot{x}_0$,

$$\dot{x}_j = \begin{cases} 1,\ i_j = 0 \\ x_j,\ i_j = 1 \end{cases} \tag{7.4}$$

In the previous equations, a_i, $b_i \in \{0, 1\}$, $0 \le j \le n-1$. Furthermore, PPRM forms can be extended to FPRM forms with any fixed polarity p, $p = (p_{n-1}p_{n-2}\cdots p_0)$, where every variable can only be either true or complemented, but not both. If a binary bit of p, p_j is 0 (or 1) then the corresponding variable is in the true (or complemented) form. Therefore, there are 2^n polarities for a n-variable function, and the positive polarity is equivalent to zero polarity. In [116], and further discussed in [148], a variable can be either a free variable or a bound variable where a bound variable is allowed to be negated in a minterm in order to determine a FPRM expansion with another polarity. This idea has been extended to the polarity for SOP forms in [153] as shown in definition 7.1.

Definition 7.1. Any Boolean function $f(x_{n-1}x_{n-2}\cdots x_0)$ can be expressed canonically as in equation (7.1). That expression is defined as the zero polarity of SOP expansion. Any variable $\ddot{x}_j$, $j \in \{0, 1, \cdots, n-1\}$ in every minterm with a polarity $p = (p_{n-1}p_{n-2}\cdots p_0)$ for the same Boolean function $f(\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots \ddot{x}_0)$ is defined as in equation (7.5).

$$\ddot{x}_j = \begin{cases} \bar{x}_j, & \text{\textit{if}}\ p_j = 1 \\ x_j, & \text{\textit{if}}\ p_j = 0 \end{cases} \tag{7.5}$$

According to equation (7.5), if a binary bit of p, p_j is 0 (or 1), that variable is in the true (or complemented) form. Now equation (7.1) is extended accordingly as follows.

$$f(\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots \ddot{x}_0) = \sum_{i=0}^{2^n-1} a_i m_i \tag{7.6}$$

where $m_i = \breve{x}_{n-1}\breve{x}_{n-2}\cdots \breve{x}_0$,

$$\breve{x}_j = \begin{cases} \bar{\ddot{x}}_j,\ i_j = 0 \\ \ddot{x}_j,\ i_j = 1 \end{cases} \tag{7.7}$$

In equation (7.7), $\bar{\ddot{x}}$ is the complemented form of $\ddot{x}$. Besides, equation (7.7) is an extension of equation (7.2). The corresponding subscript in equation (7.6), $i = (i_{n-1}i_{n-2}\cdots i_0)$ can be obtained from equation (7.7) as follows.

$$i_j = \begin{cases} 0,\ \breve{x}_j = \bar{\ddot{x}}_j \\ 1,\ \breve{x}_j = \ddot{x}_j \end{cases} \tag{7.8}$$

Definition 7.2. Consider two integers expressed by binary n-tuples, $i = \{i_{n-1}i_{n-2}\cdots i_0\}$, $j = \{j_{n-1}j_{n-2}\cdots j_0\}$. If $i_k \geq j_k$ for all k, $0 \leq k \leq n-1$, then i covers j or j is covered by i.

Definition 7.3. The distance between two integers, expressed as binary n-tuples, $p = (p_{n-1}p_{n-2}\cdots p_0)$ and $q = (q_{n-1}q_{n-2}\cdots q_0)$, is the number of binary bits that are different from each other. If the distance is "1", then they are adjacent.

Definition 7.4. The Kronecker product of two Boolean matrices, $\mathbf{P}_{a\times b}$, $\mathbf{Q}_{c\times d}$ is a matrix of dimension $ac \times bd$ given by

$$\mathbf{P}\otimes\mathbf{Q} = \begin{bmatrix} p_{00}\mathbf{Q} & p_{01}\mathbf{Q} & \cdots & p_{0(b-1)}\mathbf{Q} \\ p_{10}\mathbf{Q} & p_{11}\mathbf{Q} & \cdots & p_{1(b-1)}\mathbf{Q} \\ \vdots & \vdots & \vdots & \vdots \\ p_{(a-1)0}\mathbf{Q} & p_{(a-1)1}\mathbf{Q} & \cdots & p_{(a-1)(b-1)}\mathbf{Q} \end{bmatrix}_{ac\times bd} \tag{7.9}$$

Definition 7.5. A $2^n \times 2^n$ matrix $\mathbf{Q}$ has $2^{n+1}-1$ diagonal lines. The main diagonal line is defined as diagonal line 0. The ith diagonal line that is above the main diagonal line is defined as diagonal line i. If it is below the main diagonal line, then it is defined as diagonal line $-i$. This definition can be shown below when n is 2.

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 \\ -3 & -2 & -1 & 0 \end{bmatrix}$$

Lemma 7.1. *Any Boolean function can be uniquely expressed by its on-set coefficient set $\mathbb{C}_0 = \{i\}$ as shown later in example 7.1 with the default SOP polarity zero; while with polarity p, the corresponding set is $\mathbb{C}_p = \{i \wedge p\}$, where "$\wedge$" is the bitwise XOR operation*[153].

Lemma 7.2. *Any n-variable Boolean function can be uniquely expressed by a 2^n-dimensional vector, either $\mathbf{A}_p = [a_0, a_1, \cdots, a_{2^n-1}]^t$ in SOP form or $\mathbf{B}_p = [b_0, b_1, \cdots, b_{2^n-1}]^t$ in Reed-Muller form based on equations (7.1) and (7.3) respectively. These two vectors can be converted mutually by the following recursive transform matrix $\mathbf{T}_n$ as long as they have the same polarity p*[153], $0 \le p \le 2^n - 1$.

$$\mathbf{T}_1 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

when n is 1;

$$\mathbf{T}_n = \begin{bmatrix} \mathbf{T}_{n-1} & 0 \\ \mathbf{T}_{n-1} & \mathbf{T}_{n-1} \end{bmatrix}$$

when n is greater than 1. Besides, $\mathbf{T}_n^2 = \mathbf{I}$ where $\mathbf{I}$ is the unit matrix. Alternatively, they can be expressed as in equation (7.10).

$$\mathbf{A}_p = \mathbf{T}_n\mathbf{B}_p \qquad \textit{or} \qquad \mathbf{B}_p = \mathbf{T}_n\mathbf{A}_p \tag{7.10}$$

From lemma 7.1, when an n-variable Boolean function is expressed by a 2^n-dimensional vector, the effect of a SOP polarity p is to exchange the orders of two coefficients i and $i \wedge p$ since $(i \wedge p) \wedge p = i$, $0 \le i, p \le 2^n - 1$. Consequently, the element number of $\mathbb{M}_p$ does not vary with any polarity p.

Example 7.1. Consider a 3-variable function $f(x_2x_1x_0) = \sum(1,2,5)$ with the default SOP polarity zero. Since $\bar{\bar{x}} = x$, we have

$$\begin{aligned}
f(x_2x_1x_0) &= \sum(1,2,5) \\
&= \sum\{(001),(010),(101)\} \\
&= \bar{x}_2\bar{x}_1x_0 + \bar{x}_2x_1\bar{x}_0 + x_2\bar{x}_1x_0 \\
&= \bar{x}_2\bar{x}_1\bar{\bar{x}}_0 + \bar{x}_2x_1\bar{x}_0 + x_2\bar{x}_1\bar{\bar{x}}_0
\end{aligned}$$

If the polarity p is 1, then $\ddot{x}_2 = x_2$, $\ddot{x}_1 = x_1$, and $\ddot{x}_0 = \bar{x}_0$ according to equation (7.5). Hence, from equations (7.6) and (7.8), this function can also be represented as $f(\breve{x}_2\breve{x}_1\breve{x}_0)$ with polarity 1,

$$\begin{aligned}
f(\breve{x}_2\breve{x}_1\breve{x}_0) &= f(x_2x_1x_0) \\
&= \bar{x}_2\bar{x}_1\bar{\bar{x}}_0 + \bar{x}_2x_1\bar{x}_0 + x_2\bar{x}_1\bar{\bar{x}}_0 \\
&= \sum\{(000),(011),(100)\} \\
&= \sum(0,3,4)
\end{aligned}$$

This function is $\mathbb{C}_0 = \{1,2,5\}$ with zero polarity, while it is $\mathbb{C}_1 = \{0,3,4\}$ with polarity 1 when expressed by on-set coefficient sets. Notice that $\mathbb{C}_0$ and $\mathbb{C}_1$ have the same number of elements. This result can be verified by lemma 7.1 because $\{1 \wedge 1, 2 \wedge 1, 5 \wedge 1\} = \{0,3,4\}$. Therefore, any n-variable Boolean function can be expanded canonically as in equation (7.6) with polarity p according to definition 7.1. Alternatively, this function can be represented by PPRM form through the transform matrix in lemma 7.2. In this function, $\mathbf{A}_0 = [0,1,1,0,0,1,0,0]^t$, hence, $\mathbf{B}_0 = \mathbf{T}_3\mathbf{A}_0 = [0,1,1,0,0,0,1,1]^t$ with polarity zero. When the polarity is 1, $f(\check{x}_2\check{x}_1\check{x}_0) = \sum(0,3,4)$, thus $\mathbf{A}_1 = [1,0,0,1,1,0,0,0]^t$. From lemma 7.2, $\mathbf{B}_1 = \mathbf{T}_3\mathbf{A}_1 = [1,1,1,0,0,0,0,1]^t$. The only difference between $\mathbf{A}_0$ and $\mathbf{A}_1$ is the order of on-set coefficients. In $\mathbf{A}_0$, the three on-set coefficients are at positions 1, 2 and 5 while in $\mathbf{A}_1$, the three on-set coefficients are at positions 0, 3, and 4. In other words, in order to obtain $\mathbf{A}_1$ the elements in $\mathbf{A}_0$ at positions 1, 2 and 5 are exchanged with the elements at positions 0, 3 and 4. If they are expressed by on-set coefficient sets in Reed-Muller logic, then $\mathbb{R}_0 = \{1,2,6,7\}$ and $\mathbb{R}_1 = \{0,1,2,7\}$ with polarity 0 and 1 respectively.

7.3 Properties of the polarities for SOP and FPRM expressions

Theorem 7.1. *Given two 2^n-dimensional vectors $\mathbf{A}_0$, $\mathbf{A}_p$ for an n-variable completely specified Boolean function with SOP polarities 0 and p respectively, then*

$$\mathbf{A}_p = \mathbf{S}_n|_p\mathbf{A}_0 \tag{7.11}$$

where

$$\mathbf{S}_n|_p = \begin{array}{cc} \begin{array}{ccccccc} & & & \text{column}\,(i\wedge p) & & & \end{array} & \\ \left[\begin{array}{ccccccc} & & & \cdots & & & \\ & & & \vdots & & & \\ 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ & & & \vdots & & & \\ & & & \cdots & & & \end{array}\right]_{2^n\times 2^n} & \begin{array}{c} \\ \\ \text{row } i \\ \\ \\ \end{array} \end{array} \tag{7.12}$$

In other words, there is only one "1" entry in column $(i \wedge p)$ of any row i in $\mathbf{S}_n|_p$, $0 \le i, p \le 2^n - 1$.

Proof. From lemma 7.1, any element i in the on-set coefficient set $\mathbb{C}_0$ with polarity 0 will be converted to $i \wedge p$ with polarity p for the same function. Besides, any minterm that is not on-set is off-set. Therefore, for any row i in $\mathbf{S}_n|_p$, $0 \le i \le 2^n - 1$, there is only one "1" entry in column $(i \wedge p)$. We use $\mathbf{S}$ to represent this transform matrix since it can be taken as a "sorting" process of the on-set coefficients according to polarity p. When n is fixed, $\mathbf{S}_n|_p$ can be written as $\mathbf{S}_p$ for simplicity. □

Example 7.2. In example 7.1, $n = 3$, $\mathbf{A}_0 = [0,1,1,0,0,1,0,0]^t$. When p is 1,

$$\mathbf{S}_3|_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Hence, $\mathbf{A}_1 = \mathbf{S}_3|_1\mathbf{A}_0 = [1,0,0,1,1,0,0,0]^t$.

Theorem 7.2. *For any polarity p of an n-variable Boolean function, $0 \leq p \leq 2^n - 1$, $n \geq 1$,*

$$(\mathbf{S}_n|_p)^2 = \mathbf{I} \tag{7.13}$$

where $\mathbf{I}$ is the unit matrix.

Theorem 7.2 can be easily proved because for any coefficient i, $i \wedge p \wedge p = i$, $0 \leq i \leq 2^n - 1$.

Theorem 7.3. *For two polarities p and q of n-variable Boolean function, $0 \leq p, q \leq 2^n - 1$, $\mathbf{S}_p = \mathbf{S}_{p \wedge q}\mathbf{S}_q$.*

Theorem 7.3 can be easily proved because for any coefficient i, $(i \wedge p \wedge q) \wedge q = i \wedge p$, $0 \leq i \leq 2^n - 1$.

From theorem 7.3, theorem 7.1 can be generalized when the given vector is $\mathbf{A}_q$ with polarity q, rather than $\mathbf{A}_0$ with default polarity 0, $0 \leq q \leq 2^n - 1$. In this case, equation (7.11) is extended to equation (7.14).

$$\mathbf{A}_p = \mathbf{S}_n|_{p \wedge q}\mathbf{A}_q \tag{7.14}$$

Theorem 7.4. *The transform matrix $\mathbf{S}_n|_p$ in theorem 7.1 can be represented in the following recursive way where p is the SOP polarity expressed as a binary n-tuple $p = (p_{n-1}p_{n-2}\cdots p_0)$, and n is the variable number, $n \geq 1$.*

If n is 1, then

$$\mathbf{S}_1|_{p=0} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \mathbf{S}_1|_{p=1} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

otherwise,

$$\mathbf{S}_n|_p = \begin{cases} \begin{bmatrix} \mathbf{S}_{n-1}|_{p'} & 0 \\ 0 & \mathbf{S}_{n-1}|_{p'} \end{bmatrix}_{2^n\times 2^n} & \text{if } p_{n-1}=0 \\ \\ \begin{bmatrix} 0 & \mathbf{S}_{n-1}|_{p'} \\ \mathbf{S}_{n-1}|_{p'} & 0 \end{bmatrix}_{2^n\times 2^n} & \text{if } p_{n-1}=1 \end{cases} \tag{7.15}$$

where p_{n-1} is the most significant bit (MSB) of p, and p' is the same as p except that the MSB is set to 0.

Proof. It can be proved by induction on n as below.

(1) when n is 2, from equation (7.12),

$$\mathbf{S}_2|_{p=0} = \begin{bmatrix} 1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1 \end{bmatrix} = \begin{bmatrix} \mathbf{S}_1|_0 & 0 \\ 0 & \mathbf{S}_1|_0 \end{bmatrix}$$

Similarly,

$$\mathbf{S}_2|_{p=1} = \begin{bmatrix} 0&1&0&0\\1&0&0&0\\0&0&0&1\\0&0&1&0 \end{bmatrix} = \begin{bmatrix} \mathbf{S}_1|_1 & 0 \\ 0 & \mathbf{S}_1|_1 \end{bmatrix}$$

$$\mathbf{S}_2|_{p=2} = \begin{bmatrix} 0&0&1&0\\0&0&0&1\\1&0&0&0\\0&1&0&0 \end{bmatrix} = \begin{bmatrix} 0 & \mathbf{S}_1|_0 \\ \mathbf{S}_1|_0 & 0 \end{bmatrix}$$

$$\mathbf{S}_2|_{p=3} = \begin{bmatrix} 0&0&0&1\\0&0&1&0\\0&1&0&0\\1&0&0&0 \end{bmatrix} = \begin{bmatrix} 0 & \mathbf{S}_1|_1 \\ \mathbf{S}_1|_1 & 0 \end{bmatrix}$$

(2)Suppose when n is k, $k \geq 1$, equation (7.15) is true. When n is $k+1$, $p = (p_k p_{k-1} \cdots p_0)$ and the corresponding vector $\mathbf{A}$ can be divided into two parts, $\mathbf{A} = [\mathbf{A}'|\mathbf{A}'']$, where $\mathbf{A}'$ is the first half of 2^k elements, and $\mathbf{A}''$ is the other half of 2^k elements.

(a)When p_k is 0, then $p = p'$. From lemma 7.1, a coefficient i exchanges the order with another coefficient $i \wedge p$, $0 \leq i \leq 2^k - 1$. In other words, all the coefficients exchange their orders within $\mathbf{A}'$ and $\mathbf{A}''$. Therefore,

$$\mathbf{S}_{k+1}|_p = \mathbf{S}_{k+1}|_{p'} = \begin{bmatrix} \mathbf{S}_k|_{p'} & 0 \\ 0 & \mathbf{S}_k|_{p'} \end{bmatrix}$$

(b)When p_k is 1, then $p = 2^k + p'$ where "+" is the arithmetic addition. From lemma 7.1, a coefficient i exchanges the order with another coefficient $i \wedge p = i \wedge (2^k + p')$. Hence, a coefficient i in $\mathbf{A}'$ will swap with the coefficient $i \wedge (2^k + p')$ that is in $\mathbf{A}''$, $0 \leq i \leq 2^k - 1$. In the same way, a coefficient i in $\mathbf{A}''$ will swap with the coefficient $i \wedge (2^k + p')$ that is in $\mathbf{A}'$, $2^k \leq i \leq 2^{k+1} - 1$. Thus,

$$\mathbf{S}_{k+1}|_p = \begin{bmatrix} 0 & \mathbf{S}_k|_{p'} \\ \mathbf{S}_k|_{p'} & 0 \end{bmatrix}$$

From both (a) and (b), equation (7.15) is true when n is $k + 1$. □

Based on theorems 7.1 and 7.4, any Boolean function expressed in SOP forms with any polarity p can be computed through the transform matrix $\mathbf{S}$ in either equation (7.12) or (7.15). In the same way, a transform matrix to convert a Reed-Muller expansion of n-variable Boolean function from polarity 0 to any other polarity p can be obtained from theorem 7.5 based on theorem 7.4.

Theorem 7.5. *Consider an n-variable Boolean function $f(x_{n-1}x_{n-2}\cdots x_0)$ expressed by a 2^n-dimensional vector, $\mathbf{B}_0 = [b_0, b_1, \cdots, b_{2^n-1}]^t$ with the default polarity 0 in Reed-Muller logic. Then this function can be represented by another vector $\mathbf{B}_p$ with polarity p shown in equation (7.16).*

$$\mathbf{B}_p = \mathbf{P}_n|_p \mathbf{B}_0 \tag{7.16}$$

where

$$\mathbf{P}_n|_p = \bigotimes_{i=n-1}^{0} \begin{bmatrix} 1 & p_i \\ 0 & 1 \end{bmatrix} \tag{7.17}$$

and p_i is any binary bit of p, $p = (p_{n-1}p_{n-2}\cdots p_0)$, $0 \leq i \leq n - 1$.

Proof. Given a vector $\mathbf{B}_0 = [b_0, b_1, \cdots, b_{2^n-1}]^t$ to express a Boolean function with the default polarity 0 in Reed-Muller logic. Then the following equations can be deduced based on lemma 7.2 and theorem 7.1.

$$\begin{aligned}
\mathbf{B}_p &= \mathbf{T}_n\mathbf{A}_p && (7.18)\\
&= \mathbf{T}_n\mathbf{S}_n|_p\mathbf{A}_0 && (7.19)\\
&= \mathbf{T}_n\mathbf{S}_n|_p\mathbf{T}_n\mathbf{B}_0 && (7.20)
\end{aligned}$$

Comparing equation (7.20) with equation (7.16), it can be seen that

$$\mathbf{P}_n|_p = \mathbf{T}_n\mathbf{S}_n|_p\mathbf{T}_n \tag{7.21}$$

It will be shown that $\mathbf{T}_n\mathbf{S}_n|_p\mathbf{T}_n = \bigotimes_{i=n-1}^{0} \begin{bmatrix} 1 & p_i \\ 0 & 1 \end{bmatrix}$ by induction on n as below.

(1)When n is 1, there are only two polarities, 0 and 1.

$$\begin{aligned}
\mathbf{T}_1\mathbf{S}_1|_{p=0}\mathbf{T}_1 &= \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}\\
&= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\\
&= \bigotimes_{i=1-1}^{0} \begin{bmatrix} 1 & p_i \\ 0 & 1 \end{bmatrix}
\end{aligned}$$

$$\begin{aligned}
\mathbf{T}_1\mathbf{S}_1|_{p=1}\mathbf{T}_1 &= \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}\\
&= \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}\\
&= \bigotimes_{i=1-1}^{0} \begin{bmatrix} 1 & p_i \\ 0 & 1 \end{bmatrix}
\end{aligned}$$

(2) Suppose when n is k, $k \geq 1$, $\mathbf{P}_k|_{p'} = \mathbf{T}_k\mathbf{S}_k|_{p'}\mathbf{T}_k$ for any polarity p'. When n is $k+1$, the MSB of p can be either 0 or 1. Let p' be the same as p except that the MSB is 0.

(a)If the MSB of p is 0, then $p = p'$.

$$\begin{aligned}
\mathbf{T}_{k+1}\mathbf{S}_{k+1}|_p\mathbf{T}_{k+1} &= \mathbf{T}_{k+1}\mathbf{S}_{k+1}|_{p'}\mathbf{T}_{k+1} \\
&= \begin{bmatrix} \mathbf{T}_k & 0 \\ \mathbf{T}_k & \mathbf{T}_k \end{bmatrix} \begin{bmatrix} \mathbf{S}_k|_{p'} & 0 \\ 0 & \mathbf{S}_k|_{p'} \end{bmatrix} \begin{bmatrix} \mathbf{T}_k & 0 \\ \mathbf{T}_k & \mathbf{T}_k \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{T}_k & 0 \\ \mathbf{T}_k & \mathbf{T}_k \end{bmatrix} \begin{bmatrix} \mathbf{S}_k|_{p'}\mathbf{T}_k & 0 \\ \mathbf{S}_k|_{p'}\mathbf{T}_k & \mathbf{S}_k|_{p'}\mathbf{T}_k \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{T}_k\mathbf{S}_k|_{p'}\mathbf{T}_k & 0 \\ 0 & \mathbf{T}_k\mathbf{S}_k|_{p'}\mathbf{T}_k \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{P}_k & 0 \\ 0 & \mathbf{P}_k \end{bmatrix} \\
&= \bigotimes_{i=k}^{0} \begin{bmatrix} 1 & p_i \\ 0 & 1 \end{bmatrix}
\end{aligned}$$

(b)If the MSB of p is 1,

$$\begin{aligned}
\mathbf{T}_{k+1}\mathbf{S}_{k+1}|_p\mathbf{T}_{k+1} &= \begin{bmatrix} \mathbf{T}_k & 0 \\ \mathbf{T}_k & \mathbf{T}_k \end{bmatrix} \begin{bmatrix} 0 & \mathbf{S}_k|_{p'} \\ \mathbf{S}_k|_{p'} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{T}_k & 0 \\ \mathbf{T}_k & \mathbf{T}_k \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{T}_k & 0 \\ \mathbf{T}_k & \mathbf{T}_k \end{bmatrix} \begin{bmatrix} \mathbf{S}_k|_{p'}\mathbf{T}_k & \mathbf{S}_k|_{p'}\mathbf{T}_k \\ \mathbf{S}_k|_{p'}\mathbf{T}_k & 0 \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{T}_k\mathbf{S}_k|_{p'}\mathbf{T}_k & \mathbf{T}_k\mathbf{S}_k|_{p'}\mathbf{T}_k \\ 0 & \mathbf{T}_k\mathbf{S}_k|_{p'}\mathbf{T}_k \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{P}_k & \mathbf{P}_k \\ 0 & \mathbf{P}_k \end{bmatrix} \\
&= \bigotimes_{i=k}^{0} \begin{bmatrix} 1 & p_i \\ 0 & 1 \end{bmatrix}
\end{aligned}$$

From both (a) and (b), equation (7.17) is true when n is $k+1$. □

Theorem 7.5 offers another method to compute Reed-Muller coefficients with polarity p from only the on-set Reed-Muller coefficients with polarity 0 which can be shown as follows .

Procedure 7.1. *Given an on-set Reed-Muller coefficient set $\mathbb{R}_0$ for an n-variable Boolean function with polarity 0. A coefficient set $\mathbb{R}_p$ with any other polarity p can be obtained through the following steps:*

(1) Set $\mathbb{R}_p$ to be $\emptyset$.

(2) In matrix $\mathbf{P}_n|_p$, delete column i if and only if $i \notin \mathbb{R}_0$, $0 \le i \le 2^n - 1$. Thus $\mathbf{P}'_n|_p$ is obtained with less columns.

(3) Count the number of "1"s in any row i of $\mathbf{P}'_n|_p$, $0 \leq i \leq 2^n - 1$. If the number is odd, add i to $\mathbb{R}_p$.

(4) $\mathbb{R}_p$ is the on-set Reed-Muller coefficient set with polarity p.

Example 7.3. When n is 3, any Boolean function can be represented by a 2^n-dimensional vector, $\mathbf{B}_0 = [b_0, b_1, \cdots, b_7]^t$ with the default polarity 0 in Reed-Muller logic. With polarity $p = (p_2 p_1 p_0)$, the vector $\mathbf{B}_p$ will be,

$$\mathbf{B}_p = \mathbf{P}_3|_p \mathbf{B}_0$$

where

$$\begin{aligned}
\mathbf{P}_3|_p &= \bigotimes_{i=2}^{0} \begin{bmatrix} 1 & p_i \\ 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & p_2 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & p_1 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & p_0 \\ 0 & 1 \end{bmatrix}
\end{aligned}$$

$$= \begin{bmatrix}
1 & p_0 & p_1 & p_1p_0 & p_2 & p_2p_0 & p_2p_1 & p_2p_1p_0 \\
 & 1 & & p_1 & & p_2 & & p_2p_1 \\
 & & 1 & p_0 & & & p_2 & p_2p_0 \\
 & & & 1 & & & & p_2 \\
 & & & & 1 & p_0 & p_1 & p_1p_0 \\
 & & & & & 1 & & p_1 \\
 & & & & & & 1 & p_0 \\
 & & & & & & & 1
\end{bmatrix} \tag{7.22}$$

In equation (7.22), all the empty elements are 0.

In example 7.1, the on-set Reed-Muller coefficient set $\mathbb{R}_0 = \{1, 2, 6, 7\}$ with polarity 0. By procedure 7.1, $\mathbb{R}_1$ can be computed with polarity $p = 1 = (001)$ in the following steps:

(1) Set $\mathbb{R}_1$ to be $\emptyset$.

(2) On the matrix $\mathbf{P}_3|_p$ from equation (7.22), delete columns 0, 3, 4, 5. Thus

$$\mathbf{P}_3^{'}|_p = \begin{bmatrix} p_0 & p_1 & p_2p_1 & p_2p_1p_0 \\ 1 & & & p_2p_1 \\ & 1 & p_2 & p_2p_0 \\ & & & p_2 \\ & & p_1 & p_1p_0 \\ & & & p_1 \\ & & 1 & p_0 \\ & & & 1 \end{bmatrix}$$

where the empty elements are 0.

(3) When $p = 1 = (001)$, $\mathbf{P}_3^{'}|_1 = \begin{bmatrix} 1 & & & \\ 1 & & & \\ & 1 & & \\ & & & \\ & & & \\ & & & \\ & & 1 & 1 \\ & & & 1 \end{bmatrix}$.

Rows 0, 1, 2 and 7 have odd number of "1"s. Hence $\mathbb{R}_1$ $=\{0, 1, 2, 7\}$ as in example 7.1.

Theorem 7.6. *For any polarity p of an n-variable Boolean function, $0 \leq p \leq 2^n - 1$, $n \geq 1$,*

$$(\mathbf{P}_n|_p)^2 = \mathbf{I} \tag{7.23}$$

where $\mathbf{I}$ *is the unit matrix.*

Proof. It can be proved by induction on n as below.

(1) When n is 1, from equation (7.17), $\mathbf{P}_1|_p = \begin{bmatrix} 1 & p \\ 0 & 1 \end{bmatrix}$. Hence,

$$\begin{bmatrix} 1 & p \\ 0 & 1 \end{bmatrix}^2 = \begin{bmatrix} 1 & p \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & p \\ 0 & 1 \end{bmatrix} = \mathbf{I}$$

(2) Suppose when n is k, $(\mathbf{P}_k|_{p'})^2 = \mathbf{I}$, $k \geq 1$, $0 \leq p' \leq 2^k - 1$. If $n = k + 1$, let the

MSB of p be p_k. From equation (7.17),

$$\begin{aligned}
(\mathbf{P}_{k+1}|_p)^2 &= \left(\begin{bmatrix} 1 & p_k \\ 0 & 1 \end{bmatrix} \mathbf{P}_k|_{p'}\right)^2 \\
&= \begin{bmatrix} \mathbf{P}_k|_{p'} & p_k\mathbf{P}_k|_{p'} \\ 0 & \mathbf{P}_k|_{p'} \end{bmatrix}^2 \\
&= \begin{bmatrix} \mathbf{P}_k|_{p'}\mathbf{P}_k|_{p'} & 0 \\ 0 & \mathbf{P}_k|_{p'}\mathbf{P}_k|_{p'} \end{bmatrix} \\
&= \mathbf{I}
\end{aligned}$$

□

Theorem 7.7. *For two polarities p and q of n-variable Boolean function, $0 \le p, q \le 2^n - 1$, $\mathbf{P}_p = \mathbf{P}_{p\wedge q}\mathbf{P}_q$.*

Proof. It can be proved by induction on n as below.

(1) When n is 1, from equation (7.17), $\mathbf{P}_1|_p = \begin{bmatrix} 1 & p \\ 0 & 1 \end{bmatrix}$, $\mathbf{P}_1|_q = \begin{bmatrix} 1 & q \\ 0 & 1 \end{bmatrix}$.

$$\begin{bmatrix} 1 & p \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & p \wedge q \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & q \\ 0 & 1 \end{bmatrix}$$

This satisfies the theorem when $n = 1$.

(2) Suppose when n is k, $\mathbf{P}_k|_{p'} = \mathbf{P}_k|_{p'\wedge q'}\mathbf{P}_k|_{q'}$, $k \ge 1$, $0 \le p', q' \le 2^k - 1$. If $n = k + 1$, let the MSB of p and q be p_k and q_k respectively. From equation (7.17),

$$\begin{aligned}
\mathbf{P}_{k+1}|_{p\wedge q}\mathbf{P}_{k+1}|_q &= \left(\begin{bmatrix} 1 & p_k \wedge q_k \\ 0 & 1 \end{bmatrix} \otimes \mathbf{P}_k|_{p'\wedge q'}\right)\left(\begin{bmatrix} 1 & q_k \\ 0 & 1 \end{bmatrix} \otimes \mathbf{P}_k|_{q'}\right) \\
&= \begin{bmatrix} \mathbf{P}_k|_{p'\wedge q'} & (p_k \wedge q_k)\mathbf{P}_k|_{p'\wedge q'} \\ 0 & \mathbf{P}_k|_{p'\wedge q'} \end{bmatrix} \begin{bmatrix} \mathbf{P}_k|_{q'} & q_k\mathbf{P}_k|_{q'} \\ 0 & \mathbf{P}_k|_{q'} \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{P}_k|_{p'} & p_k\mathbf{P}_k|_{p'} \\ 0 & \mathbf{P}_k|_{p'} \end{bmatrix} \\
&= \mathbf{P}_{k+1}|_p
\end{aligned}$$

□

Now theorem 7.5 can be extended to theorem 7.8 when the given vector is $\mathbf{B}_q$ with polarity q rather than $\mathbf{B}_0$ with the default polarity 0.

Theorem 7.8. *Any n-variable Boolean function can be expressed by 2^n-dimensional vectors $\mathbf{B}_p$ and $\mathbf{B}_q$ with polarity p and q respectively, $0 \le p, q \le 2^n - 1$. Then,*

$$\mathbf{B}_p = \mathbf{P}_{p \wedge q} \mathbf{B}_q \tag{7.24}$$

Proof. Let this n-variable Boolean function be expressed by $\mathbf{B}_0$ with polarity 0 in Reed-Muller logic. From theorem 7.5 and 7.7, we have

$$\begin{aligned}\mathbf{B}_p &= \mathbf{P}_p \mathbf{B}_0 \\ &= \mathbf{P}_{p \wedge q} \mathbf{P}_q \mathbf{B}_0 \\ &= \mathbf{P}_{p \wedge q} \mathbf{B}_q\end{aligned}$$

In the same way, procedure 7.1 can be adapted as follows. □

Procedure 7.2. *Given an on-set Reed-Muller coefficient set $\mathbb{R}_p$ for an n-variable Boolean function with polarity p. A coefficient set $\mathbb{R}_q$ with any other polarity q can be obtained through the following steps:*

(1) Set $\mathbb{R}_q$ to be $\emptyset$.

(2) In matrix $\mathbf{P}_n|_{p \wedge q}$, delete column i if and only if $i \notin \mathbb{R}_p$, $0 \leq i \leq 2^n - 1$. Thus $\mathbf{P}'_n|_{p \wedge q}$ is obtained with less columns.

(3) Count the number of "1"s in any row i of $\mathbf{P}'_n|_{p \wedge q}$, $0 \leq i \leq 2^n - 1$. If the number is odd, add i to $\mathbb{R}_q$.

(4) $\mathbb{R}_q$ is the on-set Reed-Muller coefficient set with polarity q.

From theorem 7.1 and theorem 7.5, there are 2^n polarities to express any n-variable Boolean function with either SOP or FPRM forms. By lemma 7.2 the effect of polarity for SOP forms is to reorder all the on-set coefficients. Besides, the distance between two on-set coefficients, defined in definition 7.3, is always the same because $(i \wedge p) \wedge (j \wedge p) = (i \wedge j)$ for two integers i, j with polarity p. Therefore, the best polarity of FPRM form is equivalent to the best "order" of SOP form while the pairwise distances of any two on-set coefficients are fixed. This may provide a good understanding of the "center of gravity" or "Boolean center" proposed in [146]. Furthermore, the conversion from one polarity to another polarity for both canonical forms can be implemented by two transform matrices $\mathbf{S}$ and $\mathbf{P}$ based on equations (7.11) and (7.16). Besides, theorems 7.2, 7.3 and theorems 7.6, 7.7 show that the polarities of SOP and FPRM forms have very similar properties. While comparing equations (7.12) and (7.17), it can be concluded that transform matrix $\mathbf{P}$ is much more complex than $\mathbf{S}$, which makes the synthesis and minimization of FPRM expressions much more difficult.

7.4 Best polarity for single output functions

Results from theorem 7.5 agree with those from theorem 1 in [94]. The fast transformation in [94], however, requires that all the 2^n elements of the vector be saved, which is not practical for large values of n. Based on our experience, the number of on-set coefficients usually does not increase exponentially with the number of variables. Procedure 7.2 only needs on-set Reed-Muller coefficients to convert the coefficients of a Boolean function from one polarity to any other polarity, yet there are still 2^n rows in $\mathbf{P}_n$ which must be saved to find the best polarity. This is not practical for large functions. A new fast algorithm to find the best polarity for completely specified single output Boolean function is proposed in this section based on definition 7.5.

Theorem 7.9. *Given any matrix $\mathbf{P}_n|_p$ for an n-variable Boolean function with polarity p, all the elements below the main diagonal line are 0. Furthermore, for two adjacent polarities p and q there are only two diagonal lines 0 and $p \wedge q$ in $\mathbf{P}_n|_{p\wedge q}$ where the elements are not 0. In diagonal line 0, the number of "1"s is 2^n while it is 2^{n-1} in diagonal line $p \wedge q$. Besides, any row that has two "1"s in columns c and c', $c' > c$, satisfies equation (7.25).*

$$c' = c + (p \wedge q) \tag{7.25}$$

where "+" is the arithmetic addition. The "1"s in diagonal line $p \wedge q$ can only reside in columns that cover $p \wedge q$, $0 \leq i \leq 2^n - 1$ based on definition 7.2.

Proof. theorem 7.9 can be easily proved by induction on n based on equation (7.17). □

Example 7.4. When n is 3, suppose $p = 2 = (010)$ and $q = 3 = (011)$. Hence $p \wedge q = (001) = 1$. By equation (7.22), we have

$$\mathbf{P}_3|_1 = \begin{bmatrix} 1 & 1 & & & & & & \\ & 1 & & & & & & \\ & & 1 & 1 & & & & \\ & & & 1 & & & & \\ & & & & 1 & 1 & & \\ & & & & & 1 & & \\ & & & & & & 1 & 1 \\ & & & & & & & 1 \end{bmatrix} \tag{7.26}$$

where all the empty elements are 0.

It can be seen from equation (7.26) that all the elements below the main diagonal line are 0 in $\mathbf{P}_3|_1$. There are only two diagonal lines 0 and 1 where the elements are not 0. In diagonal lines 0, the number of "1"s is 8, while it is $2^{3-1} = 4$ on diagonal lines 1. For any rows that have two "1"s, the difference between these two columns is always $p \wedge q = 1$.

Finally, the "1"s in diagonal line 1 can only reside in columns 1, 3, 5 and 7 that cover "1" based on definition 7.2.

By theorem 7.8, if an n-variable Boolean function can be expressed by a vector $\mathbf{B}_p = (b_{p(n-1)}, b_{p(n-2)}, \cdots, b_{p0})$ with polarity p, then it can be converted to $\mathbf{B}_q = \mathbf{P}_n|_r \mathbf{B}_p = (b_{q(n-1)}, b_{q(n-2)} \cdots, b_{q0})$ with polarity q, where $r = p \wedge q$. When polarities p and q are adjacent, if there is only one "1" on any row i in $\mathbf{P}_n|_r$, $0 \le i \le 2^n - 1$, then

$$b_{qi} = \sum_{j=0}^{2^n-1} p_{ij} b_{pj} = b_{pi} \tag{7.27}$$

where p_{ij} is the element in ith row and jth column of $\mathbf{P}_n|_r$. Otherwise, if there are two "1"s in one row, then from equation (7.25) we have,

$$b_{qi} = \sum_{j=0}^{2^n-1} p_{ij} b_{pj} = b_{pi} \oplus b_{p(p+r)} \tag{7.28}$$

Consequently, the element b_{pi} in $\mathbf{B}_p$ is different from b_{qi} in $\mathbf{B}_q$ if and only if row i in $\mathbf{P}_n|_r$ has two "1"s. Therefore, we only need to detect all the rows in $\mathbf{P}_n|_r$ where there are two "1"s. From theorem 7.9, all the "1"s in diagonal line r can only reside in the column that covers r. Based on this observation and the result in procedure 7.2, a fast procedure can be obtained to convert an on-set Reed-Muller coefficient set $\mathbb{R}_p$ with polarity p to $\mathbb{R}_q$ with polarity q of an n-variable Boolean function.

Procedure 7.3. *Given an on-set Reed-Muller coefficient set $\mathbb{R}_p$ for an n-variable Boolean function with polarity p. A coefficient set $\mathbb{R}_q$ with any adjacent polarity q can be achieved through the following operations on $\mathbb{R}_p$ itself where $p \wedge q = r$.*

(1) For any coefficient i in the set $\mathbb{R}_p$, if i does not cover r, then i is an element of $\mathbb{R}_q$ because $b_{qi} = b_{pi}$ from equation (7.27). Therefore leave i in the set. If i covers r, from equation (7.28) we need to search the set $\mathbb{R}_p$ for the coefficient $(i - r)$. If there is such a coefficient, then delete coefficient $(i - r)$ from the set $\mathbb{R}_p$ because $b_{qi} = b_{pi} \oplus b_{p(p+r)} = 1 \oplus 1 = 0$. Otherwise, if there is not such a coefficient, then add coefficient $(i - r)$ to the set $\mathbb{R}_p$ because $b_{qi} = b_{pi} \oplus b_{p(p+r)} = 0 \oplus 1 = 1$.

(2) The new set obtained in step (1) is the on-set Reed-Muller coefficient set with polarity p.

Example 7.5. In example 7.1, the 3-variable function can be expressed by an on-set coefficient set $\mathbb{R}_1 = \{0, 1, 2, 7\}$ when polarity p is 1. If polarity q is 0, then $r = p \wedge q = (001) \wedge (000) = 1$. The following steps will obtain the coefficient set with polarity 0.

(1)In $\mathbb{R}_1$, the first element is 0. Because it does not cover r, go to the next coefficient 1. Since 1 covers r, we need to search $1 - 1 = 0$ that is in $\mathbb{R}_1$. Hence, 0 is deleted from $\mathbb{R}_1$.

The third element 2 in $\mathbb{R}_1$does not cover r. Finally, for the last element 7, it covers r. In the same way, search $7 - 1 = 6$ which is not in $\mathbb{R}_1$. Therefore add "6" to $\mathbb{R}_1$.

(2)The new set $\{1, 2, 7, 6\}$ obtained in step (1) is the same result for $\mathbb{R}_0$ as in example 7.1.

From example 7.5, it can be seen that procedure 7.3 is very efficient in time and space. To obtain the new on-set coefficient set for another polarity, the old on-set coefficients need only be accessed once plus the time for possible search. Besides, the new coefficient set can be saved in the same array as used by the old one. In order to improve the speed, we can first sort the old coefficient set in ascending order so that the fast binary search can be applied.

Based on procedure 7.3 and theorem 7.8, a fast algorithm to find the best polarity for an n-variable single output completely specified Boolean function is shown below.

Procedure 7.4. *Given an on-set Reed-Muller coefficient set $\mathbb{R}_p$ for an n-variable Boolean function with the original polarity p. Let three integer variables, $COST$, $BESTCOST$ and $BESTPOLARITY$ represent the number of elements in the coefficient set, the least cost and the corresponding polarity respectively. For any i, $0 < i \leq 2^n - 1$, carry out steps (1) to (3).*

(1) Generate a polarity q_i in gray code order, that is adjacent with q_{i-1}. Let $r = q_{i-1} \wedge q_i$.

(2) Pass $\mathbb{R}_{q_{i-1}}$, and r to procedure 7.3 to get the new on-set coefficient set $\mathbb{R}_{q_i}$.

(3) Set $COST$ to be the element number of $\mathbb{R}_{q_{i-1}}$. If $COST$ is less than $BESTCOST$, then change $BESTCOST$ and $BESTPOLARITY$ to $COST$ and q_i respectively.

Output ($BESTPOLARITY \wedge p$) that is the best polarity with $BESTCOST$ on-set terms.

7.5 Best polarity for multiple output functions

In procedure 7.4 which is for a single output Boolean function, all the on-set Reed-Muller coefficients can be saved in the same array. For multiple output Boolean functions, the output part can be saved in another array with equal dimension so that each coefficient corresponds with one output element. Suppose the output part for a coefficient i is $\theta = (\theta_{m-1}\theta_{m-2} \cdots \theta_0)$, where $\theta_j \in \{0, 1\}$, $0 \leq j \leq m - 1$, and m is the number of outputs. If a term is the output for a sub-function f_k, then $\theta_k = 1$. Otherwise $\theta_k = 0$, as shown in figure 7.1 of example 7.6 later. This, which can be called Reed-Muller PLA format, is similar to that used for SOP [26]. In procedure 7.3, if a coefficient i does not cover $r = p \wedge q$, where p and q are the old and new polarities, then nothing will be changed for single output function. This result is also true for a multiple output function. If a coefficient i, whose output part is $\theta = (\theta_{m-1}\theta_{m-2} \cdots \theta_0)$, covers r, then there will be two possibilities:

(1) There is a coefficient $(i - r)$ in the coefficient set whose output part can be expressed as a binary m-tuple, $\eta = (\eta_{m-1}\eta_{m-2} \cdots \eta_0)$. In the single output function, $(i - r)$

should be deleted from the set. For any particular sub-function f_j, there are four cases to decide the new output part for the coefficient $(i-r)$.

☛ (a) $\theta_j = 0$, $\eta_j = 0$

➥ That means neither of two coefficients is a term of f_j. So the new output part for the coefficient $(i-r)$ is still η_j.

☛ (b) $\theta_j = 0$, $\eta_j = 1$

➥ That means coefficient i is not a term for f_j, but $(i-r)$ is. From procedure 7.3, the new output part for the coefficient $(i-r)$ is still η_j.

☛ (c) $\theta_j = 1$, $\eta_j = 0$

➥ That means coefficient i is a term for f_j, but $(i-r)$ is not. From procedure 7.3, $(i-r)$ should be added to the new output part for f_j. Therefore, the new output of coefficient $(i-r)$ for f_j is changed to 1.

☛ (d) $\theta_j = 1$, $\eta_j = 1$

➥ That means both coefficients are terms for f_j. From procedure 7.3, coefficient $(i-r)$ should be deleted from f_i. So the new output part for the coefficient $(i-r)$ is changed to 0.

From the above four cases, it can be seen that η_j should be replaced by $\theta_j \wedge \eta_j$.

(2) There is not such a coefficient $(i-r)$ in the coefficient set. From procedure 7.3, $(i-r)$ should be added to the coefficient set. At the same time, the output part θ will be copied for the outputs of both coefficients.

Based on these observations, procedure 7.4 can be extended to multiple output function as follows.

Procedure 7.5. *Given an on-set Reed-Muller coefficient set $\mathbb{R}_p$ and the corresponding output set $\mathbb{M}_p$ for an n-variable Boolean function with the original polarity p. The number of elements in $\mathbb{R}_p$ is COST. Set BESTPOLARITY and BESTCOST to be 0 and COST respectively. For any i, $0 < i \leq 2^n - 1$, carry out steps (1) to (4).*

(1) Generate a polarity q_i in gray code order, that is adjacent with q_{i-1}. Let $r = q_{i-1} \wedge q_i$. Set RELATIVECOST to be 0.

(2) For any coefficient j in the set $\mathbb{R}_{q_{i-1}}$, whose output element is θ, if it does not cover r, then nothing should be changed. If it covers r, then search the set $\mathbb{R}_{q_{i-1}}$ for coefficient $(j-r)$. If there is such a coefficient whose output element is η, then change η to $\eta \wedge \theta$. If η

becomes 0, then delete it from the output set and coefficient $(j - r)$ should also be removed from the coefficient set. Besides, set $RELATIVECOST$ to be $(RELATIVECOST - 1)$. Otherwise, if there is not such a coefficient, then add coefficient $(j - r)$ to the coefficient set $\mathbb{R}_{q_{i-1}}$ and copy θ as its output element. Further, set $RELATIVECOST$ to be $(RELATIVECOST + 1)$.

(3) Now both $\mathbb{R}_{q_{i-1}}$ and $\mathbb{M}_{q_{i-1}}$ becomes $\mathbb{R}_{q_i}$ and $\mathbb{M}_{q_i}$. Set $COST$ to be $(COST + RELATIVECOST)$. If $COST$ is less than $BESTCOST$, then change $BESTCOST$, $BESTPOLARITY$ to $COST$ and q_i respectively.

(4) Sort the elements in $\mathbb{R}_{q_i}$ in ascending order. Update the output set accordingly to keep the function the same as before sorting.

Output $(BESTPOLARITY \wedge p)$ that is the best polarity with $BESTCOST$ on-set terms.

Example 7.6. Figure 7.1(a) shows a 3-variable 2-output Boolean function expressed in Reed-Muller PLA format with polarity $p = 0$. There are totally 5 on-set coefficients. Therefore, the coefficient and output sets are $\mathbb{R}_0 = \{0, 1, 2, 6, 7\}$, $\mathbb{M}_0 = \{1, 1, 3, 2, 3\}$ respectively. Besides, $COST = BESTCOST = 5$, $BESTPOLARITY = 0$. We now compute the on-set coefficients and the cost with polarity 1 following procedure 7.5.

(1)For the next polarity $q_1 = 1$, $r = q_0 \wedge q_1 = 0 \wedge 1 = 1$. $RELATIVECOST$ is initialized to be 0.

(2)For the first element $j = 0$ in $\mathbb{R}_0$, nothing will be changed because "0" does not cover "1". But the second element $j = 1$ covers r. So we need to search for $j - r = 1 - 1 = 0$ that is the first element. Then the output of the first element $\eta = 1$ should be replaced by $\eta \wedge \theta = 1 \wedge 1 = 0$. Now η is 0, thus the coefficient 0 and its output should be deleted. Besides, $RELATIVECOST = 0 - 1 = -1$. The following coefficients 2 and 6 do not cover r, hence go to the final coefficient $j = 7$ that covers r. There is a coefficient $j - r = 7 - 1 = 6$. Hence the output of the coefficient 6, η will be changed to $\eta \wedge \theta = 2 \wedge 3 = 1$. Now η is not zero, so coefficient 6 is still in the array but with the different output element. This result is shown in figure 7.1(b).

(3)Now we have $\mathbb{R}_1 = \{1, 2, 6, 7\}$, $\mathbb{M}_1 = \{1, 3, 1, 3\}$ as shown in fig.7.1(c). $COST$ is updated to $COST + RELATIVECOST = 5 - 1 = 4$ that is less than $BESTCOST$. Therefore, $BESTCOST = 4$, and $BESTPOLARITY = 1$.

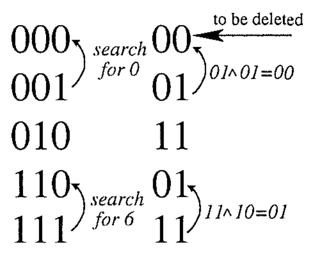
(4) Sort $\mathbb{R}_1 = \{1, 2, 6, 7\}$ and repeat steps (1)-(4) of procedure 7.5 in the same way for other polarities.

Finally, we find the best polarity that is 1 with 4 on-set coefficients.

In [35] and [56], the number of terms for multiple output function is computed through XOR operation on the subfunctions. However, in step 2 of procedure 7.5, all the outputs saved in an array are processed in a parallel way. Therefore, the speed of the algorithm does not directly depend on the number of outputs. In other words, step 2 of procedure

```
.i 3
.o 2
000  01
001  01
010  11
110  10
111  11
.e
```

(a) with polarity 0



(b) with polarity 1

```
.i 3
.o 2
001  01
010  11
110  01
111  11
.e
```

(c) with polarity 1 after deletion

Figure 7.1: Example for a 3-variable 2-output Reed-Muller function

7.5 does not care about the number of outputs. This conclusion can be verified in the following experimental results.

7.6 Experimental results

In our program, a Boolean function is first converted from SOP PLA format to the FPRM form with polarity zero using the method in chapter 5. If there is no PLA format, then apply SIS to convert from BLIF format to PLA format[134]. For incompletely specified Boolean functions, we just set the don't cares to off-sets. From the FPRM form with polarity 0, procedure 7.5 is applied to find the best polarity which has been implemented with C language. The program is compiled by the GNU C compiler egcs-2.91.66 and tested MCNC and IWLS'93 benchmarks on a personal computer with Cyrix6x86-166 CPU and 32M RAM under Linux operating system. For large Boolean functions, we test the program on a personal computer with PIII-450 CPU and 64M RAM. The results are shown in tables 7.1 and 7.2 respectively, where "/" represents "not available". The number of variables(n), the number of outputs(o), the number of terms with polarity 0(*init. term#*), the best polarity (*best polarity*) and the corresponding number of terms(*least term#*) are also presented. In [56], no results for the exact minimization of large Boolean functions are available. Furthermore, in [35], only small functions with $n \leq 10$ are tested due to the high space complexity. Therefore, no comparison is shown in table 7.2. From both tables 7.1 and 7.2, it can be seen that the speed of the program depends on the number of variables and the number of on-set coefficients. In procedure 7.5, for any polarity q_i of an n-variable multiple output function, $0 \leq i \leq 2^n - 1$, all the on-set coefficients need accessing once only plus a possible search in order to obtain the FPRM form with the next polarity. Further, all the 2^n polarities should be evaluated for exact minimization. Let M

be the average number of on-set coefficients as in equation (7.29).

$$M = \sum_{i=0}^{2^n-1} M_i \Big/ 2^n \tag{7.29}$$

where M_i is the number of on-set coefficients of FPRM form with polarity q_i and $\sum$ is the arithmetic addition. Then the time complexity of the program is $O(\sum_{i=0}^{2^n-1} M_i) = O(2^n M)$. In procedure 7.5, the same array can be used repeatedly by all the on-set coefficients. So the space complexity is $O(M)$. For example, testcase "m181" in table 7.1 has 15 variable, 9 outputs, but only about 200 on-set coefficients on average. It takes less time than a function with less variables such as "co14" since there are much more on-set coefficients in "co14". Furthermore, it can be seen that the speed of our program does not depend on the number of outputs directly. If two functions have the same product of $(2^n M)$ but different numbers of outputs, then the time to find the best polarity is the same. For "bw" testcase which has 28 outputs, it takes about the same time as "squar5" or "rd53". In [35], "bw" takes much longer time than "squar5" or "rd53" as shown in the last column of table 7.1. For large Boolean functions when $n > 15$, the number of on-set coefficients is usually small especially for arithmetic functions such as "t481", "ryy6" and "pm1" etc. Hence it is still very quick to find the best polarity. Comparing with the results in [35] and [56], our program is very efficient for exact polarity minimization of large Boolean functions.

7.7 Summary

The properties of the polarities of the SOP and FPRM forms are presented in this chapter. The comparison shows that these two kinds of polarities have great similarity but the transform matrix for the conversion between two FPRM forms has much more complex structure than the counterpart for SOP forms. Furthermore, the best polarity of FPRM forms with the least number of terms corresponds with the polarity for SOP forms with the best order of on-set minterms. This should lead to a new way for exact minimization of FPRM expansions without complex transform and exhaustive search. A fast algorithm is proposed to find the best polarity for completely specified multiple output functions using gray code sequence. The space and time complexities are $O(M)$ and $O(2^n M)$ respectively where M is the average number of on-set coefficients. In other words, the space and time complexities depend on the structure of the Boolean functions. This constitute a significant improvement on previous algorithms making it possible to tackle large functions with more than 15 variables. If a function has a "good" structure for Reed-Muller logic, then the space and time complexities are small. This characteristic is specially attractive for large arithmetic Boolean functions where M is usually small. The experimental results confirm these conclusions.

	n	*o*	*init. term#*	*best polarity*	*least term#*	*time (s)**	*time (s)*[56]†	*time (s)*[35]‡
bw	5	28	32	31	22	~0	/	0.94
squar5	5	8	23	0	23	~0	/	0.10
rd53	5	3	20	0	20	~0	0.5	0.10
con1	7	2	19	80	17	~0	/	0.10
rd73	7	3	63	0	63	0.01	2.3	0.10
5xp1	7	10	61	0	61	0.01	/	0.10
rd84	8	4	107	0	107	0.03	5.5	0.10
root	8	5	225	236	118	0.03	8.8	/
dist	8	5	216	15	185	0.04	12.5	/
log8mod	8	5	103	116	53	0.01	6.5	/
misex1	8	7	60	252	20	0.01	/	0.10
life	9	1	184	255	100	0.11	9.2	/
9sym	9	1	210	15	173	0.07	/	0.20
clip	9	5	217	71	206	0.12	/	0.20
sao2	10	4	1022	179	100	0.27	48.1	0.30
add6	12	7	132	0	132	0.85	295.1	/
co14	14	1	8192	16383	14	19.97	488.4	/
tial	14	8	4406	29	3683	66.09	8480.4	/
gary	15	11	6815	15998	349	37.78	16216.1	/
m181	15	9	213	31960	67	1.82	1149.0	/

∗Cyrix6x86-166 PC with 32M RAM

†HP Apollo series 700 workstation

‡HP Apollo series 715 workstation

Table 7.1: Test results for small Boolean functions

	n	*o*	*init. term#*	*best polarity*	*least term#*	*time (s)**
t481	16	1	41	39321	13	0.36
ryy6	16	1	80	49152	64	13.83
cmb	16	4	4097	63	132	3.72
pm1	16	13	37	36	27	0.77
table5	17	15	74504	109311	2458	342.94
tcon	17	16	24	0	24	0.43
pcle	19	9	72	1280	32	14.05
mux	21	1	81	0	81	48.01
cm150a	21	1	163	1	82	46.43
cc	21	20	59	327784	41	25.75
duke2	22	29	7088	118995	255	2045.22
ttt2	24	21	788	12596031	107	1750.09
misex2	25	18	1100	28290559	87	1785.20

∗Pentium III-450 PC with 64M RAM

Table 7.2: Test results for large Boolean functions

Chapter 8

Optimization of Reed-Muller PLA Implementations

8.1 Introduction

It is concluded in [126] that the Reed-Muller programmable logic array (RMPLA) structures, based on AND/XOR operations, are more economical than the conventional programmable logic array (PLA) structures based on AND/OR operations[26]. This, however, was demonstrated mainly for small functions. Further, the Reed-Muller circuits have great advantage of easy testability[118]. However, applications of RMPLAs have so far not become popular due to the following two obstacles.

1. XOR gates used to have slow speed and require large silicon area to realize in comparison with OR gates.
2. The problem of minimization of Reed-Muller functions is difficult although there has been a great deal of research in recent years.

With the development of new technologies and the advent of various field programmable gate array (FPGA) devices, the first obstacle has become irrelevant. In programmable devices, the XOR gate is either easily realized in "universal modules" or directly available. For instance, in the AT6000 FPGA series from ATMEL Corporation, logic blocks can be configured as various two-input gates such as XORs, ANDs and NANDs. In other FPGAs, both AND and XOR gates are available in the macrocells or logic array blocks (LABs)[29]. However, the minimization of Reed-Muller expressions remains much more difficult than sum-of-products (SOPs)[26]. Although there have been extensive research on Reed-Muller logic optimization, most of the available methods are not suitable for very large functions. There are three basic procedures associated with the optimization of RMPLA implementations.

1. Conversion algorithms to convert a Boolean function from SOP to fixed polarity Reed-Muller (FPRM) format. The optimization method is more efficient starting from the Reed-Muller domain than from the standard Boolean domain[120], which are also called functional domain and operational domain respectively[68]. One obvious advantage of FPRM forms over SOP or mixed polarity Reed-Muller forms, is the absence of redundant variables[114].

2. Polarity optimization methods to find the best polarity with the least number of product terms starting from the initial FPRM expression obtained from the preceding procedure. It is conjectured in [120] that "the minimum number of mixed polarity terms of a function is independent of the Reed-Muller polarity from which the mixed polarity representation was obtained". Nevertheless, starting the minimization from an FPRM form with the best polarity will greatly reduce the initial complexity. Thus the speed of the decomposition method can be improved, as will be illustrated in our examples later.

3. Mixed polarity minimization to further reduce the number of product terms by combining the adjacent ones.

For the first procedure, many algorithms have been published which are based on Reed-Muller matrix transformations[68, 73], tabular techniques[4, 6, 98], and Binary Decision Diagrams (BDDs)[8, 9, 116] among others. Most of these approaches are not suitable for very large functions. In chapter 6, very large multiple output Boolean functions can be converted mutually between conventional SOPs and fixed polarity Reed-Muller (FPRM) formats using multiple segment technique presented in chapter 5. Experimental result shows that it only takes about 0.1 seconds to convert a function with 199 inputs and 67 outputs run on a personal computer(PC). The second procedure which is to find the best polarity is computationally extensive in both space and time especially for large functions. It is generally accepted that exact minimization of FPRMs is only suitable when the number of input variables is less than 15 on a common PC[5, 56, 127]. Several heuristic methods have been proposed which apply the simulated annealing [111] or genetic algorithm techniques[157]. The product term number can be further reduced with mixed polarity by combining the adjacent product terms such as using xorlink operation[137]. The final results can be implemented in two-level PLA format which are called AND-XOR PLAs with one-bit decoders in [126].

An alternative simplification scheme for RMPLAs is given in [120]. An SOP expression is first represented with disjoint cubes using any methods such as in [61]. Then "-" and "0" are swapped for all the variables to obtain mixed polarity Reed-Muller expression with disjoint cubes. The rationales of this transformation are: (a) $c_0 + c_1 = c_0 \oplus c_1$ if cubes c_0 and c_1 are disjoint; (b) "-" and "0" represent a missing and complemented variables respectively in Boolean domain, which correspond to "0" and "-" respectively in Reed-

Muller domain with the same functionality. Finally a (quasi-)optimal FPRM expression is exploited and a (quasi-)minimum expression with mixed polarity product terms is obtained by combining the adjacent terms. Unfortunately, experimental results are reported only for small functions[120].

There are also some other approaches that apply similar strategy as ESPRESSO starting from a mixed polarity Reed-Muller forms[49]. In this chapter, the decomposition method, based on the concept of $\frac{3}{4}$-majority cube[142], is further investigated and generalized to very large multiple output functions using the algorithm in chapter 5. The produced expressions belong to the most general class of AND/XOR forms, namely exclusive-OR sum-of-products (ESOPs)[127].

8.2 Review of the decomposition method

8.2.1 Background

Definition 8.1. An n-variable Boolean function can be represented canonically by an FPRM form with polarity p expressed in a binary n-tuple, $p = (p_{n-1}p_{n-2}\cdots p_0)_2$ as follows.

$$f(x_{n-1}x_{n-2}\cdots x_0) = \sum_{i=0}^{2^n-1} b_i \pi_i \tag{8.1}$$

where "$\sum$" is the XOR operator. Further, i can be written as a binary n-tuple $i = (i_{n-1}i_{n-2}\cdots i_0)_2$, $b_i \in \{0,1\}$, $\pi_i = \ddot{x}_{n-1}\ddot{x}_{n-2}\cdots\ddot{x}_0$,

$$\ddot{x}_j = \begin{cases} 1, & i_j = 0 \\ \dot{x}_j, & i_j = 1 \end{cases} \qquad \dot{x}_j = \begin{cases} \bar{x}_j, & p_j = 1 \\ x_j, & p_j = 0 \end{cases} \quad \textit{and} \quad 0 \le j \le n-1 \tag{8.2}$$

If b_i is 1, then the corresponding π_i is called an on-set pi-term; otherwise, it is an off-set pi-term. In fig.8.1(a), a 3-variable function is represented by a Reed-Muller coefficient map, b_j-map[155], which is the counterpart of a Karnaugh map in the standard Boolean logic. The algorithmic rules of grouping in a b_j-map are different from a Karnaugh map. In fig.8.1(a), the function is represented with polarity 0, $f(x_2x_1x_0) = 1 \oplus x_0 \oplus x_1x_0 \oplus x_2$. Each variable is in positive form and there are 4 on-set pi-terms and 4 off-set pi-terms. When the polarity is $6 = (110)_2$, both x_2 and x_1 are in complemented forms while x_0 is in positive form from equation (8.2). Thus the same function can be represented by two on-set pi-terms only, $f(\bar{x}_2\bar{x}_1x_0) = \bar{x}_1x_0 \oplus \bar{x}_2$ whose b_j-map is shown in fig.8.1(c).

After combining adjacent on-set pi-terms in an FPRM form, the result is in mixed polarity Reed-Muller form which can be implemented by an RMPLA shown in fig.8.2. Each variable has two inputs to the programmable array of AND gates due to the mixed

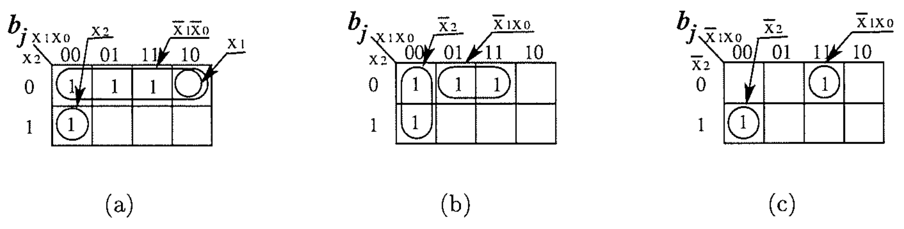


Figure 8.1: Limitation of $\frac{3}{4}$-majority m-cubes
(a) grouping of a $\frac{3}{4}$-majority cube with polarity 0 (b) simpler grouping with polarity 0
(c) grouping with polarity 6 where x_1 and x_2 are complemented

polarity. After the generation of all the on-set product terms, they are connected to the programmable array of XOR gates to produce the output for each function. The implementation of the function in fig.8.1(a) is shown in fig.8.2(b), where "×" indicates intact fuse.

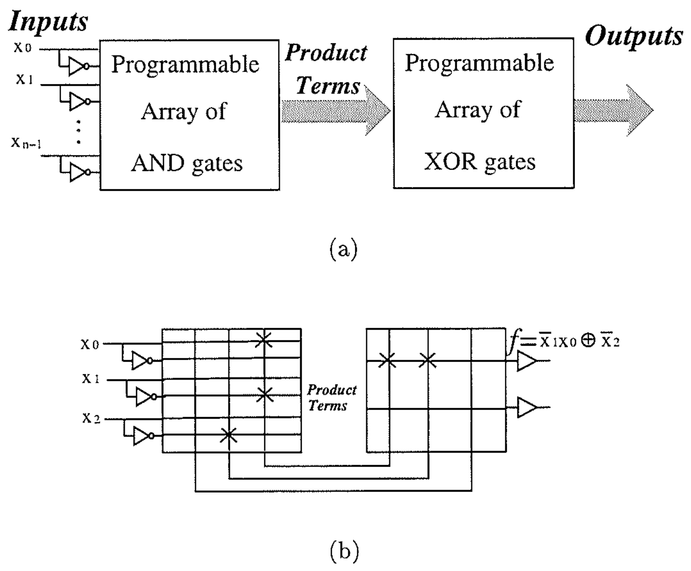

Inputs
x_0
x_1
x_{n-1}
Programmable Array of AND gates
Product Terms
Programmable Array of XOR gates
Outputs
(a)
x_0
x_1
x_2
Product Terms
$f = \bar{x}_1x_0 \oplus \bar{x}_2$
(b)

Figure 8.2: Structures of Reed-Muller PLAs
(a)Generic structure of Reed-Muller PLAs (b) Implementation of $f = \bar{x}_1x_0 \oplus \bar{x}_2$

8.2.2 Limitations of the decomposition method

The concept of $\frac{3}{4}$-majority m-cube is introduced in [142] which is an m-dimensional cube covering at least $\frac{3}{4} \times 2^m$ on-set pi-terms. In the b_j-map shown in fig.8.1(a), the largest $\frac{3}{4}$-majority cube is a 2-dimensional cube $\bar{x}_1\bar{x}_0$ covering $\frac{3}{4} \times 2^2 = 3$ pi-terms. Note that algorithmic rules for grouping in a b_j-map, which can be found in [155], are quite different from the counterparts of a traditional Karnaugh map. After this cube is selected, f is decomposed as $f = \bar{x}_1\bar{x}_0 \oplus g$, where g covers both the remaining on-set pi-terms and the off-set pi-terms covered by $\bar{x}_1\bar{x}_0$. Now g has only two on-set pi-terms, as shown in fig.8.1(a), resulting in the mixed polarity expression of f in equation (8.3).

$$f = \bar{x}_1\bar{x}_0 \oplus x_1 \oplus x_2 \tag{8.3}$$

There are two advantages to the $\frac{3}{4}$-majority cubes: (a) It is simple to identify the largest $\frac{3}{4}$-majority cube by computer. (b) The complexity of a function decreases quickly with the selection of the largest $\frac{3}{4}$-majority cube although it does not guarantee the minimality of the result. Every time a $\frac{3}{4}$-majority m-cube is selected, at least $\frac{3}{4} \times 2^m$ on-set pi-terms are deleted and at most $\frac{1}{4} \times 2^m$ off-set pi-terms are added. It is conjectured in [142] that "if off-set pi-terms are incorporated with a $\frac{3}{4}$-majority m-cube to form an m-cube, the total number of groupings will be equal to or less than that resulted from grouping only on-set pi-terms in the $\frac{3}{4}$-majority m-cube". However, this conjecture is not always true. In fig.8.1(b), the same function can be expressed only by two product terms, $f = \bar{x}_2 \oplus \bar{x}_1x_0$. It takes one less product term than the grouping using $\frac{3}{4}$-majority cube in fig.8.1(a). The procedure of the decomposition method consists of the following steps according to [142]:

1. Let the total number of on-set pi-terms of an n-variable function be M.
2. Determine the smallest k such that $k \geq \log_2 M$.
3. Find a prime implicant of order k. Make an arbitrary selection if there is more than one prime implicants of order k.
4. If no prime implicant of order k can be found in step 3, let $m = 1$, and find a $\frac{3}{4}$-majority k-cube with $(2^k - m)$ on-set pi-terms. If there is more than one choice, make an arbitrary selection.
5. If no $\frac{3}{4}$-majority k-cube can be found in step 4, increment m by one and repeat step 4 until a $\frac{3}{4}$-majority k-cube is found.
6. If no $\frac{3}{4}$-majority k-cube can be found in step 5, decrement k by one and repeat step 3 through 5 until a prime implicant or a $\frac{3}{4}$-majority k-cube is found.
7. Obtain the residue function $f_r(x_{n-1}x_{n-2}\cdots x_0)$ by XORing $f(x_{n-1}x_{n-2}\cdots x_0)$ with the prime implicant or $\frac{3}{4}$-majority k-cube selected in step 3 through 5.
8. Repeat step 1 through 7 for the residue functions until a 0-functions is obtained in step 7.

There are several limitations to the above decomposition method.

1. The number of product terms, produced by the heuristic decomposition method, largely depends on the polarity of the initial FPRM for some functions[142]. No exact

polarity minimization algorithm is applied there to reduce the initial complexity of the problem.

2. The identification of the largest $\frac{3}{4}$-majority cube in steps (4) and (5) is inefficient. If there are more than one $\frac{3}{4}$-majority cubes, then an arbitrary selection is made.

3. The algorithm is only suitable for small single output functions, mainly based on tri-state maps[141].

4. The developed program has not been tested with the common benchmarks. Thus no comparison is available to verify the effectiveness of the concept of $\frac{3}{4}$-majority cubes.

8.3 Improved decomposition method for large single output functions

8.3.1 Initial FPRMs with the best polarity

Due to the lack of an efficient approach to find the best polarity for the initial FPRMs in [142], various polarities are tried and then the best expression is obtained by comparing all the available solutions. This manual search method is not practical for large functions. In this chapter, however, an efficient algorithm, which is based on the concept of the polarity for SOP forms proposed in chapter 5, is applied to find the best FPRM expression with the least on-set pi-terms. Although this approach does not necessarily improve the quality of the final result, it can greatly reduce the initial complexity of the problem and improve the speed for decomposition. For example, there are 4 pi-terms for the function shown in fig.8.1(a) with polarity 0. Using the method based on the concept of $\frac{3}{4}$-majority cube, an expression can be obtained for the function, which is shown in equation (8.3). However, starting from an FPRM form with the best polarity 6 as shown in fig.8.1(c), only two on-set pi-terms are necessary to express the same function. From fig.8.1(c), it is easily concluded that both $\bar{x}_2$ and $\bar{x}_1x_0$ are the largest $\frac{3}{4}$-majority cubes. Hence equation (8.4) can be obtained as follows.

$$f = \bar{x}_2 \oplus \bar{x}_1x_0 \tag{8.4}$$

In comparison with equation (8.3), equation (8.4) has less numbers of both product terms and literals, which leads to simpler RMPLA structure.

8.3.2 Identification of the largest $\frac{3}{4}$-majority cubes and prime implicants

In [142], the selection of a $\frac{3}{4}$-majority cube is mainly derived from a tri-state map, which is not suitable for large functions. It is implied in [142] that only the largest $\frac{3}{4}$-majority cube needs to be identified for decomposition. In this chapter, a list of on-set pi-terms is applied for the representation of large functions. Consequently, the selection of the largest $\frac{3}{4}$-majority cube is adapted in procedure 8.1.

Procedure 8.1. *Given a list of M on-set pi-terms for an n-variable Boolean function, $M \geq 1$. Let three variable sets, $\mathbb{S}_0$, $\mathbb{S}_1$, $\mathbb{S}_2$ and an integer l be $\emptyset$, $\emptyset$, $\{x_0, x_1, \cdots x_{n-1}\}$, and $(n-1)$ respectively. Copy the original pi-term list to a temporary list $\mathbb{L}$ and let M' be the number of on-set pi-terms of $\mathbb{L}$. The largest $\frac{3}{4}$-majority cube can be found through the following steps.*

(1) If $M \geq \frac{3}{4} \times 2^n$, return $\bar{x}_{n-1}\bar{x}_{n-2} \cdots \bar{x}_0$ as the largest $\frac{3}{4}$-majority cube. Otherwise, go to the next step.

(2) Let a_{j0} and a_{j1} be the occurring numbers of "0" and "1" respectively in $\mathbb{L}$ for each variable x_j in $\mathbb{S}_2$. Hence $a_{j0} + a_{j1} = M'$. Select the larger number a_j from a_{j0} and a_{j1}. If they are the same, then select a_{j0} to reduce the literal number within a product term.

(3) Determine the largest number among a_js for all the variables in $\mathbb{S}_2$. Suppose this number is a_k which corresponds with variable x_k, $0 \leq k \leq n-1$. If a_k is the number of "0"s, then move x_k from $\mathbb{S}_2$ to $\mathbb{S}_0$, and remove all the on-set pi-terms whose number of variable x_k is 0 from $\mathbb{L}$. Otherwise, if a_k is the number of "1"s, then move x_k from $\mathbb{S}_2$ to $\mathbb{S}_1$, and remove all the on-set pi-terms whose number of variable x_k is 0 from $\mathbb{L}$. Update the number of on-set pi-terms of $\mathbb{L}$, M'.

(4) If $a_k \geq \frac{3}{4} \times 2^l$, then return $\prod_{x_i \in \mathbb{S}_2} \bar{x}_i \cdot \prod_{x_i \in \mathbb{S}_1} x_i$ as the largest $\frac{3}{4}$-majority cube where "$\cdot$" is the AND operation. Otherwise, decrement l by one and go to step (2).

Example 8.1. There are four on-set pi-terms for the 3-variable function f in fig.8.1(a), which can be represented in a list in fig.8.3(a). Thus we have $M = 4$, $\mathbb{S}_0 = \mathbb{S}_1 = \emptyset$, $\mathbb{S}_2 = \{x_2, x_1, x_0\}$, $l = n - 1 = 2$, $\mathbb{L} = \{0, 1, 3, 4\} = \{000, 001, 011, 100\}$ and $M' = 4$.

(1) Because $M \ngeq \frac{3}{4} \times 2^3$, go to step (2) of procedure 8.1.

(2) There are three variables x_0, x_1 and x_2 in $\mathbb{S}_2$. For x_0, there are two *0*s and two *1*s in the on-set pi-terms of $\mathbb{L}$ shown in fig.8.3(a). Hence $a_{00} = a_{01} = 2$. In the same way, we have $a_{10} = a_{20} = 3$, $a_{11} = a_{21} = 1$. Additionally, the larger number a_j is selected between a_{j0} and a_{j1}, $0 \leq j \leq n-1$. Therefore, $a_0 = 2$, and $a_1 = a_2 = 3$ as shown in fig.8.3(a). Then go to step (3) of procedure 8.1.

(3) In step (3) of procedure 8.1, both a_1 and a_2 are the largest numbers. Suppose a_1 is first selected as the largest number. From step (2), it can be seen that a_1 is the number of *0*s, instead of *1*s. Hence x_1 is moved from $\mathbb{S}_2$ to $\mathbb{S}_0$. As a result, $\mathbb{S}_0 = \{x_1\}$, $\mathbb{S}_2 = \{x_0, x_2\}$ and $\mathbb{S}_1 = \emptyset$. Additionally, remove pi-term "011" from $\mathbb{L}$ since the number of x_1 is 1. Now $\mathbb{L} = \{000, 001, 100\} = \{0, 1, 4\}$.

(4) Because $a_1 = 3 \geq \frac{3}{4} \times 2^2$, return $\bar{x}_2\bar{x}_0$ as the largest $\frac{3}{4}$-majority cube as shown in fig8.3(b). Alternatively, if a_2 is selected as the largest number in step (3), then x_2 is moved from $\mathbb{S}_2$ to $\mathbb{S}_0$. Consequently, $\mathbb{S}_0 = \{x_2\}$, $\mathbb{S}_2 = \{x_0, x_1\}$ and $\mathbb{S}_1 = \emptyset$. Therefore another $\frac{3}{4}$-majority cube, $\bar{x}_1\bar{x}_0$ is found which has been shown in fig.8.1(a).

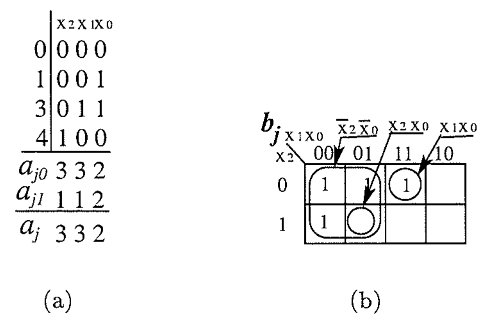


Figure 8.3: Example of procedure 8.1
(a) on-set pi-term list (b) selection of the largest $\frac{3}{4}$-majority cube $\bar{x}_2\bar{x}_0$

Procedure 8.1 can be efficiently implemented in C language using bitwise operations. After the largest $\frac{3}{4}$-majority cube c is identified, an n-variable function f_0 can be decomposed as $f_0 = c \oplus f_1$, where f_1 is an n-variable function. Furthermore, all the on-set pi-terms covered by c are deleted and the off-set pi-terms covered by c are added to f_1. Call procedure 8.1 iteratively to find the largest $\frac{3}{4}$-majority cube for function f_1 until a zero function is obtained. For example, if $\bar{x}_2\bar{x}_0$ is returned as the largest $\frac{3}{4}$-majority cube for the function f_0 in fig.8.3(a), then $f_0 = \bar{x}_2\bar{x}_0 \oplus f_1$ and the on-set pi-terms of f_1 are $\{3, 5\}$. Iteratively call procedure 8.1 for f_1 and get another expression as shown in fig.8.3(b).

$$f_0 = \bar{x}_2\bar{x}_0 \oplus x_1x_0 \oplus x_2x_0 \tag{8.5}$$

From equation (8.5), because $\bar{x}_2\bar{x}_0$ and x_2x_0 have the same set of variables, the expressions produced by the decomposition method belong to the most general class of AND/XOR expressions, namely exclusive-OR sum-of products expressions (ESOPs)[127]. Additionally, in steps (3) and (4) of example 8.1, selection of variable x_1 and x_2 leads to different $\frac{3}{4}$-majority cubes, $\bar{x}_2\bar{x}_0$ and $\bar{x}_1\bar{x}_0$. Comparing equation (8.3) and equation (8.5), it can be seen they have the same number of product terms but different literal numbers. Based on our experiments, the final expression is usually quite sensitive to the selection of a $\frac{3}{4}$-majority cube when there are more than one choices. This problem will be discussed in section 8.3.3.1.

In the decomposition method of [142], there are two loops to select a $\frac{3}{4}$-majority cube. The first loop consists of steps (4) and (5) where the dimension of a cube, k remains fixed. If there is no $\frac{3}{4}$-majority cube, then decrease the dimension k by one in step (6) which constructs the second loop. However in procedure 8.1, only one loop is needed with respect to the dimension of a cube in step (4). There is no loop in step (3), which saves CPU time extensively to decide a $\frac{3}{4}$-majority cube.

In [142], the largest prime implicant is selected prior to the selection of the largest $\frac{3}{4}$-majority cube. Actually, an m-dimensional prime implicant is only a special case of a $\frac{3}{4}$-majority cube, where the number of the covered on-set pi-terms is 2^m. Therefore, there is no difference between the selection of the largest prime implicant and the largest $\frac{3}{4}$-majority cube. Consequently, procedure 8.1 can be applied to decide both the largest prime implicant and the largest $\frac{3}{4}$-majority cube as will be shown in example 8.2.

8.3.3 Further improvements for the decomposition method

8.3.3.1 Order of $\frac{3}{4}$-majority cubes

In example 8.1, there are two largest $\frac{3}{4}$-majority cubes, $\bar{x}_2\bar{x}_0$ and $\bar{x}_1\bar{x}_0$. Each time only one $\frac{3}{4}$-majority cube should be returned to decompose a function. From our experimental results, it is found that the number of product terms largely depends on the selection of a suitable $\frac{3}{4}$-majority cube when there are more than one choices. This problem can be called the determination of the order of $\frac{3}{4}$-majority cubes. In [142], an arbitrary cube is selected due to the lack of an efficient selection strategy. However, a good $\frac{3}{4}$-majority cube can chosen by deleting some on-set pi-terms that are not adjacent to any other on-set pi-terms so that the decomposed function have a good structure. Then update the numbers of *1*s and *0*s and select the largest $\frac{3}{4}$-majority cubes based on the new numbers. If the updated numbers are still the same, then select the default variable with the smallest index. From our experiments, this modification usually leads to better results than the arbitrary selection although it takes more time to decide the adjacency relation among on-set pi-terms in the list. This improved selection will be illustrated in example 8.2.

Example 8.2. Given an on-set pi-term list $\mathbb{L} = \{0, 2, 9, 15\}$ in fig.8.4(a) for a 4-variable function. Following procedure 8.1, we have $\mathbb{S}_0 = \mathbb{S}_1 = \emptyset$, $\mathbb{S}_2 = \{x_0, x_1, x_2, x_3\}$. Besides, $\mathbb{L} = \{0, 2, 9, 15\}$, $l = 4 - 1 = 3$, and $M' = 4$. Because $4 \not\geq \frac{3}{4} \times 2^4$ in step (1), count the occurring number for each variable in $\mathbb{S}_2$ which is shown in fig.8.4(a). In step (3) of procedure 8.1, a_2, which equals to 3, is selected because it is the largest number. Additionally, x_2 is moved from $\mathbb{S}_2$ to $\mathbb{S}_0$ since a_2 is the number of *0*s. Thus $\mathbb{S}_0 = \{x_2\}$, $\mathbb{S}_1 = \emptyset$, $\mathbb{S}_2 = \{x_0, x_1, x_3\}$ and $\mathbb{L} = \{0, 2, 9\}$ as shown in fig8.4.(b). In step (4) of procedure 8.1, since $a_2 \not\geq \frac{3}{4} \times 2^3$, decrease l to 2 and go to step (2) of procedure 8.1. In the same way, count the occurring numbers for three variables in $\mathbb{S}_2$, which are shown in fig.8.4(b). It can be seen that all the three occurring numbers, a_0, a_1, and a_3 are the same. Furthermore, the on-set pi-term $9 = \{1001\}$ is not adjacent to the other two pi-terms. Hence this pi-term can be deleted from the list $\mathbb{L}$, as shown in fig.8.4(c). Update the occurring numbers in fig.8.4(c), $a_0 = a_3 = 2$, $a_1 = 1$. Then a_0 is selected since it is the smallest index. Therefore, $\mathbb{S}_0 = \{x_0, x_2\}$, $\mathbb{S}_1 = \emptyset$, $\mathbb{S}_2 = \{x_1, x_3\}$ and $\mathbb{L} = \{0, 2\}$ as shown in fig.8.4(d). In step (4) of procedure 8.1, since $a_0 \not\geq \frac{3}{4} \times 2^2$, change l to 1 and go to step (2) of procedure 8.1. From fig.8.4(d), a_3 is selected because it is the only largest occurring number. Therefore, $\mathbb{S}_0 = \{x_0, x_2, x_3\}$, $\mathbb{S}_1 = \emptyset$ and $\mathbb{S}_2 = \{x_1\}$ in step (3) of procedure 8.1. Now $a_3 \geq \frac{3}{4} \times 2^1$, so cube $\bar{x}_1$ is returned as the largest $\frac{3}{4}$-majority cube. It can be seen that $\bar{x}_1$ in this case is actually the largest prime implicant.

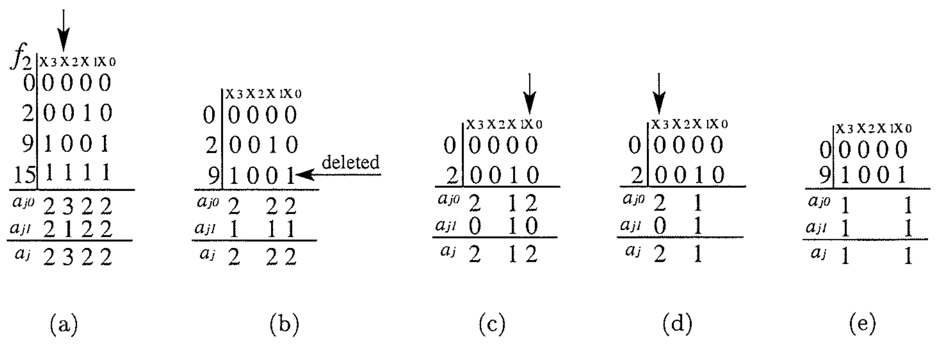


Figure 8.4: Order of $\frac{3}{4}$-majority cubes
(a)original function (b) delete one pi-term that is not adjacent to others (c, d) select a variable with the largest occurring number (e)two pi-terms if x_1 is selected in (b)

In example 8.2, the deletion of pi-term $9 = (1001)_2$ can avoid selecting variable x_1 in fig.8.4(b). Otherwise, the selected $\frac{3}{4}$-majority cubes would actually be pi-terms 1 and x_3x_0, as shown in fig.8.4(e), which needs more product terms to realize the function consequently.

8.3.3.2 One more expansion for decomposition

An n-variable function can be decomposed by Davio expansions with respect to variable x_i as follows.

$$\begin{aligned} f &= f_{x_i=0} \oplus x_i(f_{x_i=0} \oplus f_{x_i=1}) && (8.6)\\ &= f_{x_i=1} \oplus \bar{x}_i(f_{x_i=0} \oplus f_{x_i=1}) && (8.7)\\ &= f_0|_{x_i} \oplus \dot{x}_i f_1|_{x_i} && (8.8) \end{aligned}$$

where both $f_0|_{x_i}$ and $f_1|_{x_i}$ are n-1 variable functions and $\dot{x}_i \in \{x_i, \bar{x}_i\}$. All the functions can be expressed by recursive decomposition of equation (8.8). Equation (8.8) is complete to express any function but does not necessarily produce the simplest solution even with the best polarity and application of the $\frac{3}{4}$-majority cubes. For a class of Boolean functions that satisfy the following condition,

$$f_0|_{x_i} \oplus f_1|_{x_i} = \bar{x}_{n-1}\bar{x}_{n-2}\cdots\bar{x}_{i+1}\bar{x}_{i-1}\cdots\bar{x}_0 \qquad (8.9)$$

the decomposition method in [142] does not necessarily produce "good" results even with the best polarity. Instead, procedure 8.2 can be applied for the decomposition of this class of functions.

Procedure 8.2. *For an n-variable function f that satisfy equation (8.9) with respect to variable x_i, $0 \leq i \leq n-1$, $n > 2$, then f can be expressed by equation (8.12) or (8.15).*

$$\begin{aligned} f &= f_0|_{x_i} \oplus \dot{x}_i f_1|_{x_i} && (8.10) \\ &= f_0|_{x_i} \oplus \dot{x}_i(\bar{x}_{n-1}\bar{x}_{n-2}\cdots\bar{x}_{i+1}\bar{x}_{i-1}\cdots\bar{x}_0 \oplus f_0|_{x_i}) && (8.11) \\ &= \bar{x}_{n-1}\bar{x}_{n-2}\cdots\bar{x}_{i+1}\dot{x}_i\bar{x}_{i-1}\cdots\bar{x}_0 \oplus \bar{\dot{x}}_i f_0|_{x_i} && (8.12) \end{aligned}$$

$$\begin{aligned} f &= f_0|_{x_i} \oplus \dot{x}_i f_1|_{x_i} && (8.13) \\ &= (\bar{x}_{n-1}\bar{x}_{n-2}\cdots\bar{x}_{i+1}\bar{x}_{i-1}\cdots\bar{x}_0 \oplus f_1|_{x_i}) \oplus \dot{x}_i f_1|_{x_i} && (8.14) \\ &= \bar{x}_{n-1}\bar{x}_{n-2}\cdots\bar{x}_{i+1}\bar{x}_{i-1}\cdots\bar{x}_0 \oplus \bar{\dot{x}}_i f_1|_{x_i} && (8.15) \end{aligned}$$

From equation (8.9), the total on-set pi-terms of $f_0|_{x_i}$ and $f_1|_{x_i}$ is 2^{n-1}. To reduce the complexity of the subsequent function, a function with less on-set pi-term is selected between $f_0|_{x_i}$ and $f_1|_{x_i}$. In other words, if $f_0|_{x_i}$ has less on-set pi-terms than $f_1|_{x_i}$, then equation (8.12) is selected to decompose the function. If $f_1|_{x_i}$ has less on-set pi-terms than $f_0|_{x_i}$, then equation (8.15) is selected to decompose the function. If the on-set pi-term number is the same, then equation (8.15) is selected to reduce the literal number since literal $\dot{x}_i$ does not appear in the first product term in equation (8.15). If $n \leq 2$, this new expansion is not applied to save the literal numbers, which can be illustrated in example 8.3.

Example 8.3. A 3-variable function f is shown in fig.8.5(a) both in a b_j-map and a pi-term list with the best polarity. While applying procedure 8.1, it is very difficult to decide the order of $\frac{3}{4}$-majority cubes discussed in section 8.3.3.1. Actually, each individual pi-term in fig.8.5(a) is the largest $\frac{3}{4}$-majority cube. Hence we have

$$f = x_0 \oplus x_1 \oplus x_2 \oplus x_2x_1x_0 \tag{8.16}$$

Using the notations in equation (8.8), we have $f_0|_{x_1} = x_0 \oplus x_2$ and $f_1|_{x_1} = 1 \oplus x_2x_0$. Therefore, $f_0|_{x_1} \oplus f_1|_{x_1} = x_0 \oplus x_2 \oplus 1 \oplus x_2x_0 = \bar{x}_2\bar{x}_0$, and equation (8.9) is satisfied. Because the on-set pi-term numbers of $f_0|_{x_1}$ and $f_1|_{x_1}$ are equal, from procedure 8.2, equation (8.15) is selected to decompose the function as $f = \bar{x}_2\bar{x}_0 \oplus \bar{x}_1 g$, where g is a 2-variable function shown in fig.8.5(b). Note that the b_j-map of g is the right half of the b_j-map shown in fig.8.5(a). Additionally, function g also satisfies equation (8.9). In the same way, we have,

$$g = \bar{x}_2x_0 \oplus \bar{x}_0 \tag{8.17}$$

or

$$g = \bar{x}_2 \oplus x_2\bar{x}_0 \tag{8.18}$$

corresponding with equations (8.12) and (8.15) respectively. If $\frac{3}{4}$-majority cube is used instead, then another expression for g,

$$g = 1 \oplus x_2x_0 \tag{8.19}$$

is obtained since both 1 and x_2x_0 are the largest $\frac{3}{4}$-majority cubes. It can be seen that equation (8.19) is simpler than equation (8.17) or (8.18), which is the result of applying equation (8.12) or (8.15). Hence the new expansion, equation (8.12) or (8.15) will not be applied when $n \leq 2$. As a result, $f = \bar{x}_2\bar{x}_0 \oplus \bar{x}_1 \oplus x_2\bar{x}_1x_0$ which is one product term less than equation (8.16).

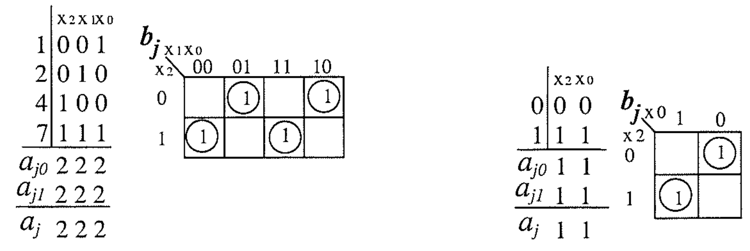


Figure 8.5: Example of procedure 8.2
(a) original function in a pi-term list and a b_j-map (b) decomposed function g using equation (8.15) to expand

8.3.3.3 Even faster decomposition

In step (2) of procedure 8.1, if the number of *0*s or *1*s for a variable x_j, a_j is M, then all the on-set pi-terms are inside either cube $\bar{x}_{n-1}\bar{x}_{n-2}\cdots\bar{x}_{j+1}x_j\bar{x}_{j-1}\cdots\bar{x}_0$ if a_j is the number of *1*s or cube $\bar{x}_{n-1}\bar{x}_{n-2}\cdots\bar{x}_{j+1}\bar{x}_{j-1}\cdots\bar{x}_0$ if a_j is the number of *0*s. In this case, this n-variable function f_0 can be decomposed as in equation (8.20).

$$f_0 = \dot{x}_j f_1 \tag{8.20}$$

where f_1 is a n-1 variable function and $\dot{x}_j$ is x_j if a_j is the number of *1*s for variable x_j; otherwise $\dot{x}_j$ is 1.

Example 8.4. In fig.8.6(a), a 4-variable function f_0 has 4 on-set pi-terms with the best polarity. It can be seen from fig.8.6(a) that a_2 is 4 and it is the number of *1*s for variable

x_2. From equation (8.20), we have $f_0 = x_2 f_1$ where f_1 is a 3-variable function shown in fig.8.6(b). Iteratively call procedure 8.1 and return the largest $\frac{3}{4}$-majority cubes , $\bar{x}_3 x_1 \bar{x}_0$ and $\bar{x}_1 x_0$. Thus,

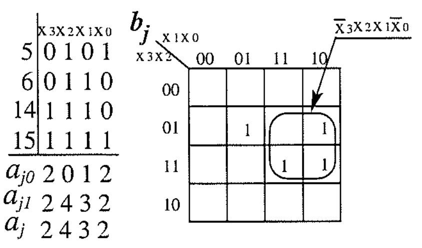
$$f_0 = x_2(\bar{x}_3 x_1 \bar{x}_0 \oplus \bar{x}_1 x_0) = \bar{x}_3 x_2 x_1 \bar{x}_0 \oplus x_2 \bar{x}_1 x_0 \tag{8.21}$$

Alternatively, call procedure 8.2 and find that equation (8.9) is satisfied for function f_1 with respect to variable x_1. Furthermore, equation (8.12) is selected to decompose function f_1,

$$f_1 = \bar{x}_3 x_1 \bar{x}_0 \oplus \bar{x}_1 f_2 \tag{8.22}$$

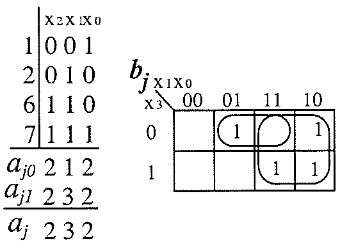
where f_2 is a 2-variable function shown in fig.8.6(c). It is obvious that $f_2 = x_0$. Replace f_2 in equation (8.22) and the same result $f_0 = \bar{x}_3 x_2 x_1 \bar{x}_0 \oplus x_2 \bar{x}_1 x_0$ is obtained as equation (8.21).

Additionally, in fig.8.4(c), after deleting a pi-term 9, the occurring numbers of x_0, x_2, and x_3 are equal to $M = 2$. Hence these three variables can be deleted because they are the numbers of *0*s. Consequently, a product term $\bar{x}_1$ can be returned directly.



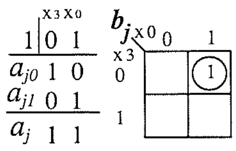
	$x_3x_2x_1x_0$
5	0 1 0 1
6	0 1 1 0
14	1 1 1 0
15	1 1 1 1
a_{j0}	2 0 1 2
a_{j1}	2 4 3 2
a_j	2 4 3 2

(a)



	$x_2x_1x_0$
1	0 0 1
2	0 1 0
6	1 1 0
7	1 1 1
a_{j0}	2 1 2
a_{j1}	2 3 2
a_j	2 3 2

(b)



	x_3x_0
1	0 1
a_{j0}	1 0
a_{j1}	0 1
a_j	1 1

(c)

Figure 8.6: Boolean function of example 8.4
(a) the original function f_0 (b, c) functions f_1 and f_2 respectively after decomposition

8.3.4 Improved decomposition method

Based on the previous improvements to the decomposition method proposed in [142] and procedure 8.1, the following heuristic procedure can be proposed for the mixed polarity

Reed-Muller minimization.

Procedure 8.3. *Given an n-variable Boolean function f_0 in the conventional PLA format based on AND/OR operations. First call the program in* chapter 5 *to convert the function from SOP to FPRM format with polarity 0 and find the best polarity p_{best}. Then call the program again to convert f_0 from SOP to FPRM format with polarity p_{best}. Save all the on-set pi-terms in a list $\mathbb{L}$, and let M be the number of on-set pi-terms. Additionally, let product term number $T = 0$.*

(1) If n or M is zero, then return f_0 as a constant and go to step (7).

(2) If $M \geq \frac{3}{4} \times 2^n$, then cube $\bar{x}_{n-1}\bar{x}_{n-2}\cdots\bar{x}_0$ is the largest $\frac{3}{4}$-majority cube. Therefore, $f_0 = \bar{x}_{n-1}\bar{x}_{n-2}\cdots\bar{x}_0 \oplus f_1$, where f_1 is an n-variable function covering all the off-set pi-terms. These new on-set pi-terms are saved in the list $\mathbb{L}$. Increase T by 1 and change M to $2^n - M$. Replace f_0 with f_1 and go to step (1). Otherwise, go to the next step.

(3) Let a_{j0} and a_{j1} be the occurring numbers of "0" and "1" respectively in $\mathbb{L}$ for each variable x_j, $0 \leq j \leq n-1$. Select the larger number a_j from a_{j0} and a_{j1}. If they are the same, then select a_{j0} to reduce the literal number within a product term.

(4) If there are α variables whose occurring numbers a_j equal to M, $1 \leq \alpha \leq n-1$, then from equation (8.20), f_0 can be decomposed as $f_0 = (\prod_{i=0}^{\alpha-1} \dot{x}_i) f_1$, where f_1 is an $(n-\alpha)$ variable function. Replace f_0 with f_1 and go to step (1). Otherwise, go to the next step.

(5) If there are β variables whose occurring numbers $a_{j0} + a_{j1} = 2^{n-1}$, $1 \leq \beta \leq n-1$, $n > 2$, then from section 8.3.3.2, call procedure 8.2 to check if the function can be decomposed by equations (8.12) or (8.15). If there is one variable x_i that satisfies equation (8.9), then the function is decomposed based on procedure 8.2. Decrease n by one and increase T by one. Update both $\mathbb{L}$, M and go to step (1). Otherwise, go to the next step.

(6) Call procedure 8.1 to find the largest $\frac{3}{4}$-majority cube c. If there are more than one choices, then apply the adjacency relation discussed in section 8.3.3.1 to decide one of them. Consequently, f_0 is decomposed by cube c, $f_0 = c \oplus f_1$, where f_1 is an n-variable function, covering both the remaining on-set pi-terms of f_0 and the off-set pi-term covered by c. Update both $\mathbb{L}$ and M accordingly. Increase T by 1 and go to step (1) for f_1.

(7) Update all the variables according to p_{best}. In other words, if the polarity of variable x is 1, then complement all the occurrences of x in the expression obtained from the previous steps.

Example 8.5. A 5-variable function f_0 with 18 on-set minterms is given in PLA format whose function is $\sum(0, 3, 4, 7, 8, 12, 16, 17, 20-25, 28-31)$. It can be first converted to FPRM format with polarity 0 as $\textstyle\sum\!\!\!\!\!\bigoplus(0-2, 11, 17, 22, 27)$ and the best polarity is 20. Hence convert the function from SOP to FPRM with polarity 20, which is shown in fig.8.7(a) as $f_0 = \textstyle\sum\!\!\!\!\!\bigoplus(0, 6, 17, 18, 22, 27)$. Thus $\mathbb{L} = \{0, 6, 17, 18, 22, 27\}$, $T = 0$ and $M = 6$. Now this function can be simplified following procedure 8.3.

(1) Neither n or M is zero, go to step (2) of procedure 8.3.

(2) $M \ngeq \frac{3}{4} \times 2^5$, go to step (3) of procedure 8.3..

f_0	x_4	$\downarrow$ x_3	x_2	x_1	x_0
0	0	0	0	0	0
6	0	0	1	1	0
17	1	0	0	0	1
18	1	0	0	1	0
22	1	0	1	1	0
27	1	1	0	1	1
a_{j0}	2	5	4	2	4
a_{j1}	4	1	2	4	2
a_j	4	5	4	4	4

(a)

f_1	x_4	x_3	$\downarrow$ x_2	x_1	x_0
0	0	0	0	0	0
17	1	0	0	0	1
27	1	1	0	1	1
2	0	0	0	1	0
a_{j0}	2	3	4	2	2
a_{j1}	2	1	0	2	2
a_j	2	3	4	2	2

(b)

f_2	x_4	$\downarrow$ x_3	x_1	x_0
0	0	0	0	0
9	1	0	0	1
15	1	1	1	1
2	0	0	1	0
a_{j0}	2	3	2	2
a_{j1}	2	1	2	2
a_j	2	3	2	2

(c)

f_3	$\downarrow$ x_4	x_3	x_1	$\downarrow$ x_0
0	1	0	0	1
15	1	1	1	1
a_{j0}	0	1	1	0
a_{j1}	2	1	1	2
a_j	2	1	1	2

(d)

f_4	x_3	x_1
0	0	0
3	1	1
a_{j0}	1	1
a_{j1}	1	1
a_j	1	1

(e)

Figure 8.7: Example for procedure 8.3
(a) original function f_0 with the best polarity (b, c, d, e) functions f_1, f_2, f_3 and f_4 respectively

(3) Count a_{j0} and a_{j1} for all the variables which are shown in fig.8.7(a), $0 \leq j \leq 4$. From fig.8.7(a), we have $a_0 = a_1 = a_2 = a_4 = 4$, and $a_3 = 5$.

(4) No variable whose occurring number a_j equals to M, go to step (5) of procedure 8.3.

(5) Similarly, the condition in step (5) of procedure 8.3 is not satisfied, go to step (6).

(6) Call procedure 8.1 and cube $\bar{x}_4\bar{x}_2x_1$ is returned as the largest $\frac{3}{4}$-majority cube. Hence f_0 is decomposed as $f_0 = \bar{x}_4\bar{x}_2x_1 \oplus f_1$, where f_1 is a 5-variable function shown in fig.8.7(b). Additionally, $T = 1$, $M = 4$, and $\mathbb{L} = \{0, 2, 17, 27\}$ because cube $\bar{x}_4\bar{x}_2x_1$ covers three on-set pi-terms, $6, 18, 22$, and one off-set pi-term 2. Go to step (1) of procedure 8.3 for f_1.

(7) Neither n nor M is zero. Furthermore, $M \ngeq \frac{3}{4} \times 2^5$. Hence count the occurring numbers for these five variables as shown in fig.8.7(b) in step (3) of procedure 8.3. In step (4) of procedure 8.3, the occurring number a_2 of variable x_2 equals to M. From equation (8.20), $f_1 = 1 \cdot f_2 = f_2$, where f_2 is a 4-variable function shown in fig.8.7(c). Thus $T = 1$, $M = 4$, $n = 4$, and go to step (1) of procedure 8.3.

(8) Neither n nor M is zero. Furthermore, $M \ngeq \frac{3}{4} \times 2^4$. Hence count the occurring numbers for each variable which is shown in fig.8.7(c) in step (3) of procedure 8.3. Then a_3 is selected because it is the only largest number 3. No condition in step (4) and (5) of procedure 8.3 is satisfied, so call call procedure 8.1 to find the largest $\frac{3}{4}$-majority cube in step (6) of procedure 8.3. From example 8.2, cube $\bar{x}_1$ is returned as the largest $\frac{3}{4}$-majority cube. Therefore, $f_2 = \bar{x}_1 \oplus f_3$ where f_3 is shown in fig.8.7(d). Additionally, $T = 2$, $M = 2$, and $\mathbb{L} = \{0, 15\}$ since $\bar{x}_1$ covers two pi-terms 0 and 2.

(9) In step (4) of procedure 8.3, the occurring numbers, a_0 and a_4 of variable x_0 and x_4 equal to M. From equation (8.20), $f_3 = x_4x_0f_4$, where f_4 is a 2-variable function shown in fig.8.7(e). Go to step (1) of procedure 8.3 with $\mathbb{L} = \{0, 3\}$ and $n = 2$.

(10) Following the steps in procedure 8.3, it can be seen that $f_4 = 1 \oplus x_3x_1$ with $T = 4$. From the above steps, an expression of the function is obtained.

$$f_0'(\bar{x}_4x_3\bar{x}_2x_1x_0) = \bar{x}_4\bar{x}_2x_1 \oplus \bar{x}_1 \oplus x_4x_0 \oplus x_4x_3x_1x_0 \tag{8.23}$$

(11) In step (7) of procedure 8.3, since $p_{best} = 20 = (10100)$, variable x_2 and x_4 should be complemented in equation (8.23). The final expression of the function is obtained as follows.

$$f_0(x_4x_3x_2x_1x_0) = x_4x_2x_1 \oplus \bar{x}_1 \oplus \bar{x}_4x_0 \oplus \bar{x}_4x_3x_1x_0 \tag{8.24}$$

8.3.5 Functional verification

The output of the program realizing procedure 8.3 is in RMPLA format, where "0", "1", and "-" represents the corresponding variable is missing, positive and complemented respectively. The following procedure is applied for the functional verification.

Procedure 8.4. *Given an FPRM form f for an n-variable function with zero polarity, procedure 8.3 can produce an RMPLA format. This RMPLA output can be functionally verified in the following steps.*

1. Replace each "-" with "0" and "1" so that a product term with i "-"s produces 2^i pi-terms;

2. Delete the generated pi-terms with an even occurrences.

3. Compare the remaining pi-terms with the pi-terms in the original FPRM form f. If they are the same, then the output RMPLA is functionally correct; otherwise, it is wrong.

Example 8.6. Equation (8.24) in example 8.5 can be expressed by an RMPLA format[26], shown in fig.8.8(a). This result can be verified by the following steps.

1. Replace each "-" with "0" and "1" for each product term, which is shown in fig.8.8(b).

2. No duplicate pi-term exists. Thus none of them is deleted. The decimal numbers are also shown in brackets in fig.8.8(b).

3. The pi-terms obtained from fig.8.8(b) are $0-2, 11, 17, 22, 27$, which are the same as the original FPRM form with zero polarity in example 8.5.

8.4 Generalization to multiple output functions

8.4.1 Decomposition method using encoding

For multiple output functions, an important criterion is how best to share the common product terms with several subfunctions. In [120], an n-variable m-output function is simplified by minimizing an *$(m+n)$*-variable single output function denoted as a hyperfunction.

```
.i 5
.o 1
.p 4
1 0 1 1 0  1
0 0 0 – 0  1
– 0 0 0 1  1
– 1 0 1 1  1
```

(a)

```
.i 5
.o 1
.p 7
1 0 1 1 0  1  (22)
0 0 0 0 0  1   (0)
0 0 0 1 0  1   (2)
0 0 0 0 1  1   (1)
1 0 0 0 1  1  (17)
0 1 0 1 1  1  (11)
1 1 0 1 1  1  (27)
```

(b)

Figure 8.8: Example for procedure 8.4
(a) the output RMPLA format of the function in example 8.5 (b) the corresponding on-set pi-terms

This is not suitable for large functions run on a PC. In this chapter, an encoding scheme is applied to merge the multiple outputs into single output by adding $\lceil \log_2^m \rceil$, instead of m extra variables, where $\lceil \log_2^m \rceil$ is the smallest integer equal to or greater than $\log_2^m$. The following algorithm, shown as procedure 8.5, is proposed, through which the problem of sharing the common product terms can be solved implicitly.

Procedure 8.5. *Suppose there is an n-variable m-output Boolean function,* $\{f_{m-1}, f_{m-2}, \cdots, f_0\}$*,* $m \geq 2$*. Then* $\gamma = \lceil \log_2^m \rceil$ *extra variables are used to merge these outputs into a single output function* f *of* $\gamma + n$ *variables. For simplicity, the binary code of* i *is added to the* γ *most significant bits (MSBs) of each pi-term for the jth subfunction* f_j*,* $0 \leq j \leq m-1$*. This transformation can be expressed in equation (8.25).*

$$f = \bigoplus_{j=0}^{m-1} [\dot{x}_{\gamma+n-1}\dot{x}_{\gamma+n-2}\cdots\dot{x}_n]_j f_j \tag{8.25}$$

In equation (8.25), j *can be expressed by a binary* γ*-tuple,* $(j_{\gamma-1}j_{\gamma-2}\cdots j_0)_2$*, and*

$$\dot{x}_i = \begin{cases} 1, & j_{i-n} = 0 \\ x_i, & j_{i-n} = 1 \end{cases}, \quad n \leq i \leq \gamma + n - 1$$

Suppose there is a 4-variable 2-output function, $f_0(x_3, x_2, x_1, x_0)$ and $f_1(x_3, x_2, x_1, x_0)$. An extra variable x_4 can be applied to merge these two outputs into a 5-variable function $f(x_4, x_3, x_2, x_1, x_0)$. From procedure 8.5, "0" is added to the MSB of each pi-term of $f_0(x_3, x_2, x_1, x_0)$ while "1" is added to the MSB of each pi-term of $f_1(x_3, x_2, x_1, x_0)$.

Therefore we have,

$$f(x_4, x_3, x_2, x_1, x_0) = f_0(x_3, x_2, x_1, x_0) \oplus x_4 f_1(x_3, x_2, x_1, x_0) \tag{8.26}$$

After a simplified expression of $f(x_4, x_3, x_2, x_1, x_0)$ is obtained by calling procedure 8.3, theorem 8.1 can be applied to split the expression and generate the representations for each individual subfunction using definition 8.2 of "cover" relation and its properties.

Definition 8.2. A binary relation of "cover", $\mathcal{R}$ on a set $\mathbb{S} = \{0, 1, 2\}$ can be expressed by a set of ordered pairs[121], $\mathcal{R} = \{(0,0), (1,1), (2,0), (2,1)\}$. Additionally, a product term $\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots\ddot{x}_0$ can be represented by a ternary number $a_{n-1}a_{n-2}\cdots a_0$ that satisfies equation (8.27).

$$a_i = \begin{cases} 0, & \ddot{x}_i = 1 \\ 1, & \ddot{x}_i = x_i \\ 2, & \ddot{x}_i = \bar{x}_i \end{cases} \tag{8.27}$$

where $0 \leq i \leq n-1$. A ternary number $a_{n-1}a_{n-2}\cdots a_0$ covers another ternary number $b_{n-1}b_{n-2}\cdots b_0$ if $a_i\mathcal{R}b_i$ for any i, $0 \leq i \leq n-1$. A product term $\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots\ddot{x}_0$ covers another product term $\ddot{y}_{n-1}\ddot{y}_{n-2}\cdots\ddot{y}_0$ if their corresponding ternary number a covers b.

Example 8.7. A 4-variable product term $x_3\bar{x}_2x_1x_0$ can be expressed by a ternary number $(1211)_3$. It covers two ternary numbers, $(1011)_3$ and $(1111)_3$ based on the "cover" relation $\mathcal{R}$ in definition 8.2. Accordingly, the product term $x_3\bar{x}_2x_1x_0$ covers two other product terms $x_3x_1x_0$ and $x_3x_2x_1x_0$. In the same way, a 4-variable product term $\bar{x}_1$ can be represented by a ternary number $(0020)_3$, which covers two other ternary numbers, $(0000)_3$ and $(0010)_3$. In other words, $\bar{x}_1$ covers two product terms, 1 and x_1.

If a product term c_0 covers another product term c_1, then cube c_1 is inside c_0 in the corresponding b_j-map[155]. For example, in fig.8.3(b), product term $\bar{x}_2\bar{x}_0$ covers x_2x_0 based on definition 8.2. It can be seen that cube x_2x_0 is inside $\bar{x}_2\bar{x}_0$. Similarly, cube x_1 is also properly inside cube $\bar{x}_1\bar{x}_0$ in fig.8.1(a) because product term $\bar{x}_1\bar{x}_0$ covers x_1 based on definition 8.2.

Lemma 8.1. *Any ternary number covers at least another ternary number.*

Lemma 8.1 is obvious. If there is no "2" in all the bits of a ternary number, then it covers itself. Otherwise, it at least covers two other ternary numbers based on definition 8.2.

Lemma 8.2. *Any product term, $\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots\ddot{x}_0$, can be represented by XORing all the* *pi-terms it covers. Equivalently,*

$$\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots\ddot{x}_0 = \sum \pi_j \tag{8.28}$$

$$= \sum_{j=0}^{2^{k_i}-1} [\dot{x}_{n-1}\dot{x}_{n-2}\cdots\dot{x}_0]_j \tag{8.29}$$

where k_i is the number of variables whose corresponding ternary number of product term $\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots\ddot{x}_0$ is 2, $k_i \geq 0$. In other words, $[\dot{x}_{n-1}\dot{x}_{n-2}\cdots\dot{x}_0]_j$ is a pi-term covered by $\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots\ddot{x}_0$ based on definition 8.2, $\dot{x}_i \in \{1, x_i\}$, $0 \leq i \leq n-1$, $0 \leq j \leq 2^{k_i}-1$.

Lemma 8.2 is also obvious. In example 8.7, the product term $x_3\bar{x}_2x_1x_0$, whose ternary number is $(1211)_3$, covers two pi-terms, $x_3x_2x_1x_0$ and $x_3x_1x_0$, whose ternary numbers are $(1111)_3$ and $(1011)_3$ respectively. Hence $x_3\bar{x}_2x_1x_0 = \sum_{j=0}^{2^1-1} [\dot{x}_3\dot{x}_2\dot{x}_1\dot{x}_0]_j = x_3x_2x_1x_0 \oplus x_3x_1x_0$. Similarly, $\bar{x}_1 = 1 \oplus x_1$ since $\bar{x}_1$ covers both 1 and x_1.

Theorem 8.1. *Suppose an n-variable m-output Boolean function is merged into a single output function f of $\gamma+n$ variables by procedure 8.5, where $\gamma = \lceil \log_2^m \rceil$, $m \geq 2$. Let f be expressed by T product terms as in equation (8.30).*

$$f(x_{\gamma+n-1}x_{\gamma+n-2}\cdots x_0) = \sum_{i=0}^{T-1} \ddot{x}_{\gamma+n-1}\ddot{x}_{\gamma+n-2}\cdots\ddot{x}_0 \tag{8.30}$$

where $\ddot{x}_i \in \{1, x_i, \bar{x}_i\}$, $0 \leq i \leq \gamma+n-1$. Then any individual subfunction f_j can be represented by a subset of these T product terms in equation (8.31).

$$f_j = \sum_{i=0}^{T-1} [\ddot{x}_{\gamma+n-1}\ddot{x}_{\gamma+n-2}\cdots\ddot{x}_n]_j \cdot \ddot{x}_{n-1}\cdots\ddot{x}_0 \tag{8.31}$$

where "$\cdot$" is the AND operation, j can be expressed by a binary γ-tuple, $(j_{\gamma-1}j_{\gamma-2}\cdots j_0)_2$, $0 \leq j \leq m-1$. In equation (8.31), if the γ MSBs of the ternary number of the product term $\ddot{x}_{\gamma+n-1}\ddot{x}_{\gamma+n-2}\cdots\ddot{x}_0$ covers j, then $[\ddot{x}_{\gamma+n-1}\ddot{x}_{\gamma+n-2}\cdots\ddot{x}_n]_j$ is 1; otherwise it is 0. In other words, for each product term, $\ddot{x}_{\gamma+n-1}\ddot{x}_{\gamma+n-2}\cdots\ddot{x}_0$ in equation (8.30), if its γ MSBs of the corresponding ternary number, defined in definition 8.2, covers j, then product term, $\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots\ddot{x}_0$, is included in equation (8.31). Otherwise, it is not included for f_j. Hence the total product term number to realize these m subfunctions is T, which is the same product term number for the single output function f shown in equation (8.30).

Proof. In equation (8.30), each mixed polarity product term, $\ddot{x}_{\gamma+n-1}\ddot{x}_{\gamma+n-2}\cdots\ddot{x}_0$, can be divided into two parts, $(\ddot{x}_{\gamma+n-1}\ddot{x}_{\gamma+n-2}\cdots\ddot{x}_n) \cdot (\ddot{x}_{n-1}\cdots\ddot{x}_0)$, where "$\cdot$" is the AND operation. The first part $(\ddot{x}_{\gamma+n-1}\ddot{x}_{\gamma+n-2}\cdots\ddot{x}_n)$ can be replaced by all the pi-terms covered by the product term according to lemma 8.2. Let every product term in equation (8.30) be symbolized as $PT_i = [\ddot{x}_{\gamma+n-1}\ddot{x}_{\gamma+n-2}\cdots\ddot{x}_0]_i$, $0 \leq i \leq T-1$. From equations (8.29) and (8.30), we have,

$$\begin{aligned} f(x_{\gamma+n-1}\cdots x_1x_0) &= \bigoplus_{i=0}^{T-1} PT_i && (8.32)\\ &= \bigoplus_{i=0}^{T-1} [\ddot{x}_{\gamma+n-1}\ddot{x}_{\gamma+n-2}\cdots\ddot{x}_0]_i && (8.33)\\ &= \bigoplus_{i=0}^{T-1} [(\ddot{x}_{\gamma+n-1}\ddot{x}_{\gamma+n-2}\cdots\ddot{x}_n)\cdot(\ddot{x}_{n-1}\cdots\ddot{x}_0)]_i && (8.34)\\ &= \bigoplus_{i=0}^{T-1} \left[\left(\bigoplus_{j=0}^{2^{k_i}-1} [\dot{x}_{\gamma+n-1}\dot{x}_{\gamma+n-2}\cdots\dot{x}_n]_j\right)\cdot(\ddot{x}_{n-1}\cdots\ddot{x}_0)\right]_i && (8.35)\\ &= \bigoplus_{j=0}^{2^{k_i}-1}\bigoplus_{i=0}^{T-1} [[\dot{x}_{\gamma+n-1}\dot{x}_{\gamma+n-2}\cdots\dot{x}_n]_j\cdot(\ddot{x}_{n-1}\cdots\ddot{x}_0)]_i && (8.36) \end{aligned}$$

In equation (8.35), k_i is the number of variables whose corresponding ternary number of product term PT_i is "2". Additionally, $[\dot{x}_{\gamma+n-1}\dot{x}_{\gamma+n-2}\cdots\dot{x}_n]_j$ is an pi-term covered by $\ddot{x}_{\gamma+n-1}\ddot{x}_{\gamma+n-2}\cdots\ddot{x}_n$. Compare equations (8.36) and (8.25), it can be concluded that each subfunction f_j can be represented by XORing the product term $\ddot{x}_{n-1}\ddot{x}_{n-2}\cdots\ddot{x}_0$ if the γ MSBs of the corresponding product term, $\ddot{x}_{\gamma+n-1}\ddot{x}_{\gamma+n-2}\cdots\ddot{x}_0$, cover j. Thus equation (8.31) is proved.

From Lemma 8.1, any product term in equation (8.30) can be used by at least one subfunction. Additionally, the product terms of any subfunctions are the subset of the product terms in equation (8.30). Therefore, the total product term number to realize all the subfunctions is the same as the product term number in equation (8.30). □

Example 8.8. Suppose two 4-variable functions, $g_0(x_3x_2x_1x_0)$ and $g_1(x_3x_2x_1x_0)$ are merged into a 5-variable function g by adding an extra variable x_4 so that $g = g_0 \oplus x_4g_1$ based on equation (8.25) in procedure 8.5. Suppose the expression of g is the same as in equation (8.24),

$$g = x_4x_2x_1 \oplus \bar{x}_1 \oplus \bar{x}_4x_0 \oplus \bar{x}_4x_3x_1x_0 \qquad (8.37)$$

These four product terms can be denoted as ternary numbers, namely, $(10110)_3$, $(00020)_3$, $(20001)_3$, and $(21011)_3$. From theorem 8.1, the product terms of subfunction g_0 are the ones whose MSB covers "0" based on the definition 8.2. Because both "0" and "2" cover "0", there are three product terms $(00020)_3$, $(20001)_3$, and $(21011)_3$ whose MSB cover "0".

Therefore, g_0 can be realized by XORing $(0020)_3$, $(0001)_3$, and $(1011)_3$. Thus we have,

$$g_0 = \bar{x}_1 \oplus x_0 \oplus x_3 x_1 x_0 \tag{8.38}$$

Similarly, the product terms of subfunction g_1 are the ones whose MSB covers "1" . From definition 8.2, there are three product terms, $(10110)_3$, $(20001)_3$, and $(21011)_3$ that satisfy this condition. Therefore g_1 can be expressed in the following equation.

$$g_1 = x_2 x_1 \oplus x_0 \oplus x_3 x_1 x_0 \tag{8.39}$$

From equations (8.38) and (8.39), two product terms, x_0 and $x_3x_1x_0$, are shared by these two subfunctions. These two subfunctions, g_0 and g_1, can be realized by 4 product terms, that is the same product term number for function g. It is worthwhile to note that g_1 is different from the cofactor of g with respect to x_4, $g_{x_4=1} = x_2x_1 \oplus \bar{x}_1$.

Suppose g is obtained by merging four 3-variable functions, $g_0^{'}(x_2x_1x_0)$, $g_1^{'}(x_2x_1x_0)$, $g_2^{'}(x_2x_1x_0)$ and $g_3^{'}(x_2x_1x_0)$. From theorem 8.1, the product terms of subfunction $g_0^{'}$ are the ones whose two MSBs cover "00". From definition 8.2, there are two product terms, $(00020)_3$ and $(20001)_3$, that satisfy this condition. Hence $g_0^{'} = \bar{x}_1 \oplus x_0$. In the same way, $g_1^{'} = x_3x_1x_0$ because it is only this product term whose two MSBs cover "01". Similarly, we have $g_2^{'} = x_2x_1 \oplus x_0$ and $g_3^{'} = x_1x_0$.

From theorem 8.1 and example 8.8, it can be concluded that the more occurrences of the complemented form of extra variables, the more product terms are shared by subfunctions.

Based on the previous discussion, the following procedure can be presented for mixed polarity Reed-Muller minimization.

Procedure 8.6. *Given an n-variable m-output Boolean function f in the conventional PLA format. First call the program in* chapter 5 *to convert it from SOP to FPRM format with polarity 0 and find the best polarity p_{best} for function f. Then call the program again to convert it from SOP to FPRM format with polarity p_{best}. Suppose all the on-set pi-terms and output parts are saved in lists $\mathbb{L}_1$ and $\mathbb{L}_2$. Let M be the number of on-set pi-terms in $\mathbb{L}_1$ and product term number $T = 0$.*

(1) Call procedure 8.5 to merge the multiple outputs into single output by adding extra variables.

(2) Call procedure 8.3 to simplify this single output function in the Reed-Muller domain directly, which excludes conversion and best polarity algorithms in procedure 8.3. Return an expression for f and the product term number T.

(3) Apply theorem 8.1 to split the expression and produce the result for each subfunctions.

(4) Call procedure 8.4 to verify the result.

8.4.2 Very large functions

For very large multiple output functions whose variable number is usually more than 25, the encoding approach in section 8.4.1 is not suitable because there are usually too many input variables for the merged single output function. For example, a 26-variable 50-output function would be merged into a 32-variable function, which is not practical to run on a common PC. However, it is observed that there are many redundant variables for each individual function for these very large functions. In other words, each subfunction is usually dependent on a small number of these inputs. Therefore, each subfunction can first be simplified individually using procedure 8.3. Then merge the common product terms to reduce the product term number. Although this strategy is simple, it is very efficient based on our experimental results which are shown in the next section.

8.5 Experimental results

Procedure 8.6 has been implemented in C language. The input to the program is the conventional PLA format which is based on AND/OR operations. The program produces the RMPLA result of the function. This program is compiled by the GNU Compiler Collection egcs-1.1.2 and tested using MCNC and IWLS'93 benchmarks on a PC with Pentium-266 CPU and 64M RAM under Linux operating system. The comparisons with ESPRESSO[26] are shown in tables 8.1 and 8.2, where "i/o" means the numbers of inputs and outputs respectively. In the column of "Reed-Muller Domain", the time to obtain the FPRM with the best polarity from the conventional SOP input is shown as "FPRM". The product term number and CPU time of the mixed polarity optimization are shown in the column of "Decomp. Method" as "term#" and "time" respectively. All the results have been verified using procedure 8.4. It can be seen from table 8.1 that the decomposition method can produce much less product terms than ESPRESSO for most testcases. For instance, "t481" needs 481 product terms to be implemented based on AND/OR operations while only 13 mixed polarity terms are sufficient based on AND/XOR operations. Due to the conventional SOP input format, most of the CPU time is actually spent on the FPRM forms.

For very large multiple output functions, an alternative scheme, which is discussed in section 8.4.2, is applied. Instead of using encoding method to merge the multiple outputs into single output, the decomposition method is first applied for each individual function, then the common product terms are merged. It can be seen that for most testcases in table 8.2, the decomposition method produces much better results than ESPRESSO with respect to product term numbers.

		Reed-Muller Domain			Boolean Domain	
testcase	i/o	time(s)	Decomp. Method		ESPRESSO	
		FPRM	term#	time(s)	term#	time(s)
5xp1	7/10	0.06	63	0.03	65	0.07
cm150a	21/1	512.35	17	0.02	17	0.01
cmb	16/4	17.65	5	0.01	15	0.07
f51m	8/8	0.12	51	0.01	76	0.06
mux	21/1	287.76	16	0.01	16	0.01
pcle	19/9	263.00	26	0.03	45	0.03
pm1	16/13	34.18	25	0.03	28	0.05
rd53	5/3	0.05	20	0.01	31	0.01
rd73	7/3	0.08	63	0.02	127	0.09
rd84	8/4	0.16	107	0.03	255	0.35
t481	16/1	9.25	13	0.01	481	0.62
tcon	17/16	56.22	25	0.04	24	0.01

Table 8.1: Comparison with ESPRESSO for general multiple output functions

		Reed-Muller Domain			Boolean Domain	
testcase	i/o	time(s)	Decomp. Method		ESPRESSO	
		FPRM	term#	time(s)	term#	time(s)
apex6	135/99	3162.56	491	0.06	656	43.12
b9	41/21	5.12	119	0.01	106	0.38
c8	28/18	0.47	52	0.01	79	0.13
cht	47/36	0.03	81	0.01	81	0.08
count	35/16	583.81	64	0.01	169	0.31
example2	85/66	7.23	234	0.01	329	1.53
i6	138/67	0.03	239	0.03	202	0.37
i7	199/67	0.05	268	0.01	264	0.78
pcler8	27/17	2.67	40	0.01	53	0.20
unreg	36/16	0.03	48	0.01	48	0.03
x3	135/99	4554.53	536	0.09	656	46.97
x4	94/71	14.79	374	0.05	520	5.49

Table 8.2: Comparison with ESPRESSO for very large multiple output functions

8.6 Summary

Although there has been extensive research on AND/XOR forms, applications of Reed-Muller logic have not become popular due to lack of efficient algorithms for mixed polarity minimization. In this chapter, an improved decomposition method is developed based on the concept of $\frac{3}{4}$-majority cubes[142] and top-down approach. Using the fast algorithms for fast conversion between SOP and FPRM formats in chapter 5 and the polarity optimization which is based on the concept of the polarity for SOP forms, the decomposition method is generalized to very large multiple output Boolean functions. Although the problem of Reed-

Muller logic minimization is much more difficult than SOPs, the improved decomposition method can produce much better results, which is consistent with the conclusion in [126]. Therefore the developed program offers an important opportunity for the practical RMPLA applications for very large multiple output functions.

Chapter 9

Conclusions and Future Work

The aim of the project is to develop various algorithms for logic synthesis for both the standard Boolean logic and Reed-Muller logic, which are based on AND/OR operations and AND/XOR operations respectively. The main contributions of this thesis can be summarized as follows.

☛ In chapter 3, the concept of two-level cube on a Karnaugh map is first generalized to a multilevel cube that can cover both "0" and "1" entries. Then an efficient procedure is presented to produce a multilevel form of any incompletely specified function on a Karnaugh map utilizing don't cares (DCs) generated as the function is decomposed. Further, an important property of Boolean functions, "containment" of cofactors is introduced so that new DCs, which are called functional DCs, can be generated to simplify large functions. The functional DCs are on function/logic level while satisfiability don't cares (SDCs) and observability don't cares (ODCs) are on circuit/network level. An algorithm based on the concept of multilevel cube and the property of "containment" is developed and implemented in C language. From the experimental results, more than 70% of the splits have the containment property for all the single output functions in the common benchmarks. The developed program can produce better results than script.rugged of SIS, both in area and speed for a number of testcases.

☛ The above mentioned algorithm is improved with respect to variable order and splitting equation problems, then generalized to multiple outputs using the encoding method in chapter 4. This encoding strategy provides a new approach to simplify Boolean relations. Experimental results show that functional DCs which are based on the concept of containment are very effective not only for single output, but also for multiple output Boolean functions.

☛ In chapter 5, the concept of polarity for canonical sum-of-products (SOP) Boolean functions is introduced. This facilitates efficient conversion between SOP and fixed polarity Reed-Muller (FPRM) forms. New algorithms are presented for the bidirectional conversion between the two paradigms. Multiple segment and multiple pointer techniques are employed to achieve fast conversion for large Boolean functions. Experimental results show that the algorithm is very efficient in terms of time and space for large Boolean functions. The algorithm is tested for randomly generated functions of up to 30 variables and 500, 000 on-set coefficients.

☛ In chapter 6, a fast algorithm is proposed to convert from SOPs to FPRM forms without generating disjoint cube covers or functional decision diagrams. This procedure is based on the property of input redundancy and is tailored for very large multiple output Boolean functions. Test result shows that it only takes about 0.1 seconds to convert a function with 199 inputs and 67 outputs run on a common personal computer.

☛ The properties of the polarity for canonical sum-of-products expressions of Boolean functions are further examined and formalized in chapter 7. A transform matrix is developed to convert SOP expressions from one polarity to any other polarity. It is shown that the effect of SOP polarity is to reorder the on-set minterms of a Boolean function. Based on these properties, we achieve the transform matrix for fixed polarity Reed-Muller expressions for the conversion between two different polarities. Comparison of these two matrices shows that the Reed-Muller transform matrix has much more complex structure. Besides, the best polarity of FPRM forms with the least on-set terms corresponds with the polarity of SOP forms with the best "order" of the on-set minterms. Applying these features of the transform matrix, a fast algorithm is presented to obtain the best polarity for multiple output completely specified Boolean functions. The advantage of the algorithm is achieved by fast binary search strategy, instead of iterative AND or XOR operations. Furthermore, the speed of the algorithm does not depend on the number of outputs because the output part is taken collectively as a normal integer. Test results for benchmark examples of up to 25 inputs and 29 outputs are given run on a common personal computer. Previously it has been generally accepted that exact minimization of fixed polarity Reed-Muller forms is only suitable when the number of input variables is less than 15.

☛ In chapter 8, decomposition techniques are utilized for mixed polarity Reed-Muller minimization, which lead to Reed-Muller programmable logic array implementations for Boolean functions. The proposed algorithm produces simplified mixed polarity Reed-Muller format from the conventional sum-of-products input based on top-down strategy. The output format belongs to the most general class of AND/XOR forms, namely exclusive-OR sum-of-products. This method is further generalized to very large multiple output Boolean functions. The developed decomposition method is implemented in C language and tested with MCNC and IWLS'93 benchmarks. Experimental results show that the decomposition method can produce much better results than ESPRESSO for many testcases. This efficient method offers compact Reed-Muller programmable logic array implementations to add to their inherent advantage of easy testability compared to the conventional programmable logic array realizations.

The above work can be further generalized and improved along the following lines.

☛ The encoding problem described in section 4.4.2 can be solved with the help of the similar problem arisen in the traditional functional decomposition[104]. The solution may be applied to the minimization of Boolean relations[28] as slightly covered in section 4.4.1.

☛ The idea of functional don't cares can be incorporated in other logic minimizers, such as SIS, to improve their performance.

☛ The conversion algorithm for single output functions, presented in chapter 5, has been successfully generalized to very large multiple output Boolean functions in chapter 6 based on redundancy removal strategy. Accordingly, the fast algorithm for exact polarity minimization in chapter 7 can be applied for very large multiple output Boolean functions based on the same strategy.

☛ The mixed polarity optimization method in chapter 8 can be utilized for multilevel Reed-Muller minimization based on decomposition scheme. Little work has been done in this area[128].

☛ The above mentioned methods for Reed-Muller optimization can be further generalized to incompletely specified Boolean functions.

Publications

The following list shows the papers published or submitted after April, 1998.

1. **L. Wang,** A. E. A. Almaini , Fast Conversion Algorithm for Very Large Boolean Functions, **Electronics Letters,** Vol.36, No.16, 1370-1371, 2000

2. **L. Wang,** A. E. A. Almaini, and A. Bystrov, Efficient Polarity Conversion for Large Boolean Functions, **IEE Proceedings Computers and Digital Techniques,** Vol.146, No.4, 197-204, 1999

3. **L. Wang,** X. Chen, and A. E. A. Almaini, Algebraic Properties of Multiple-Valued Modulo Systems and Their Applications to Current-Mode CMOS Circuits, **IEE Proceedings Computers and Digital Techniques,** Vol.145, No.5, 364-368, 1998

4. **L. Wang,** X. Chen, and A. E. A. Almaini, Modulo Correlativity and its Application in a Multiple Valued Logic System, **International J. Electronics,** Vol.85, No.5, 561-570, 1998

5. X. Wu, M. Pedram, and **L. Wang**, Multi-Code State Assignment for Low Power Sequential Circuit Design, **IEE Proceedings Circuits, Devices and Systems,** Vol.147, No.5, 271-275, 2000

6. **L. Wang** and A. E. A. Almaini, Fast Algorithm for Exact Minimization of Large Boolean Functions (submitted), **IEEE Trans. on Computers,** 2000

7. **L. Wang** and A. E. A. Almaini, Multilevel Logic Minimization Using Functional Don't Cares (accepted), **IEEE Proceedings, International Conference on VLSI Design,** Bangalore, India, 2001

8. **L. Wang** and A. E. A. Almaini, Multilevel Logic Simplification for Multiple Output Boolean Functions Using Functional Don't Cares (to be submitted), 2001

9. **L. Wang** and A. E. A. Almaini, Optimization of Reed-Muller PLA Implementations (submitted), **IEE Proceedings Circuits, Devices and Systems,** 2000

References and Bibliography

[1] Abouzeid, P., Besson, T., Sakouti, K., Saucier, G., Gaume, F., and Roane, R., Experimental Results on the Impact of Factorization and Technology Independent Mapping Options on Multilevel Synthesis, Euro ASIC'92, Los Alamitos, CA, 402-403, 1992

[2] Almaini, A. E. A., Electronic Logic Systems, 3rd ed., Prentice Hall, Londoen, 1994

[3] Almaini, A. E. A., A semicustom IC for generating optimum generalized Reed-Muller expansions, Microelectronics Journal, Vol.28, No.2, 129-142, 1997

[4] Almaini, A. E. A. and McKenzie, L., Tabular techniques for generating Kronecker expansions, IEE Proc. Comput. Digit. Tech., Vol.143, No.4, 205-212, 1996

[5] Almaini, A. E. A. and Ping, S., Algorithm for Reed-Muller expansions of Boolean functions and optimization of fixed polarities, The fourth IEEE International Conference on Electronics, Circuits and Systems, Cairo, 148-153, December 1997

[6] Almaini, A. E. A. Thomson, P. and Hanson, D., Tabular techniques for Reed-Muller logic, Int. J. Electronics, Vol.70, No.1, 23-34, 1991

[7] Almaini, A. E. A., and Woodward, An Approach to the Control Variable Selection Problem for Universal Logic Modules, Digital Process, Vol.3, 189-206, 1977

[8] Almaini, A. E. A., Zhuang, N., and Bourset, F., Minimisation of Multioutput Binary Decision Diagrams Using Hybrid Genetic Algorithms, IEE Electronic Letters, Vol.31, No.20, 1722-1723, 1995

[9] Almaini, A. E. A. and Zhuang, N., Using Genetic Algorithms for the Variable Ordering of Reed-Muller Binary Decision Diagrams, Microelectronic Journal, Vol.24, No.4, 471-480, 1995

[10] Almaini, A. E. A. and Zhuang, N., Variable Ordering of BDDs for Multioutput Boolean Functions Using Evolutionary Techniques, Fourth IEEE Int. Conference on Electronics, Circuits & Systems, ICECS'97, Cairo, 148-153, 1997

[11] Almaini, A. E. A., Zhuang, N., and Bourset, F., Minimisation of Multioutput Binary Decision Diagrams Using Hybrid Genetic Algorithms, IEE Electronic Letters, Vol.31, No.20, 1722-1723, 1995

[12] Akers, S. B., Binary Decision Diagram, IEEE Transactions on Computers, Vol.C-27, No.6, 509-516, 1978

[13] Arensman, R., Squeeze Play, Electronic Business, http://www.eb-mag.com/, June, 1998

[14] Awalt, R. K., Making the ASIC/FPGA Decision, Integrated System Design, http://www.isdmag.com/, July, 1999

[15] Ayres, R. F., VLSI Silicon Compilation and the Art of Automatic Microchip Design, Prentice-Hall, Inc., New Jersey, 1983

[16] Aziz, A., Balarin, F., Brayton, R., and Sangiovanni-Vincentelli, A., Sequential Synthesis Using S1S, ICCAD, California, 612-617, 1995

[17] Bartlett, K. A. Brayton, R. K. Hachtel, G. D. Jacoby, R. M. Morrison, C. R. Rudell, R. L. Sangiovanni-Vincentelli, A. and Wang, A. R., Multilevel Logic Minimization Using Implicit Don't Cares, IEEE Tansactions on Computer-Aided Design, Vol.7, No.6, 723-740, 1988

[18] Bartlett, K., Cohen, W., Genus, A. D., and Hachtel, G., Synthesis and Optimization of Multilevel Logic under Timing Constraints, IEEE Transactions on Computer-Aided Design, Vol.CAD-5, No.4, 582-596, 1986

[19] Benini, L., and Micheli, G. D., System-Level Power Optimization: Techniques and Tools, ACM Transactions on Design Automation of Electronic Systems, Vol.5, No.2, 115-192, 2000

[20] Bergamaschi, R. A., Brand, D., and Stok, L., Efficient Use of Large Don't Cares in High-Level and Logic Synthesis, ICCAD, CA, 272-278, 1995

[21] Berman L. and Trevillyan, L., Global Flow Optimization in Automatic Logic Design, IEEE Transactions on CAD/ICAS, Vol.CAD-10, No.5, 557-564, 1991

[22] Boole, G., An Investigation of the Laws of Thought on Which Are Founded the Mathematical Theories of Logic and Probabilities, Dover Publications, Inc., New York, 1854(reprint in 1958)

[23] Brand, D., Bergamaschi, R. A., and Stok, L., Don't Cares in Synthesis: Theoretical Pitfalls and Practical Solutions, IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, Vol.17, No.4, 285-305, 1998

[24] Brayton, R. K. and McMullen, C., The Decomposition and Factorization of Boolean Expressions, Proceedings of the International Symposium on Circuits and Systems, 49-54, Rome, May 1982

[25] Brayton, R. K., Hachtel, G. D., and Sangiovanni-Vincentelli, A. L., Multilevel Logic Synthesis, Proceedings of IEEE, Vol.78, No.2, 264-300, 1990

[26] Brayton, R. K., Hachtel, G. D. McMullen, C. T. and Sangiovanni-Vincentelli, A. L., Logic Minimization Algorithms for VLSI Synthesis, Boston, Kluwer Academic Publishers, 1984

[27] Brayton R. K., Rudell Richard, Sangiovanni-Vincentelli Alberto, and Wang ALbert R., MIS: A Multiple-Level Logic Optimization System, IEEE Transactions on Computer-Aided Design, Vol.CAD-6, No.6, 1062-1081, 1987

[28] Brayton, R. K., and Somenzi, F., Boolean Relations, MCNC/ACM, IWLS'89, May 23-26, North Carolina, 1-9, 1989

[29] Brown, S. D., Francis, R. J., Rose, J., and Vranesic, Z. G., Field-Programmable Gate Arrays, Kluwer Academic Publishers, Boston, 1992

[30] Bryant, R. E., Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification, ICCAD, California, 236-243, 1995

[31] Bryant, R. E., Graph-Based Algorithms for Boolean Function Manipulation, IEEE Transactions on Computers, Vol.C-35, No.8, 677-691, 1986

[32] Bystrov, A., Almaini, A. E. A., Reversed ROBDD Circuits, IEE Electronic Letters, Vol.34, No.15, 1447-1449, 1998

[33] Bystrov, A., Almaini, A. E. A., Testability and Test Compaction for Decision Diagram Circuits, IEE Proc.-Circuits, Devices and Systems, Vol.146, No.4, 153-158, 1999

[34] Cabodi, G., Quer, S., Meinel, C., Sack, H., Slobodova, A., and Stangier, C., Binary Decision Diagrams and the Multiple Variable Order Problem, IEEE/ACM IWLS'98, CA, 346-352, 1998

[35] Chang, C.-H., and Falkowski, B, J., Adaptive Exact Optimisation of Minimally Testable FPRM Expansions, IEE Proc. Comput. Digit. Tech., Vol.145, No.6, 385-394, 1998

[36] Chang, C-L, and Lee, R. C-T., Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York, 1973

[37] Chang, S-C, and Cheng, D. I., Efficient Boolean Division and Substitution, Proc. DAC'98, California, 342-347, 1998

[38] Chang, S-C., Marek-Sadowska, M., Cheng, K-T, An Efficient Algorithm for Local Don't Care Sets Calculation, 32nd DAC'95, CA., 663-667, 1995

[39] Chang, S. C., Marek-Sadowska, M., and Cheng, K. T., Perturb and Simplify: Multi-level Boolean Network Optimizer, IEEE Trans. on Computer Aided Design, Vol.15, No.12, 1494-1504, 1996

[40] Clarke, E. M., Fujita, M., and Zhao, X., Hybrid Decision Diagrams, ICCAD, California, 159-163, 1995

[41] Cong, J., Ding, Y., Combinational Logic Synthesis for LUT Based Field Programmable Gate Arrays, ACM Trans. Design Automation of Electronic Systems, Vol.1, No.2, 145-204, 1996

[42] Coudert, O., Two-Level Logic Minimization: An Overview, Integration, (VLSI journal), Vol.17, No.2, 97-140, 1994

[43] Curtis, H. A., A New Approach to the Design of Switching Circuits, D. Van Nostrand Company, Princeton, N. J., 1962

[44] Damarla, T., and Karpovsky, M., Detection of Stuck-at and Bridging Faults in Reed-Muller Canonical (RMC) Networks, IEE Proc. Pt.E, Vol.136, No.5, 430-433, 1989

[45] Damiani, M., and Micheli, G. De, Don't Care Set Specifications in Combinational and Synchronous Logic Circuits, IEEE Trans. on Computer-Aided Design, Vol.CAD-12, No.3, 365-388, 1993

[46] Damm. C., How Much EXOR improves on OR? Technical Reports, Forschungsbericht Nr. 93-07, 1-6, 1993

[47] Darringer J. A., Brand D., Gerbi, J. V. Joyner William H., Jr and Trevillyan, L., LSS: A System for Production Logic Synthesis, IBM J. Res. Develop. Vol.28, No.5, 537-545, 1984

[48] Darringer J. A., Joyner William H., Jr, Berman C. L, and Trevillyan, L., Logic Synthesis Through Local Transformations, IBM J. Res. Develop. Vol.25, No.4, 272-280, 1981

[49] Debnath, D., and Sasao, T., GRMIN2: A Heuristic Simplification Algorithm for Generalised Reed-Muller Expressions, IEE Proc. Comput. Digit. Tech., Vol.143, No.6, 376-384, 1996

[50] Devadas S., Ghosh A., and Keutzer K., Logic Synthesis, McGraw-Hill, Inc., New York, 1994

[51] Devadas, S., Wang, A. R., Newton, A. R., and Sangiovanni-Vincentelli, A., Boolean Decomposition in Multilevel Logic Optimization, IEEE Journal of Solid State Circuits, Vol.24, No.2, 399-408, 1989

[52] Drechsler, R., Becker, B., Overview of Decision Diagrams, IEE Proc. Comput. Digit. Tech., Vol.144, No.3, 187-193, 1997

[53] Drechsler, R., and Becker, B., Relation Between OFDDs and FPRMs, Electronics Letters, Vol.32, No.21, 1975-1976, 1996

[54] Drechsler, R., Becker, B., and Jahnke, A., On Variable Ordering and Decomposition Type Choice in OKFDDs, IEEE Trans. Computers, Vol.47, No.12, 1398-1403, 1998

[55] Drechsler, R., Drechsler, N., Gunther, W., Fast Exact Minimization of BDDs, DAC'98, California, 200-205, 1998

[56] Drechsler, R., Theobald, M., and Becker, B., Fast OFDD-Based Minimization of Fixed Polarity Reed-Muller Expressions, IEEE Trans. Computers, Vol.45, No.11, 1294-1299, 1996

[57] Dunne, P. E., The complexity of Boolean Networks, Academic Press Ltd, London, 1988

[58] Electronic Design Interchange Format, Electronic Industries Alliance, http://www.edif.org/

[59] Edwards M. D., Automatic Logic Synthesis Techniques for Digital Systems, Macmillan Press Ltd, Hampshire, 1992

[60] Falkowski, B. J., and Chang, C-H., Generalised k-variable-mixed-polarity Reed-Muller Expansions for System of Boolean Functions and Their Minimisation, IEE Proc.-Circuits, Devices and Systems, Vol.147, No.4, 201-210, 2000

[61] Falkowski, B. J., and Perkowski, M. A., Algorithm for the Generation of Disjoint Cubes for completely and incompletely Specified Boolean Functions, Int. J. Electronics, Vol.70, N0.3, 533-538, 1991

[62] Frank, M. P, Knight, T., Margolus, N., Reversibility in optimal scalable computer architectures, Proceedings of the First International Conference on Unconventional Models of Computation, Springer, 165-182, 1998

[63] Friedman, A. D., and Premachandran, R. M., Theory & Design of Switching Circuits, Bell Telephone Laboratories, Inc. and Computer Science Press, California, 1975

[64] Fujita, M., and Matsunaga, Y., Variable Ordering of Binary Decision Diagrams for Multi-Level Logic Minimization, FUJITSU Sci. Tech. J., Vol.29, No.2, 137-145, 1993

[65] Gajski, D. and Kuhn, R., Guest Editors' Introduction – New VLSI Tools, IEEE Computer, Vol.16, No.12, 11-14, 1983

[66] Geus, A.J., and Gregory, D. J., The Socrates Logic Synthesis and Optimization System, Design Systems for VLSI Circuits, Martinus Nijhoff Publishers, 473-498, 1987

[67] Green, D. H., Dual Forms of Reed-Muller Expansions, IEE Proc.-Comput. Digit. Tech., Vol.141, No.3, 184-192, 1994

[68] Green, D., Modern Logic Design, Addison-Wesley Publishing Company, Workingham, England, 1986

[69] Green, D. H., Reed-Muller Expansions with Fixed and Mixed Polarities Over GF(4), IEE Proc. Pt.E., Vol.137, No.5, 380-388, 1990

[70] Green, D. H., Reed-Muller Variable-Entered Vectors and Maps, Int. J. Electronics, Vol.78, No.1, 161-186, 1995

[71] Green, D. H., Simplification of Switching Functions Using Variable-Entered Maps, Int. J. Electronics, Vol.75, No.5, 877-886, 1993

[72] Guan, Z., A Study of Arithmetic Circuits and the Effect of Utilising Reed-Muller Techniques, PhD Thesis, Napier University, 1995

[73] Habib, M. K., Boolean Matrix Representation for the conversion of minterms to Reed-Muller Coefficients and the minimization of Exclusive-OR Switching Functions, Int. J. Electron., Vol.68, No.4, 493-506, 1990

[74] Hachtel, G. D. and Somenzi, F., Logic Synthesis and Verification Algorithms, Kluwer Academic Publishers, Boston, 1996

[75] Hamaguchi, K., Morita, A., Yajima, S., Efficient Construction of Binary Moment Diagrams for Verifying Arithmetic Circuits, ICCAD, 78-82, California, 1995

[76] Hasegawa, T. and McNally, Tell Me Again–What Does the "S" in SOC Stand For?, Integrated System Design, http://www.isdmag.com/, July, 1999

[77] Hohn, F. E., Applied Boolean Algebra, The Macmillan Company, New York, 1966

[78] Hong, S. J., Cain, R. G., Ostapko, D. L., MINI: A heuristic approach for logic minimization, IBM J. of Res. and Dev., Vol. 18, 443-458, 1974

[79] Huntington, E. V., Sets of Independent Postulates for the Algebra of Logic, Trans. Am. Math. Soc., Vol.5, 294-309, July 1904

[80] Ishiura, N., Synthesis of Multilevel Logic Circuits from Binary Decision Diagram, IEICE Trans. Inf. & Syst., Vol.E76-D, No.9, 1085-1092, 1993

[81] Ishiura, N., Sawada, H., and Yajima, S., Minimization of Binary Decision Diagrams Based on Exchanges of Variables, ICCAD, Santa Clara, CA, 472-475, 1991

[82] Jiang, H., and Majithia, J. C., Suggestion for a New Representation for Binary Function, IEEE Trans. Computers, Vol.45, No.12, 1445-1449, 1996

[83] Jones, H. and Harper, C. editors, Handbook of Components for Electronics, 2nd Edition, McGraw-Hill, Inc., New York, 1996

[84] Karnaugh, M., The Map Method for Synthesis of Combinational Login Circuits, Transactions of the AIEE, Vol.72, Pt.1, 593-598, 1953

[85] Khan, Md. M. H. A. Alam, Md. S., Mapping of Fixed Polarity Reed-Muller Coefficients from Minterms and the Minimisation of Fixed Polarity Reed-Muller expressions, Int. J. Electronics, Vol.83, No.2, 235-247, 1997

[86] Kohavi, Z., Switching and Finite Automata Theory, 2nd ed., New York, McGraw-Hill, Inc., 1978

[87] Kravets, V. N., and Sakallah, K. A., M32: A Constructive Multilevel Logic Synthesis System, ACM/IEEE DAC, San Francisco, CA, 336-341, 1998

[88] Kunz, W., and Stoffel, D., Reasoning in Boolean Networks: Logic Synthesis and Verification Using Testing Techniques, Kluwer Academic Publishers, Boston, 1997

[89] Lai, Y-T., Pedram, M., and Vrudhula, S. B. K., Formal Verification Using Edge-Valued Binary Decision Diagrams, IEEE Trans. Computers, Vol.45, No.2, 247-255, 1996

[90] Lawler E. L., An Approach to Multilevel Boolean Minimization, Journal of the Association for Computing Machinery, Vol.11, No.3, 283-295, 1964

[91] Lee, C. Y., Representation of Switching Circuits by Binary-Decision Programs, Bell Syst. Tech. J., Vol.38, No.4, 985-999, 1959

[92] Loureiro, G. V., Digital Systems Design for Testability Based on a Reed-Muller Tree-Circuit Approach, PhD Thesis, University of Manchester Institute of Science and Technology, 1993

[93] Luccio, F., and Pagli, L., On a New Boolean Function with Applications, IEEE Trans. Computers, Vol.48, No.3, 296-310, 1999

[94] Lui, P. K., and Muzio, J. C., Boolean Matrix Transforms for the Minimization of Modulo-2 Canonical Expansions, IEEE Trans. Computers, Vol.41, No.3, 342-347, 1992

[95] Lynch, E. P., Applied Symbolic Logic, John Wiley & Sons, New York, 1980

[96] McCluskey E., Minimization of Boolean Functions, Bell System Technical Journal, Vol.35, No.5, 1417-1444, 1956

[97] McKenzie, L. Almaini, A. E. A. Miller, J. F. and Thomson P., Optimization of Reed-Muller logic functions, Int. J. Electronics, Vol.75, No.3, 451-466, 1993

[98] McKenzie, L., and A. E. A. Almaini, Generating Kronecker Expansions from Reduced Boolean Forms Using Tabular Methods, Int. J. Electronics, Vol.82, No.4, 313-325, 1997

[99] Mendelson, E., Boolean Algebra and Switching Circuits, McGraw-Hill Book Company, New York, 1970

[100] Micheli G. D., Synthesis and Optimization of Digital Circuits, McGraw-Hill, Inc., New York, 1994

[101] Moore, G. E., An Update on Moore's Law, http://developer.intel.com/, Intel Developer Forum Keynote, San Francisco, 1997

[102] Moret Bernard M. E., Decision Trees and Diagrams, Computing Surveys, Vol.14, No.4, 593-623, 1982

[103] Muller, D. E., Application of Boolean Algebra to Switching Circuits Design and to Error Detection, IRE Trans. Electron. Comput., Vol.EC-3, 6-12, 1954

[104] Murgai, R., Brayton, R. K., and Sangiovanni-Vincentelli, A., Optimum Functional Decomposition Using Encoding, 31st DAC, San Diego, 408-414, 1994

[105] Muroga, S., Kambayashi, Y., Lai, H. C., Culliney, J. N., The Transduction Method–Design of Logic Networks Based on Permissible Functions, IEEE Trans. Computers, Vol.38, No.10, 1404-1423, 1989

[106] Muroga, S., Logic Synthesizers, The Transduction Method and Its Extension, SYLON, in the book of "Logic Synthesis and Optimization", edited by Sasao, T., Kluwer Academic Publishers, Boston, 59-86, 1993

[107] Narayan, A., Jain, J., Fujita, M., and Sangiovanni-Vincentelli, Partitioned ROBDDs - A Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions, IEEE/ACM Proceedings, ICCAD, San Jose, CA, 547-554, 1996

[108] Oldfield, J. V., Dorf, R. C., Field Programmable Gate Arrays, John Wiley & Sons, Inc., New York, 1995

[109] Oliveira, A. L., Carloni, L. P., Villa, T., and Sangiovanni-Vincentelli, A. L., Exact Minimization of Binary Decision Diagrams Using Implicit Techniques, IEEE Trans. Computers, Vol.47, No.11, 1282-1296, 1998

[110] Panda, S., and Somenzi, F., Who Are the Variables in Your Neighbourhood, ICCAD, California, 74-77, 1995

[111] Parrilla, L., Ortega, J., and Lloris, A., Nondeterministic AND-EXOR Minimisation by Using Rewrite Rules and Simulated Annealing, IEE Proc. Comput. Digit. Tech., Vol.146, No.1, 1-8, 1999

[112] Perkowski, M., and Grygiel, S., A Survey of Literature on Function Decomposition, Version IV, Portland State University, Portland, November, 1995

[113] Perry, D. L., VHDL, 3rd edition, McGraw-Hill Inc., New York, 1998

[114] Pitty, E. B., and Salmon, J. V., Input Irredundancy of Mixed-Polarity Reed-Muller Equations, Electronics Letters, Vol.24, No.5, 258-260, 1988

[115] Preas, B. and Lorenzetti, M., Physical Design Automation of VLSI systems, Benjamin Cummings, CA., 1988

[116] Purwar, S., An Efficient Method of Computing Generalized Reed-Muller Expansions from Binary Decision Diagram, IEEE Trans. Computers, Vol.40, No.11, 1298-1301, 1991

[117] Quine, W. V., The Problem of Simplifying Truth Functions, American Mathematics Monthly, Vol.59, No.8, 521-531, 1952

[118] Reddy, S. M., Easily Testable Realization for Logic Functions, IEEE Trans. Computers, Vol.C-21, No.11, 1183-1188, 1972

[119] Reed, I. S., A Class of Multiple-Error-Correcting Codes and the Decoding Scheme, IRE Trans. Infomation Theory, Vol.PGIT-4, 38-49, 1954

[120] Riege, M. W., and Besslich, Ph. W., Low-Complexity Synsthesis of Incompletely Specified Multiple-Output Mod-2 Sums, IEE Proc.-E, Vol.139, No.4, 355-362, 1992

[121] Rosen, K. H., Discrete Mathematics and Its Applications, 3rd ed., McGraw-Hill, Inc., New York, 1994

[122] Rudell, R., Dynamic Variable Ordering for Ordered Binary Decision Diagrams, ICCAD, 42-47, 1993

[123] Sait, S. M., VLSI Physical Design Automation Theory and Practice, McGraw-Hill, London, 1995

[124] Salcic, Z. and Smailagic, A., Digital Systems Design and Prototyping Using Field Programmable Logic, Kluwer Academic Publishers, Boston, 1997

[125] Saldanha, A. Wang A. R. and Brayton R. K., Multi-Level Logic Simplication Using Don't Cares and Filters, 26th ACM/IEEE Design Automation COmference, 277-282, 1989

[126] Sasao, T., and Besslich, P., On the Complexity of MOD-2 Sum PLA's, IEEE Trans. on Computers, Vol.39, No.2, 262-266, 1990

[127] Sasao, T. and Izuhara, F., Exact Minimization of FPRMs Using Multi-Terminal EXOR TDDs, pp. 191-210, in Sasao, T. and Fujita, M., (editors), Representations of Discrete Functions, Kluwer Academic Publishers, Boston, 1996

[128] Saul, J., An Algorithm for the Multi-level Minimization of Reed-Muller Representations, Int. Conf. on Computer Design, 634-637, Cambridge, Mass, USA, 1991

[129] Savage, J. E., The Complexity of Computing, John Wiley and Sons, 1976

[130] Savoj. H. and Brayton, R. K., The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks, Proc. DAC, 297-301, 1990

[131] Schulz, S. E., Focus Report: Logic Synthesis and Silicon Compilation Tools, Integrated System Design, http://www.isdmag.com/, May, 1996

[132] Schwarz, A. F., Handbook of VLSI Chip Design and Expert Systems, Academic Press Ltd., London, 1993

[133] Schulz, S. E., A Strategy for Linux EDA Success, Integrated System Design, http://www.isdmag.com/, March, 1999

[134] Sentovich, E. M., Singh, K. J., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P. R., Brayton, R. K., and Sangiovanni-Vincentelli A., SIS: A System for Sequential Circuit Synthesis, Electronics Research Laboratory, Memorandum No. UCB/ERL M92/41, UC, Berkeley, May, 1992

[135] Shannon, C. E., A Symbolic Analysis of Relay and Switching Circuits, Trans. AIEE, Vol.57, 713-723, December 1938

[136] Shiraishi, M.(EURMS), Elements of the Reformed Theory of Logic, (E-Book), LOREIN, Japan, 1997

[137] Song, N., and Perkowski, M. A., New Fast Approach to Approximate ESOP Minimization for Incompletely Specified Multi-Output Functions, Proc. Reed-Muller'97 Conference, Oxford Univ., UK, 61-72, 1997

[138] Tan, E. C. and Yang, H., Fast Tabular Technique for Fixed-Polarity Reed-Muller Logic with Inherent Parallel Process, Int. J. Electronics, Vol.85, No.4, 511-520, 1998

[139] The Programmable Logic Data Book, Xilinx Inc., CA, 1994

[140] Thomas, D. E., The Verilog Hardware Description Language, Fourth Edition, Kluwer academic publishers, Lowell, MA, 1998

[141] Tran, A., Tri-State Map for the Minimisation of Exclusive-OR Switching Functions, IEE Proc. Vol.136, Pt.E, No.1, 16-21, 1989

[142] Tran, A., and Wang, J., Decomposition Method for Minimisation of Reed-Muller Polynomials in Mixed-Polarity, IEE Proc.-E, Vol.140, No.1, 65-68, 1993

[143] Tsai, C-C, and Marek-Sadowska, M, Boolean Functions Classification via Fixed Polarity Reed-Muller Forms, IEEE Trans. Computers, Vol.46, No.2, 173-186, 1997

[144] Tsai, C-C, and Marek-Sadowska, M, Boolean Matching Using Generalized Reed-Muller Forms, 31st ACM/IEEE DAC, 339-344, 1994

[145] Tsai, C-C, and Marek-Sadowska, M, Generalized Reed-Muller Forms as a Tool to Detect Symmetries, IEEE Trans. Comput., Vol.45, No.1, 33-40, 1996

[146] Tsai, C-C, and Marek-Sadowska, M, Minimisation of Fixed-Polarity AND/XOR Canonical Networks, IEE Proc. Comput. Digit. Tech., Vol.141, No.6, 369-374, 1994

[147] Villa, T., Encoding Problems in Logic Synthesis, PhD Thesis, Electrical Engineering and Computer Sciences, Univ. of California at Berkeley, 1995

[148] Vinnakota, B., and Rao, V. V. B., Generation of All Reed-Muller Expansions of a Switching Function, IEEE Trans. Computers, Vol.43, No.1, 122-124, 1994

[149] Walker, R. A. and Thomas, D. E., A Model of Design Representation and Synthesis, Proc. ACM/IEEE 22nd Design Automation Conference, Las Vegas, Nevada, 453-459, 1985

[150] Wang, L., and Almaini, A. E. A., Fast Algorithm for Exact Minimization of Large Boolean Functions (submitted), IEEE Trans. on Computers, 2000

[151] Wang, L., and Almaini, A. E. A., Fast Conversion Algorithm for Very Large Boolean Functions, Electronics Letters, Vol.36, No.16, 1370-1371, 2000

[152] Wang, L., and Almaini, A. E. A., Multilevel Logic Minimization Using Functional Don't Cares, IEEE Proceedings, 14th International Conference on VLSI Design (accepted), Bangalore, India, 2001

[153] Wang, L., Almaini, A. E. A., and Bystrov, A., Efficient Polarity Conversion for Large Boolean Functions, IEE Proc.-Comput. Digit. Tech., Vol.146, No.4, 197-204, 1999

[154] Wu, H., Perkowaki, M. A., Zeng, X., and Zhuang, N., Generalized Partially-Mixed-Polarity Reed-Muller Expansion and Its Fast Computation, IEEE Trans. Computers, Vol.45, No.9, 1084-1088, 1996

[155] Wu, X. Chen, X. and Hurst, S. L., Mapping of Reed-Muller coefficients and the minimization of exclusive-OR switching functions, IEE Proc. E, Vol.129, No.1, 15-20, 1982

[156] Xu, L., Almaini, A. E., A., Miller, J. F., and McKenzie, L., Reed-Muller Universal Logic Module Networks, IEE Proc.-E., Vol.140, No.2, 105-108, 1993

[157] Yang, H., Tan, E. C., Optimization of Multi-Output Fixed-Polarity Reed-Muller Circuits Using the Genetic Algorithm, Int. J. Electronics, Vol.86, No.6, 663-670, 1999

[158] Yang, S., and Ciesielski, M., Optimum and Suboptimum Algorithms for Input Encoding and Its Relationship to Logic Minimization, IEEE Trans. on Computer Aided Design, Vol.10, No.1, 4-12, 1991

[159] Yeh, F.-M., and Kuo, S.-Y., Variable Ordering for Ordered Binary Decision Diagrams by a Divide-and-Conquer Approach, IEE Comput. Digit. Tech., Vol.144, No.5, 261-266, 1997

[160] Zemva, A., Brglez, F., Zajc, B., Multi-level Logic Optimization Based on Wave Synthesis of Permissible Mutation Functions(WASP), IEEE/ACM, IWLS'98, CA, 1998

[161] Zemva, A., Trost, A., and Zajc, B., Multi-level Logic Optimisation Based on Permissible Perturbations, IEE Proc.-Comput. Digit. Tech., Vol.147, No.2, 53-58, 2000

Disk Containing the Software

The attached floppy disk contains the programs developed in the previous chapters. Please read the ***ReadMe.txt*** file for more information.