A Study of Arithmetic Circuits and the Effect of Utilising Reed-Muller Techniques

Zhigang Guan

A thesis submitted in partial fulfilment of the requirements of Napier University for the degree of Doctor of Philosophy in the Department of Electrical, Electronic & Computer Engineering

September 1995

CONTAINS PULLOUTS

Abstract

Reed-Muller algebraic techniques, as an alternative means in logic design, became more attractive recently, because of their compact representations of logic functions and yielding of easily testable circuits. It is claimed by some researchers that Reed-Muller algebraic techniques are particularly suitable for arithmetic circuits. In fact, no practical application in this field can be found in the open literature.

This project investigates existing Reed-Muller algebraic techniques and explores their application in arithmetic circuits. The work described in this thesis is concerned with practical applications in arithmetic circuits, especially for minimizing logic circuits at the transistor level. These results are compared with those obtained using the conventional Boolean algebraic techniques. This work is also related to wider fields, from logic level design to layout level design in CMOS circuits, the current leading technology in VLSI. The emphasis is put on circuit level (transistor level) design. The results show that, although Boolean logic is believed to be a more general tool in logic design, it is not the best tool in all situations. Reed-Muller logic can generate good results which can't be easily obtained by using Boolean logic.

For testing purposes, a gate fault model is often used in the conventional implementation of Reed-Muller logic, which leads to Reed-Muller logic being restricted to using a small gate set. This usually leads to generating more complex circuits. When a cell fault model, which is more suitable for regular and iterative circuits, such as arithmetic circuits, is used instead of the gate fault model in Reed-Muller logic, a wider gate set can be employed to realize Reed-Muller functions. As a result, many circuits designed using Reed-Muller logic can be comparable to that designed using Boolean logic. This conclusion is demonstrated by testing many randomly generated functions.

The main aim of this project is to develop arithmetic circuits for practical application. A number of practical arithmetic circuits are reported. The first one is a carry chain adder. Utilising the CMOS circuit characteristics, a simple and high speed carry chain is constructed to perform the carry operation. The proposed carry chain adder can be reconstructed to form a fast carry skip adder, and it is also found to be a good application for residue number adders. An algorithm for an on-line adder and its implementation are also developed. Another circuit is a parallel multiplier based on 5:3 counter. The simulations show that the proposed circuits are better than many previous designs, in terms of the number of transistors and speed. In addition, a 4:2 compressor for a carry free adder is investigated. It is shown that the two main schemes to construct the 4:2 compressor have a unified structure. A variant of the Baugh and Wooley algorithm is also studied and generalized in this work.

Acknowledgements

I would like to express my gratitude to Prof. Almaini for his good guidance, patience, understanding, friendship and encouragement throughout this work.
I would also like to express my gratitude for the studentship offered to me by the Department of Electrical, Electronic and Computer Engineering, Napier University.
I would like to thank Mr. P. Thomson for his guidance and kindly help during the first stage of this project. I also offer my sincere thanks to my former colleagues Ms. L. Xu, Ms. L. McKenzie, and Mr. N. Zhuang. Our good-natured discussions and their advice are invaluable. Thanks are also due to the staff members in Dept of EECE, Napier University, for the generous support given to me.
Finally, this thesis is dedicated to my wife Qiao Zhang, my daughter Xiaoqiao Guan, and my parents Yanjun Guan and Zhifang Wang, I could not have completed this thesis without their unending support and help.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application of another degree or qualification of this or any other university or institution of learning.

Zhigang Guan

List of Abbreviations

BSD	Binary Signed-Digit
CFA	Carry Free Adder
CMOS	Complementary MOS
CPA	Carry Propagation Adder
DAG	Directed Acyclic Graph
ECAD	Electronic CAD
ESOP	Exclusive or SOP
EXOR	EXclusive OR
FA	Full Adder
FPGA	Field Programmable Gate Array
IC	Integrated Circuit
KRM	Kronecker Reed-Muller
LCA	Logic Cell Array
lsb	least significant bit
LSI	Large scale integration
MOS	Metal Oxide Silicon
msb	most significant bit
NMOS	N channel MOS
PLA	Programmable Logic Array
PMOS	P channel MOS
POS	Product Of Sums
PP(s)	Partial Product(s)
PPG	PP Generator
RBA	Redundant Binary Adder
RM	Reed-Muller
RNS	Residue Number System
SD	Signed-Digit
SOG	Sea Of Gates
SOP	Sum Of Products
VLSI	Very Large Scale Integration
XOR	eXclusive OR
XPLA	XOR PLA

List of Accompanying Material

1). Z. Guan, A. E. A. Almaini, and P. Thomson,"A simple and high speed CMOS carry chain adder architecture", *Int. J. Electronics*, Vol. 75, No. 4, Oct. 1993, pp. 743-752.

2). Z. Guan,"Logic Realization Using Mixed Representations Based on Reed-Muller Forms", *Proc. IEE Colloquium on Synthesis and Optimization of Logic Systems*, London, March 14, 1994, pp. 2/1-2/4.

3). Z. Guan, P. Thomson, and A. E. A. Almaini,"A Parallel CMOS 2's Complement Multiplier Based on 5:3 Counter", *Proc. IEEE Int. Conf. Comput. Design*, Boston, Oct. 10-12, 1994, pp. 298-301.

4). Z. Guan, and A.E.A. Almaini,"One Bit Adder Design Based on Reed-Muller Expansions", *Int. J. Electronics*, to be published.

Contents

Chapter 1: Introduction

Chapter 2: Reed-Muller Logic

Chapter 3: CMOS Implementation of Logic Circuits

Chapter 4: Number Systems and Two Operand Adders

Chapter 5: Carry Free Adders and Parallel Multipliers

Chapter 6: Conclusion

Chapter 1

Introduction

1.1 Reed-Muller Logic

Reed-Muller logic is an algebraic technique for logic circuit design based on AND and Exclusive OR (Modulo 2 arithmetic) operations. There is no universal definition and name for Reed-Muller logic. In the literature, it can be found that many different names are used, such as exclusive OR-switching function[Wu 82], EXOR logic[Besslich 83], Reed-Muller expansion of Boolean functions[Harking 90], Reed-Muller logic[Almaini 91], Reed-Muller algebraic techniques[Green 91B], Reed-Muller representation (expansion, form) of Boolean logic[Saul 92], modulo-2 expressions[Lui 92], AND-EXOR expression[Sasao 93B], Reed-Muller polynomial[Tran 93A], etc.. All these names are often used to describe the same thing, that is, at first, a logic function is represented in exclusive OR sum of products instead of inclusive OR sum of products, and then, the function is minimized by employing certain rules based on two basic operations, AND and Exclusive OR.

In this thesis, Reed-Muller logic, Reed-Muller expansion, Reed-Muller function, the Reed-Muller domain, etc. are employed. In contrast, Boolean logic, Boolean expansion, Boolean function, the Boolean domain, etc. are used to describe the traditional Boolean logic design where a function is represented in inclusive OR sum of products.

This technique was first introduced by Zhegalkin in 1927. He described a special kind of Exclusive Sum of Products (ESOP), which includes only uncomplemented Boolean variables. Later, Reed[Reed 54] and Muller[Muller 54] used ESOP for logic circuit design and error detection, since this time ESOP expressions have been called Reed-

Muller expressions (expansions, representations, forms, etc.) in the literature because Zhegalkin's work, written in Russian, was unknown[Steinbach 93].

Reed-Muller logic, as an alternative means in logic circuit design, can be employed to describe an arbitrary switching function completely. Systematic methods to simplify a logic function in the Reed-Muller domain have been developed[Muller 54, Wu 82, Green 86, 91B, Tran 87, 89, Almaini 91]. In the Boolean domain, a group of coefficients for a given logic function in its canonical form have certain physical meanings. Normally, "0" indicates that the output of a circuit is low, and "1" indicates that the output of a circuit is high. The coefficients directly correspond to a truth table that defines the function completely. In the Reed-Muller domain, the coefficients for a given logic function in its canonical form do not have certain physical meanings. In general, a logic function in the Reed-Muller domain is derived from a corresponding function in the Boolean domain via coefficient conversion.

For a long time, It had been conjectured that the realization of a class of logic functions in Reed-Muller expansions was more economical than the conventional Sum of Products (SOP), its counterpart in the Boolean domain. Later, this conjecture is proved mainly in Sasao and Besslich's work[Sasao 90, Sasao 93A, Sasao 93B]. In their work, many statistical data are used to show that two level Reed-Muller expansions, in general, require fewer products to represent a given logic function than SOP, the conventional two level Boolean expansion. These functions include arithmetic functions, randomly generated functions, symmetric functions, etc.. Consequently, a logic function in Reed-Muller expansion with less products can often generate a more economical circuit.

A lot of precious work about Reed-Muller logic, especially in theory, has been done in the past. Some results may be feasible for commercial products. The first automatic logic synthesis system to make use of the mixed-polarity Reed-Muller expansion is GATEMAP[Pitty 88, Salmon 89]. GATEMAP concurrently maintains three different representations of each logic function throughout all stages of its operation. These three representations are: 1). sum-of-products of the function; 2). sum-of-products of the inverse of the function; 3). mixed-polarity of the function. Finally, the best one is chosen to implement the function. A wide variety of functions are tested by GATEMAP. Reed-Muller equations are found to be normally of comparable size to the sum-of-products, with exceptions being where the Reed-Muller equations are significantly smaller.

Another main advantage of Reed-Muller logic considered by many researchers is that, when a logic circuit is realized with Reed-Muller expansion in two level fixed polarity form, it requires a short test set to detect a single stuck-at fault (stuck-at 0 and stuck-at 1) in the circuit, and the test set is independent of its actual function being

realized[Reddy 72]. Based on the same principle of Reddy's, the function-independent test set for detecting single stuck-at fault and single bridging fault (AND and OR bridging faults) is explored[Bhattacharya 85, Damarla 89].

Although Reed-Muller logic has the two obvious advantages over the conventional Boolean logic, it is still unpopular compared with the applications in the Boolean Domain. Many researchers believed that the main reason for this is lack of efficient algorithms for its minimization[Besslich 83, Saul 90, Sarabi 92]. Therefore, most of the previous work done has been devoted mainly to focus on various algorithms to minimize a given logic function in the Reed-Muller domain and convert between the two domains in the most efficient way[Wu 82, Besslich 83, Green 96, 91B, Tran 87, 89, Helliwell 88, Saul 90, 91, 92, Almaini 91, Sasao 93A, B, McKenzie 93].

1.2 Arithmetic Circuits

Since the inception of digital computers, much effort has been directed towards the search for faster and simpler arithmetic techniques. Because of the high hardware cost, the earlier computers consisted only of simple and economical arithmetic circuits, such as adder, multiplier, etc.. Many complex arithmetic operations were accomplished by software.

With the advance of microelectronics, especially in VLSI technology, many complex arithmetic circuits have become feasible and common. These arithmetic circuits consist not only of dividers, square roots, matrix multipliers and trigonometric processors for general purpose or scientific calculation, but also of some special arithmetic processors for digital signal processing, such as, convolvers and FFT processors.

The earlier approach to designing arithmetic devices concentrated mainly on simple and high-speed circuits. With the development of VLSI technology, in order to deal with the increasing complexity of design, fabrication and test, regularity, modularity, regular and local connection are also emphasized. This is because, a regular and modular circuit not only is easily realized in VLSI, but also benefits testing. In addition, a regular and modular circuit can be easily pipelined, which will increase the throughput of a system in overlapped fashion.

The previous study of many researchers shows that, a lot of complex arithmetic operations, in practice, can be decomposed into two simple and basic operations, addition and multiplication [Hwang 79, Urquhart 84, Joseph 84, Scott 85, Koren 93]. Based on this, arithmetic circuit study may be classified into two categories. One is to attempt to study basic arithmetic circuits, such as adders and multipliers, and

minimize them not only at a higher level, but also at a lower level, such as logic level, circuit level, and layout level. This study is closely related to a logic design tool, for example, Boolean logic or Reed-Muller logic. The other is to attempt to investigate some complex arithmetic circuits and minimize them at a higher level, such as system level or architectural level, in which it is assumed that the basic components, adders, multipliers, and registers, etc., have been minimized. This study is rarely related to the logic design tool. Therefore, the implementation of many complex arithmetic processors is influenced directly by the performance of the adder and the multiplier. Owing to this reason, addition and multiplication have been the most widely studied, and many practical algorithms and implementations for various adders and multipliers have been presented.

It should be mentioned that subtraction and division also are very important basic operations. Subtraction can be achieved by addition in 2's complement number without any additional hardware. Division is often studied at a higher level, rather than at logic level or circuit level, this is because division is inherently composed of a sequential series of addition and subtraction. Consequently, the implementation of a divider, ultimately, depends heavily on the adder. In other words, compared with other arithmetic circuits, studying the adder and multiplier not only is emphasized at a higher level based on number systems, mathematical rules, etc., but also is emphasized at logic level and circuit level for their actual implementations.

Addition and multiplication also can be classified into fixed-point operation and floating-point operation. Float-point operation is more complex than fixed-point operation, but it is often studied at a higher level rather than at logic level and its implementation is still based on the implementation of fixed-point operation. Therefore, in this project, only fixed-point adders and multipliers are discussed, because they are more closely related to logic and circuit optimization.

In addition, only binary logic will be considered in this work. Although multi-valued logic has many theoretical advantages, its practical application is widely limited because of the shortage of reliable and stable basic components. Also, many algorithms in binary logic can be steadily developed into their counterparts in multi-valued logic.

1.3 Objective

"It is well-known that many useful circuits such as arithmetic units and parity checkers are heavily XOR oriented and it is more economical to implement their modulo-2 expressions"[Lui 92, Helliwell 88, Perkowski 89, 90, Saul 91, 92, 93,

Sarabi 92, Csanky 93, Lester 93]. In fact, except for modulo 2 sum, few successful examples can be found relating to practical arithmetic circuits. One of the main reasons for this may be that, in most of the previous work about Reed-Muller logic, the minimization of functions is carried out without regard to target technology. Because the implementation complexity of Reed-Muller logic differs from that of Boolean logic, it is difficult to estimate and compare the results of Reed-Muller logic with that of Boolean logic accurately if only the number of products (literals) is employed. Also, arithmetic functions are established based on PLA or ROM implementation, and there is no architectural or structural design when using these functions. In this way, the results can not be easily generalized and therefore, they are not often employed in practical applications.

In this project, logic circuit optimization using Reed-Muller techniques is explored, and compared with Boolean techniques in MOS circuits, especially in CMOS circuits. The comparison is carried out mainly at the circuit level (transistor level), instead of the gate level, and the results are measured by the number of transistors. This makes the comparison more practical in actual designs. That is, one important difference from most previous work is that, it is desirable to know the minimum circuits more than the most compact logic functions in this work.

The use of Reed-Muller techniques for the synthesis of arithmetic circuits is studied. The circuits consist mainly of adders and parallel multipliers which are more closely related to circuit optimization, and the results are compared with that based on Boolean logic. This helped to gain greater insight into the subject of arithmetic operations, to evaluate the suitability of Reed-Muller techniques, and ultimately to develop more efficient arithmetic circuits.

1.4 Thesis Outline

Chapter 2 reviews the background theory of Reed-Muller logic that is used in later chapters. Representation of Reed-Muller logic for logic functions is first introduced. Secondly, the basic operations of Reed-Muller logic are described. Finally, some algorithms to minimize Reed-Muller functions are studied.

Chapter 3 describes basic circuits in CMOS. In order to compare Reed-Muller logic with Boolean logic in CMOS circuits, *switching network theory* is first introduced. Reed-Muller functions and Boolean functions realized in static CMOS circuit style are studied and compared. This study is explored from the gate level to the layout level,

but mainly at the transistor level. The main result about testing Reed-Muller circuits is reviewed. The possibility to minimize Reed-Muller circuits is investigated.

Chapter 4 first introduces some commonly used number systems and reviews the previous work about two operand adders, and then the design methodology for a general arithmetic circuit is discussed. Later, the possibility of using Reed-Muller logic to improve the carry lookahead circuit, as an example, is investigated. A simple and fast CMOS carry chain adder architecture is presented. This design can be reconstructed for the carry-skip adder, and is also developed for residue number adders. Finally, an algorithm for the on-line adder and its implementation are proposed.

Chapter 5 commences with a review of the main results of multipliers proposed previously by other researchers. Parallel multipliers in the review are emphasized. A general structure for parallel multipliers is described and carry free adders are discussed. Two main types of the most widely used carry free adder modules, the redundant binary adder and the 5:3 counter, are investigated. A unified structure for these two modules is explored. Comparison and evaluation based on a survey of the literature is presented. Finally, a parallel multiplier based on the 5:3 counter is proposed. A variant of the Baugh and Wooley algorithm is generalized in the proposed design.

Chapter 6 concludes the thesis by summarising the main findings and contributions. Some related issues to be further explored for future study are suggested.

Chapter 2

Reed-Muller Logic

2.1 Introduction

Reed-Muller (RM) logic, as an alternative means for logic design, is based on two basic operations in modulo 2 arithmetic: modulo 2 addition and modulo 2 multiplication. These two operations are identical to Boolean Exclusive-OR (EXOR or XOR) operation and AND operation. The resulting algebra is that of the finite or Galois field GF(2). This mode of representation supports the familiar mathematical operations such as matrices, transforms, polynomials, etc.[Green 86]. The resulting circuits are much easier to test than their counterparts in the Boolean domain[Reddy 72].

Unlike the situations in the Boolean domain, there exist a great deal of canonical expansions[Davio 78, Green 91A, Sasao 93B] in the RM domain, this makes it more difficult to minimize a logic function in the RM domain than in the Boolean domain. In addition, for a given logic function, its canonical expansions in the Boolean domain correspond directly to a truth table that completely specifies this function. For a logic function in the RM domain, there is no similar relationship between its canonical expansions and a truth table. This means that, no general means, like a truth table in the Boolean domain, can be used to specify a logic problem in the RM domain initially. In general, a RM function is derived from a Boolean function by a coefficient conversion.

In this chapter, the background theory of RM logic is introduced. This shall be used in subsequent chapters.

2.2 The algebra of GF(2)

The definitions for the two basic operation in GF(2) are shown in Table 2.2.1.
Assume that a, b, and c are two-valued variables, and symbols "$\oplus$" and "$\cdot$" represent modulo 2 addition (XOR operation) and modulo 2 multiplication (AND operation), respectively. The symbol "$\cdot$" for AND operation can be missed if no confusion arises, i.e., $a \cdot b = ab$.

Table 2.2.1. Arithmetic operation tables of GF(2).

(1). Modulo 2 addition.		(2). Modulo 2 multiplication.	
a b	$a \oplus b$	a b	$a \cdot b$
0 0	0	0 0	0
0 1	1	0 1	0
1 0	1	1 0	0
1 1	0	1 1	1

Some basic laws can be described as follows

1. Closure laws $a \oplus b$ and $a \cdot b$ are also two-valued (2.2.1)
2. Associative laws $a \oplus b \oplus c = (a \oplus b) \oplus c = a \oplus (b \oplus c)$
$a \cdot b \cdot c = (a \cdot b) \cdot c = a \cdot (b \cdot c)$ (2.2.2)
3. Distributive laws $a \cdot (b \oplus c) = a \cdot b \oplus a \cdot c$ (2.2.3)
4. Commutative laws $a \oplus b = b \oplus a$, $a \cdot b = b \cdot a$ (2.2.4)
5. Identities $a \oplus 0 = a$, $a \cdot 1 = a$ (2.2.5)

It is easily seen that these five laws are similar to that in Boolean algebra, the difference is that operator "$\oplus$" is used to replace operator "+", where "+" is the inclusive OR operation, simply called OR operation.
There are two distinct operations which are quite different from that in Boolean algebra, they are:

6. $a \oplus a = 0$ (2.2.6)

7. $a \oplus 1 = \bar{a}$ (2.2.7)

$a \oplus a = 0$ is also called inverse operation, namely, every element is its own additive inverse. This means that addition and subtraction are the same in modulo 2 operation, i.e., $a = -a$.

The two distinct operations lead to very significant differences in the resulting algebra, and consequently in logic circuit design, implementation, and test aspects.

From (2.2.6) and (2.2.7), other important properties can be derived as follows

$$a \oplus b = \bar{a} \oplus \bar{b} \quad (2.2.8)$$

$$\bar{a} \oplus b = a \oplus \bar{b} = \overline{a \oplus b} = 1 \oplus a \oplus b \quad (2.2.9)$$

(2.2.8) and (2.2.9) present flexibility in logic circuit implementation. According to (2.2.8), AND gates can be replaced by more economical NAND gates. (2.2.9) shows that electrical polarity is easy to adjust for a given logic circuit realized in RM expansion.

There exist many other operations for RM expansions, most of them can be derived from these basic operations 1~7. For example, $\bar{a} \oplus a = 1$ can be easily deduced from equations (2.2.6) and (2.2.7).

Connectives between inclusive OR (OR) and exclusive OR (XOR) are:

$$\bar{a}b + a\bar{b} = a \oplus b \quad (2.2.10)$$

$$a + b = a \oplus b \oplus ab \quad (2.2.11)$$

$$f_1 a + f_2 \bar{a} = f_1 a \oplus f_2 \bar{a} \quad (2.2.12)$$

Equation (2.2.12) is often termed Shannon's decomposition, it can be also written as

$$f_1 a + f_2 \bar{a} = f_1 a \oplus f_2 \bar{a} = f_1 \oplus (f_1 \oplus f_2)\bar{a} = (f_1 \oplus f_2)a \oplus f_2 \quad (2.2.13)$$

2.3 Zero Polarity RM Canonical Expansion

A given logic function with n variables can be described by a truth table that is in its vector form

$$A=\begin{bmatrix} a_0 & a_1 & a_2 & \cdots\cdots & a_{2^n-1} \end{bmatrix}$$

where $a_i \in\{0,1\}$, termed Boolean coefficients or a_i coefficients, represent the output values of a truth table.

In the Boolean domain, the disjunctive canonical expansion, or so called sum of product (SOP) canonical expansion, can be used for representing a given logic function. The expansion is based on minterm m_i, and it forms an equivalent algebraic expression to the truth table, it is

$$f(x_{n-1},\cdots,x_1,x_0)=\sum_{i=0}^{2^n-1} a_i m_i = a_0 m_0 + a_1 m_1 + \cdots\cdots + a_{2^n-1} m_{2^n-1} \qquad (2.3.1)$$

where $\sum$ represents logical summation (OR operation).

If each of the n variables is restricted to appear only in true form, and also a coefficient vector is defined as

$$B=\begin{bmatrix} b_0 & b_1 & b_2 & \cdots\cdots & b_{2^n-1} \end{bmatrix}$$

where $b_i \in\{0,1\}$, termed b_i coefficients, then, a matrix operation of n vectors with a form $[1 \quad x_i]$ is written as

$$\begin{aligned} &\{[1 \quad x_0]*[1 \quad x_1]*[1 \quad x_2]*\cdots\cdots*[1 \quad x_{n-1}]\}\cdot B \\ &=[1 \quad x_0 \quad x_1 \quad x_1x_0 \quad x_2 \quad x_2x_0 \quad \cdots \quad \cdots \quad x_{n-1}\cdots x_2x_1x_0]\cdot B \\ &=[b_0 \quad b_1x_0 \quad b_2x_1 \quad b_3x_1x_0 \quad b_4x_2 \quad b_5x_2x_0 \quad \cdots \quad \cdots \quad b_{2^n-1}x_{n-1}\cdots x_2x_1x_0] \end{aligned} \qquad (2.3.2)$$

where "$*$" indicates the Kronecker product[Green 86]. An example is used to illustrate the Kronecker product. Assume that two 2×2 matrixes

$$A=\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}, \qquad B=\begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} \tag{2.3.3}$$

the Kronecker product of them is

$$A*B=\begin{bmatrix} a_{00}[B] & a_{01}[B] \\ a_{10}[B] & a_{11}[B] \end{bmatrix} \tag{2.3.4}$$

In general, if A is a $p\times q$ matrix and B is a $r\times s$ matrix, then $A*B$ is a $pr\times qs$ matrix. The "$*$" operation is associative but not commutative, i.e.,

$$A*B*C=A*(B*C)=(A*B)*C \tag{2.3.5}$$

$$A*B\neq B*A \tag{2.3.6}$$

If the XOR operation is used to be connectives between the elements in (2.3.2), an expansion can be represented as

$$\begin{aligned} f(x_{n-1},\cdots,x_1,x_0)=b_0\oplus b_1x_0\oplus b_2x_1\oplus b_3x_1x_0\oplus b_4x_2\oplus b_5x_2x_0\oplus \\ \cdots\cdots\oplus b_{2^n-1}x_{n-1}\cdots x_2x_1x_0 \end{aligned} \tag{2.3.7}$$

Equation (2.3.7) is the so-called zero polarity RM expansion or positive polarity RM expansion, which can be used to describe any given logic function with n variables.

2.4 Relationships Between a_i and b_i Coefficients

Unlike a_i coefficients (Boolean coefficients), b_i coefficients don't correspond to the output of a truth table, that is, b_i coefficients can't be obtained directly from a truth table, because b_i coefficients don't have a definite logical significance. a_i coefficients and b_i coefficients can be transformed to each other. This transform is based on a transform matrix T_1 that is defined as

$$T_1 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \tag{2.4.1}$$

For a two variable function, the transform can be written as

$$\begin{aligned} b_0 &= a_0 \\ b_1 &= a_0 \oplus a_1 \\ b_2 &= a_0 \oplus a_2 \\ b_3 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \end{aligned} \tag{2.4.2}$$

In matrix form, this transform is expressed as

$$B = T_2 \cdot A = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad \text{over GF(2)} \tag{2.4.3}$$

where T_2 is a transform matrix for two variables. It is easily verified that

$$T_2 = T_1 * T_1 = \begin{bmatrix} T_1 & 0 \\ T_1 & T_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \tag{2.4.4}$$

In general, the transform from a_i coefficients to b_i coefficients in matrix form can be expressed as

$$B = T_n \cdot A \tag{2.4.5}$$

and

$$T_n = T_1 * T_{n-1} = T_1 * T_1 * \cdots\cdots * T_1 \quad n \text{ times} \tag{2.4.6}$$

Furthermore, it can be established that

$$T_1^{-1} = T_1 \tag{2.4.7}$$

so that

$$T_n^{-1} = T_n \tag{2.4.8}$$

this means that the matrix is self-inverse. Thus

$$A = T_n \cdot B \tag{2.4.9}$$

2.5 Fixed Polarity RM Canonical Expansions

The zero polarity RM canonical expansion is just one of a large number of possible expansions. If each of the n variables is restricted to appear in its true form or its

complemented form, but not both, this leads to 2^n combinations from n variables, which correspond to 2^n so called fixed polarity RM canonical expansions.
Like the zero polarity expansion, a coefficient vector C is defined as

$$C=\begin{bmatrix} c_0 & c_1 & c_2 & \cdots\cdots & c_{2^n-1} \end{bmatrix}$$

where $c_i \in \{0,1\}$, termed c_i coefficients.

Assume that $\dot{x}_i$ represents x_i or $\overline{x_i}$, but not both, in a consistent way throughout a logic function. Thus

$$\begin{aligned} &\{[1 \quad \dot{x}_0]*[1 \quad \dot{x}_1]*[1 \quad \dot{x}_2]*\cdots\cdots*[1 \quad \dot{x}_{n-1}]\}\cdot C \\ &=[1 \quad \dot{x}_0 \quad \dot{x}_1 \quad \dot{x}_1\dot{x}_0 \quad \dot{x}_2 \quad \dot{x}_2\dot{x}_0 \quad \cdots \quad \cdots \quad \dot{x}_{n-1}\cdots\dot{x}_2\dot{x}_1\dot{x}_0]\cdot C \\ &=[c_0 \quad c_1\dot{x}_0 \quad c_2\dot{x}_1 \quad c_3\dot{x}_1\dot{x}_0 \quad c_4\dot{x}_2 \quad c_5\dot{x}_2\dot{x}_0 \quad \cdots \quad \cdots \quad c_{2^n-1}\dot{x}_{n-1}\cdots\dot{x}_2\dot{x}_1\dot{x}_0] \end{aligned} \tag{2.5.1}$$

Similarly to that for the zero polarity RM expansion, the vector (2.5.1) can be written as

$$\begin{aligned} f(x_{n-1},\cdots,x_1,x_0) = c_0 \oplus c_1\dot{x}_0 \oplus c_2\dot{x}_1 \oplus c_3\dot{x}_1\dot{x}_0 \oplus c_4\dot{x}_2 \oplus c_5\dot{x}_2\dot{x}_0 \oplus \\ \cdots\cdots \oplus c_{2^n-1}\dot{x}_{n-1}\cdots\dot{x}_2\dot{x}_1\dot{x}_0 \end{aligned} \tag{2.5.2}$$

Clearly, the zero polarity RM expansion is a special case of the 2^n fixed polarity RM expansions.

2.6 Relationships Between a_i and c_i Coefficients

It is possible to employ similar matrix techniques as above to transform between a_i coefficients and c_i coefficients.

For convenience to describe the following transformation, a polarity number is defined as

$$P=(p_{n-1}p_{n-2}\cdots\cdots p_0)_2$$

where $p_i \in\{0,1\}$, corresponding to the position of x_i. This is a binary representation of the decimal integer P. If the true form of x_i is used, then $p_i=0$; and if its complemented form is used, then $p_i=1$. For example, for a three variable function, $P=3=(011)_2$, this means that x_2 is in its true form and both x_1 and x_0 are in their complemented form.

Two basic transform matrixes are defined as

$$W_0=\begin{bmatrix}1 & 0\\ 1 & 1\end{bmatrix},\qquad W_1=\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix} \tag{2.6.1}$$

Clearly,

$$W_0=T_1 \tag{2.6.2}$$

since the fixed polarity expansions contain the zero polarity expansion. Thus, a generalized matrix W can be expressed as

$$W_p=W_{p_{n-1}}*W_{p_{n-2}}*\cdots\cdots*W_{p_0} \tag{2.6.3}$$

where $W_{p_i}=W_0$ or W_1, depending on the form (true or complemented) of each variable. Thus, the transform from a_i coefficients to c_i coefficients is

$$C=W_p\cdot A \tag{2.6.4}$$

For example, if $p_i=3=(011)_2$, then

$$C = W_3 \cdot A = \{W_0 * W_1 * W_1\} \cdot A \tag{2.6.5}$$

Thus

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} \tag{2.6.6}$$

c_i coefficients can be also obtained from b_i coefficients. That is

$$C = W_p \cdot A = W_p \cdot T_n \cdot B = Z_p \cdot B \tag{2.6.7}$$

Similarly to W_p, Z_p is expressed as

$$Z_p = Z_{p_{n-1}} * Z_{p_{n-2}} * \cdots\cdots * Z_{p_0} \tag{2.6.8}$$

Z_0 and Z_1 are

$$Z_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad Z_1 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \tag{2.6.9}$$

From equation (2.6.7), it can be seen that

$$Z_0 = W_0 \cdot T_1 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{2.6.10}$$

$$Z_1 = W_1 \cdot T_1 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \tag{2.6.11}$$

2.7 Kronecker RM(KRM) Canonical Expansions

RM canonical expansion can be further extended if some of the variables can be allowed to appear in both true and complemented form in a consistent way. This means that another basis vector $[x_i \quad \overline{x_i}]$ is included along with $[1 \quad x_i]$ and $[1 \quad \overline{x_i}]$ that are used only to generate the fixed polarity expansions.

By selecting one of these three bases for each variable, it will generate 3^n distinct expansions which are called Kronecker RM (KRM) canonical expansions. The KRM expansions contain the 2^n fixed polarity RM expansions and some of mixed polarity RM expansions.

Similarly to the previous approach, a coefficient vector for KRM expansions is defined as

$$E = [e_0 \quad e_1 \quad e_2 \quad \cdots\cdots \quad e_{2^n-1}]$$

where $e_i \in \{0,1\}$, termed e_i coefficients.

Also, a polarity number is defined as

$$M = (m_{n-1} m_{n-2} \cdots\cdots m_0)_3$$

where $m_i \in \{0,1,2\}$. This is a ternary representation of the decimal integer M. If $[1 \quad x_i]$ is used, then $m_i = 0$; if $[1 \quad \overline{x_i}]$ is used, then $m_i = 1$; and if $[x_i \quad \overline{x_i}]$ is used, then $m_i = 2$.

For example, when $n=3$ and $M=15=(120)_3$, i.e., $1\times3^2+2\times3^1+0\times3^0=15$, the KRM expansion can be obtained by

$$
\begin{aligned}
&\{\begin{bmatrix}1 & \overline{x_2}\end{bmatrix}*\begin{bmatrix}x_1 & \overline{x_1}\end{bmatrix}*\begin{bmatrix}1 & x_0\end{bmatrix}\}\cdot E \\
&=\begin{bmatrix}x_1 & x_1x_0 & \overline{x_1} & \overline{x_1}x_0 & \overline{x_2}x_1 & \overline{x_2}x_1x_0 & \overline{x_2}\,\overline{x_1} & \overline{x_2}\,\overline{x_1}x_0\end{bmatrix}\begin{bmatrix}e_0\\e_1\\e_2\\e_3\\e_4\\e_5\\e_6\\e_7\end{bmatrix} \\
&=\begin{bmatrix}e_0x_1 & e_1x_1x_0 & e_2\overline{x_1} & e_3\overline{x_1}x_0 & e_4\overline{x_2}x_1 & e_5\overline{x_2}x_1x_0 & e_6\overline{x_2}\,\overline{x_1} & e_7\overline{x_2}\,\overline{x_1}x_0\end{bmatrix}
\end{aligned}
\tag{2.7.1}
$$

Similarly to the previous cases, the vector (2.7.1) is expressed as

$$
\begin{aligned}
f(x_2,x_1,x_0)=&e_0x_1\oplus e_1x_1x_0\oplus e_2\overline{x_1}\oplus e_3\overline{x_1}x_0\oplus e_4\overline{x_2}x_1\\
&\oplus e_5\overline{x_2}x_1x_0\oplus e_6\overline{x_2}\,\overline{x_1}\oplus e_7\overline{x_2}\,\overline{x_1}x_0
\end{aligned}
\tag{2.7.2}
$$

2.8 Relationships Between a_i and e_i Coefficients

Like c_i coefficients, e_i coefficients can be derived from a_i coefficients or b_i coefficients by employing similar matrix techniques. In order to obtain e_i coefficients from a_i coefficients, three basic matrixes are introduced, they are

$$
Q_0=\begin{bmatrix}1 & 0\\ 1 & 1\end{bmatrix},\quad Q_1=\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix},\quad Q_2=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}
\tag{2.8.1}
$$

Since the KRM expansions involve the fixed polarity expansions, then, e_i coefficients should contain c_i coefficients. Thus,

$$Q_0 = W_0, \quad Q_1 = W_1 \tag{2.8.2}$$

Like the matrix W, the matrix Q can be expressed as

$$Q_m = Q_{m_{n-1}} * Q_{m_{n-2}} * \cdots\cdots * Q_{m_0} \tag{2.8.3}$$

where $Q_{m_i} = Q_0$, or Q_1, or Q_2, depending on the base selected ($\begin{bmatrix} 1 & x_i \end{bmatrix}$, or $\begin{bmatrix} 1 & \overline{x_i} \end{bmatrix}$, or $\begin{bmatrix} x_i & \overline{x_i} \end{bmatrix}$) for each variable, respectively. Thus, the transform from a_i coefficients to e_i coefficients is

$$E = Q_m \cdot A \tag{2.8.4}$$

In order to obtain e_i coefficients from b_i coefficients, the three basic matrixes are

$$K_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad K_1 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad K_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \tag{2.8.5}$$

Similarly to the matrix Q, the matrix K is expressed as

$$K_m = K_{m_{n-1}} * K_{m_{n-2}} * \cdots\cdots * K_{m_0} \tag{2.8.6}$$

Clearly, the matrix K and the matrix Z should have the following relationships

$$K_0 = Z_0, \quad K_1 = Z_1 \tag{2.8.7}$$

Thus, the transform from b_i coefficients to e_i coefficients is

$$E = K_m \cdot B \tag{2.8.8}$$

2.9 Inconsistent Forms

The 3^n KRM expansions, which are generated in a systematic or consistent way, are not the only possible canonical expansions. In fact, if the polarity of each literal in the zero polarity RM expansion can be freely chosen, there exist $2^{n2^{n-1}}$ possible RM canonical expansions[Davio 78, Green 91A, Sasao 93B]. This is because there are 2^n terms for the n variable RM canonical expansion and the 2^n terms contain $n2^{n-1}$ literals. That is, in equation (2.3.7), any single literal a ($\bar{a}$) can be substituted by $\bar{a} \oplus 1$ ($a \oplus 1$).

These $2^{n2^{n-1}}$ possible RM canonical expansions have been termed inconsistent GRM forms[Green 91B]. In practice, this large set contains 3^n consistent RM expansions, i.e. KRM expansions.

Furthermore, arbitrary products terms combined by XORs are called an Exclusive OR sum of products (ESOP) which is the most general AND-XOR expression. There are at most 3^{tn} different ESOPs[Sasao 93B], where t is the number of the products.

2.10 Dual Forms of RM Expansions

Dual forms of RM expansions are investigated by Green[Green 94]. He introduces an operation termed logical equivalence (LEQ). This operation is identical to XNOR operation in the Boolean domain. In the following, the basic operations are described. For easy comparison with operations (2.2.1)~(2.2.7) described above, the same number order is employed, and symbol "$\overline{\oplus}$" is used for LEQ operation, i.e. XNOR operation.

1. Closure laws $a \overline{\oplus} b$ also two-valued (2.10.1)

2. Associative laws	$a \overline{\oplus} b \overline{\oplus} c = (a \overline{\oplus} b) \overline{\oplus} c = a \overline{\oplus} (b \overline{\oplus} c)$	(2.10.2)
3. Distributive laws	$a \cdot (b \overline{\oplus} c) = a \cdot b \overline{\oplus} a \cdot c$	(2.10.3)
4. Commutative laws	$a \overline{\oplus} b = b \overline{\oplus} a$	(2.10.4)
5. Identities	$a \overline{\oplus} 1 = a$	(2.10.5)
6.	$a \overline{\oplus} a = 1$	(2.10.6)
7.	$a \overline{\oplus} 0 = \overline{a}$	(2.10.7)

Similarly to (2.2.8) and (2.2.9)

$$a \overline{\oplus} b = \overline{a} \overline{\oplus} \overline{b} \tag{2.10.8}$$

$$\overline{a} \overline{\oplus} b = a \overline{\oplus} \overline{b} = \overline{a \overline{\oplus} b} \tag{2.10.9}$$

It is not difficult to find that equations (2.10.8) and (2.10.9) have similar properties to equations (2.2.8) and (2.2.9).
There are some relationships between XOR and XNOR operations, they are

$$a \overline{\oplus} b = \overline{a \oplus b} = a \oplus b \oplus 1 \tag{2.10.10}$$

$$\overline{a} \overline{\oplus} \overline{b} \overline{\oplus} \overline{c} \overline{\oplus} \cdots = \overline{a \oplus b \oplus c \oplus \cdots} \tag{2.10.11}$$

$$\overline{a \overline{\oplus} b \overline{\oplus} c \overline{\oplus} \cdots} = \overline{a} \oplus \overline{b} \oplus \overline{c} \oplus \cdots \tag{2.10.12}$$

In the Boolean domain, there exists another algebraic form to describe a truth table, this is the conjunctive canonical expansion, also called product of sum (POS) canonical expansion, which is based on maxterm M_i. It is

$$f(x_{n-1}, \cdots, x_1, x_0) = \prod_{i=0}^{2^n - 1} (a_i + M_i) = (a_0 + M_0) \cdot (a_1 + M_1) \cdots\cdots (a_{2^n - 1} + M_{2^n - 1}) \tag{2.10.13}$$

where $\prod$ represents logical product (AND operation).

Similarly to the conjunctive canonical expansion, in the RM domain, dual forms of RM expansions can be established.

Assume that the Kronecker matrix product operates with XNOR and OR, that is

$$[a \quad b]\bullet[c \quad d]=[a+c \quad a+d \quad b+c \quad b+d] \tag{2.10.14}$$

Also, if $\hat{B}$ is the dual form of b coefficients, i.e.

$$\hat{B}=\begin{bmatrix} \hat{b}_{2^n-1} & \cdots\cdots & \hat{b}_2 & \hat{b}_1 & \hat{b}_0 \end{bmatrix}$$

where $\hat{b}_i \in\{0,1\}$, termed $\hat{b}_i$ coefficients. Therefore, a POS canonical expansion in the RM domain can be set up wherein the OR operation provides the "sum" and XNOR gives the "product". The basis vector is $[0 \quad x_i]$.

$$f(x_{n-1},\cdots,x_1,x_0)=\{[0 \quad x_{n-1}]\bullet\cdots\cdots\bullet[0 \quad x_2]\bullet[0 \quad x_1]\bullet[0 \quad x_0]\}\circ\hat{B} \tag{2.10.15}$$

where the operator "$\circ$" represents matrix multiplication based on XNOR and OR. For example, if $n=3$

$$\begin{aligned} f(x_2,x_1,x_0)&=\{[0 \quad x_2]\bullet[0 \quad x_1]\bullet[0 \quad x_0]\}\circ\hat{B} \\ &=[0 \quad x_0 \quad x_1 \quad x_1+x_0 \quad x_2 \quad x_2+x_0 \quad x_2+x_1 \quad x_2+x_1+x_0]\circ\hat{B} \\ &=\hat{b}_7\,\overline{\oplus}\,(\hat{b}_6+x_0)\,\overline{\oplus}\,(\hat{b}_5+x_1)\,\overline{\oplus}\,(\hat{b}_4+x_1+x_0)\,\overline{\oplus}\,(\hat{b}_3+x_2) \\ &\quad\overline{\oplus}\,(\hat{b}_2+x_2+x_0)\,\overline{\oplus}\,(\hat{b}_1+x_2+x_1)\,\overline{\oplus}\,(\hat{b}_0+x_2+x_1+x_0) \end{aligned} \tag{2.10.16}$$

Similarly to the situations in ESOP form, the dual forms of the fixed polarity expansions can be deduced by using the extra basis vector $[0 \quad \overline{x_i}]$ for some variables, and those for KRM expansions can be derived by employing the third basis vector $[\overline{x_i} \quad x_i]$. In addition, the transforms of the dual coefficients can be established by using a similar matrix technique[Green 94].

2.11 Map Method

The map method is an efficient and powerful means to deal with coefficient conversion and minimization for a logic function in the RM domain, it also forms a basis of many other algorithms. A RM coefficient map termed b_i coefficient map (simply called b_i map) was introduced by Wu et al[Wu 82]. The b_i map can be used to transform a_i coefficients to b_i coefficients and c_i coefficients, and also, it can be employed to minimize a RM expansion. The minimized result is in a mixed polarity RM representation. This map, refined by Tran[Tran 87], who generalized the b_i map to deal with incompletely specified functions. In addition, minimization of a RM expansion can be carried out, not only on RM coefficient maps, but also on Karnaugh maps, even on any of the transition maps[Tran 87, 89].

The b_i map, in format, is similar to the Karnaugh map (k map), but its entries do not represent the function output in the same manner as do the minterm entries plotted on a k map. For ease of comparison, a four variable k map and a four variable b_i map are shown together in Fig. 2.11.1

x_3x_2 \ x_1x_0	00	01	11	10
00	a_0	a_1	a_3	a_2
01	a_4	a_5	a_7	a_6
11	a_{12}	a_{13}	a_{15}	a_{14}
10	a_8	a_9	a_{11}	a_{10}

(a). k map in the Boolean domain.

x_3x_2 \ x_1x_0	00	01	11	10
00	b_0	b_1	b_3	b_2
01	b_4	b_5	b_7	b_6
11	b_{12}	b_{13}	b_{15}	b_{14}
10	b_8	b_9	b_{11}	b_{10}

(b). b_i map in the RM domain.

Fig. 2.11.1. Maps for four variables.

It should be noted that the meaning of the subscript i in k map is different from that in b_i map. In k map, "0" means that a variable appears in its complemented form and "1" means that a variable appears in its true form. In b_i map, "0" means that a variable does not appear and "1" means that a variable appears in its true or complemented form, depending on the polarity[Green 86].

2.11.1 Folding Technique

A folding technique[Wu 82, Besslich 83, Tran 87] can be employed to transform a_i coefficients to b_i coefficients and c_i coefficients. This can be used to find a fixed polarity expansion with the minimum "1" entries in the map. The procedures are as follows:

(*a*). Construct the k map of a given logic function $f(x_{n-1},\cdots,x_1,x_0)$, where the number of variables are less or equal to six.
(*b*). Decide the required polarity of fixed polarity RM expansion.
(*c*). Select a particular variable x_i, fold $\overline{x_i}$ over x_i and XOR the two portions. The resulting XOR values make up the portion x_i and the portion $\overline{x_i}$ remains the same as that of the old map.
(*d*). Repeat step (*c*) for all the other variables from the last map generated in (*c*).

An example is employed to show the folding technique. A four variable logic function is given in the Boolean domain

$$f(x_3,x_2,x_1,x_0)=\sum(2,3,5,8,10,14) \qquad (2.11.1)$$

i.e. $a_2=a_3=a_5=a_8=a_{10}=a_{14}=1$, transformation of a_i coefficients (Boolean) to b_i coefficients (zero polarity) is shown in Fig. 2.11.2

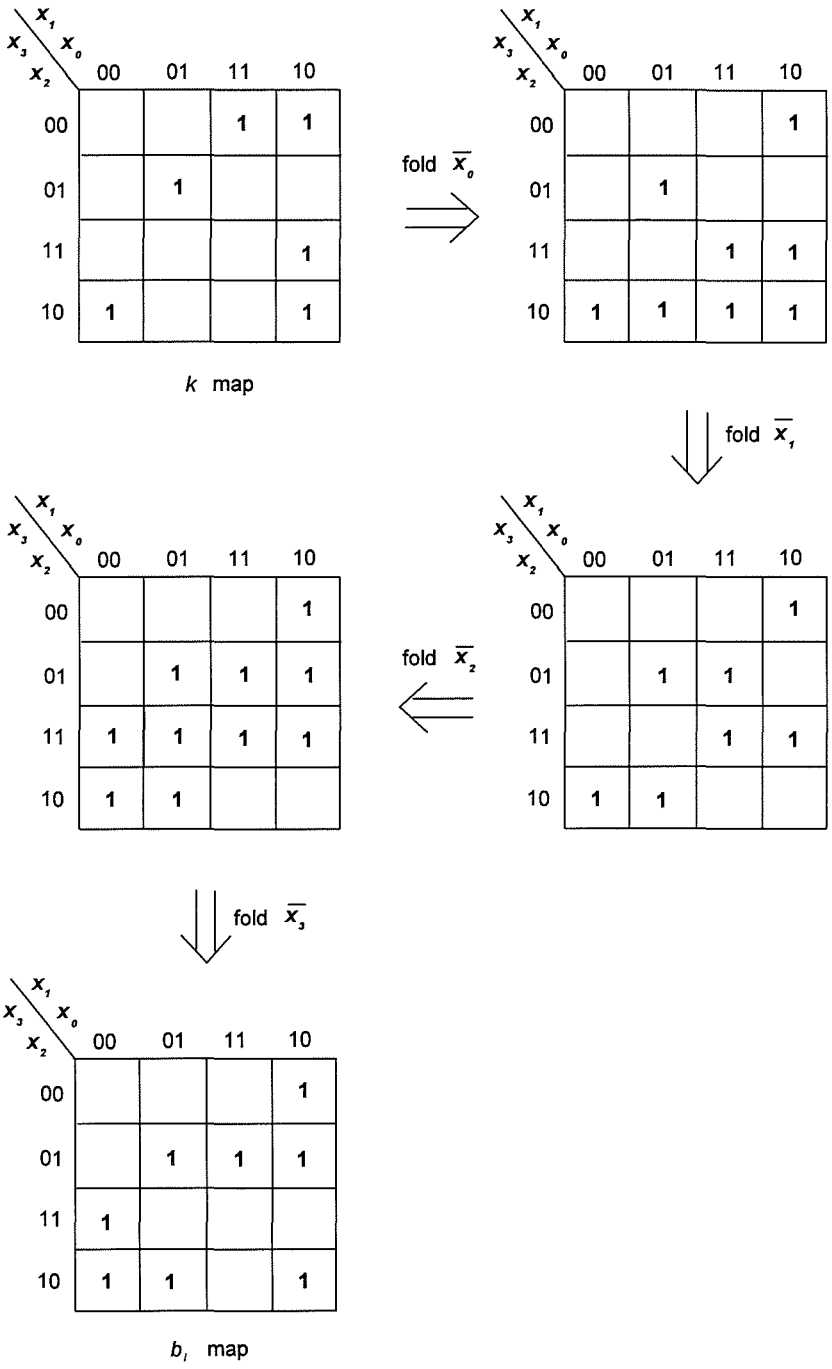


Fig. 2.11.2. An example to illustrate folding technique.

Note that the order of performing these individual folding operations does not affect the final result.

From the b_i map, $b_2 = b_5 = b_6 = b_7 = b_8 = b_9 = b_{10} = b_{12} = 1$. Hence

$$f(x_3, x_2, x_1, x_0) = x_1 \oplus x_2x_0 \oplus x_2x_1 \oplus x_2x_1x_0 \oplus x_3 \oplus x_3x_0 \oplus x_3x_1 \oplus x_3x_2 \quad (2.11.2)$$

A similar folding technique can be employed to find any one of the $2^n - 1$ fixed polarity RM expansions (not include $P = 0$) from a b_i map. This is achieved by folding x_i over $\overline{x_i}$, instead of $\overline{x_i}$ over x_i. For example, if polarity number $P = 13 = (1101)_2$ is required for equation (2.11.2), this means that variables x_3, x_2 and x_0 are required to convert to their complemented forms. This folding process is illustrated in Fig. 2.11.3. At the same time, the patterns for $P = 8 = (1000)_2$ and $P = (12) = (1100)_2$ are obtained.

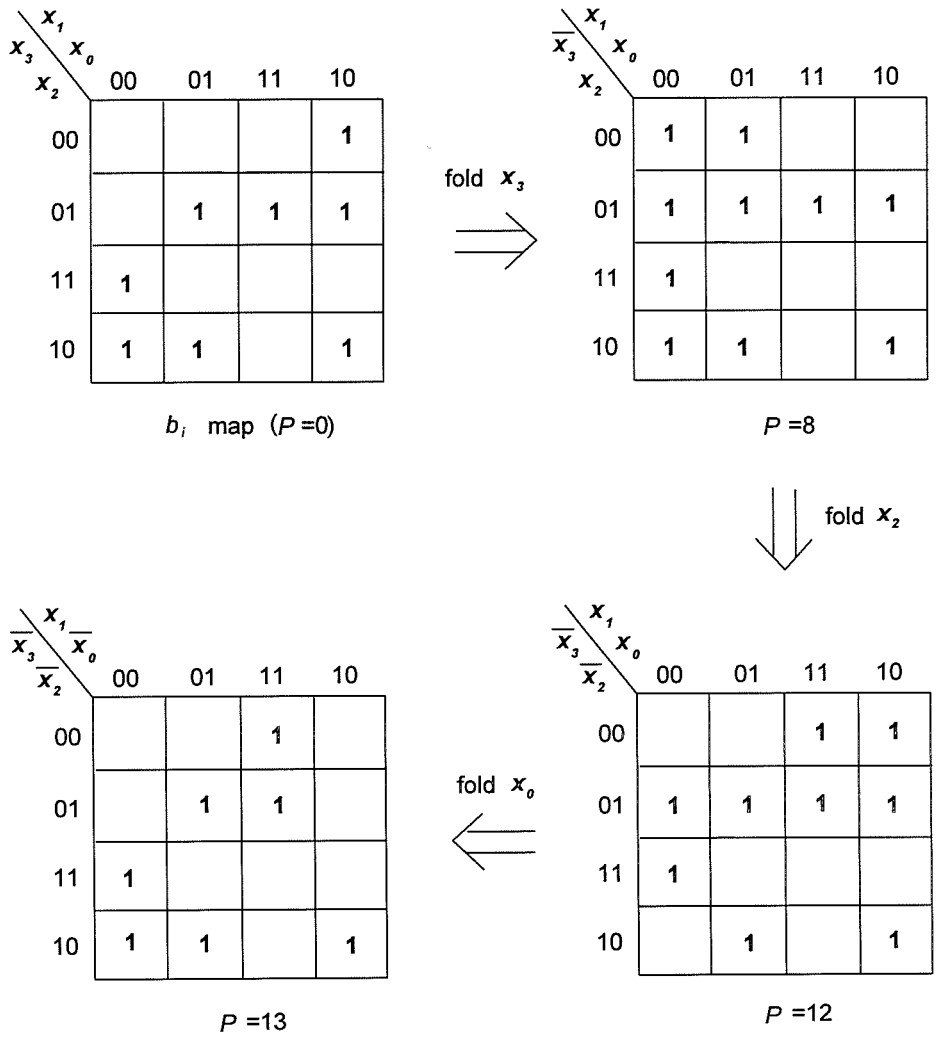


Fig. 2.11.3. Folding for $P = 13 = (1101)_2$.

Thus, for $P=8$

$$f(x_3,x_2,x_1,x_0)=1\oplus x_0\oplus x_2\oplus x_2x_0\oplus x_2x_1\oplus x_2x_1x_0 \\ \oplus\overline{x_3}\oplus\overline{x_3}x_0\oplus\overline{x_3}x_1\oplus\overline{x_3}x_2 \tag{2.11.3}$$

for $P=12$

$$f(x_3,x_2,x_1,x_0)=x_1\oplus x_1x_0\oplus\overline{x_2}\oplus\overline{x_2}x_0\oplus\overline{x_2}x_1 \\ \oplus\overline{x_2}x_1x_0\oplus\overline{x_3}x_0\oplus\overline{x_3}x_1\oplus\overline{x_3}\,\overline{x_2} \tag{2.11.4}$$

for $P=13$

$$f(x_3,x_2,x_1,x_0)=x_1\,\overline{x_0}\oplus\overline{x_2}\,\overline{x_0}\oplus\overline{x_2}x_1\,\overline{x_0}\oplus\overline{x_3}\oplus\overline{x_3}\,\overline{x_0}\oplus\overline{x_3}x_1\oplus\overline{x_3}\,\overline{x_2} \tag{2.11.5}$$

This can be checked by using $\overline{x_i}\oplus 1$ to replace x_i step by step from equation (2.11.2) to equation (2.11.5).

As described above, it can be seen that a general procedure is: a k map is first converted to a b_i map for the polarity number $P=0$, and then, the b_i map is converted to a fixed polarity as required.

In practice, a coefficient set for a given fixed polarity can be obtained directly from a k map in n steps, which employs a tri-state map introduced by Tran[Tran 89]. A tri-map is self-explanatory, its format is similar to that for k map.

In a tri-state, a variable has three states, true, complement and non-existent. Their respective labels in the maps are "1", "0" and "-". A variable in a tri-state map can exist in two of the three different states. The states of each variable can be readily found out by examining these labels. k maps with true and complement states are special cases in the tri-state maps. Polarisation is the process of converting a variable to a particular form, either true or complemented.

Polarisation for a given logic function starts from its k map, and the folding technique is used to polarize each variable, depending on the polarity required. A so called transition map is obtained every time a variable is polarized. The transition map obtained from the polarisation of the last variable is the RM coefficient map.

A tri-state map can be better understood by the illustration shown in Fig. 2.11.4, the same example as Fig. 2.11.2 is used. From the last map, it can be seen that, although the final result is the same as that in Fig. 2.11.3, the patterns for "1" entries in these two maps are different. In other words, different map formats have different explanations. There is a simple relationship between the two patterns. Assume that $g_3 g_2 g_1 g_0$ and $h_3 h_2 h_1 h_0$ in binary form are employed to represent the number of the cell in the two maps, then, the following relationship is set up

$$
\begin{array}{c}
g_3 g_2 g_1 g_0 \\
\oplus 1 \quad 1 \quad 0 \quad 1 \\
h_3 h_2 h_1 h_0
\end{array}
$$

$P = 1101$ is the polarity required. This can be readily generalized.

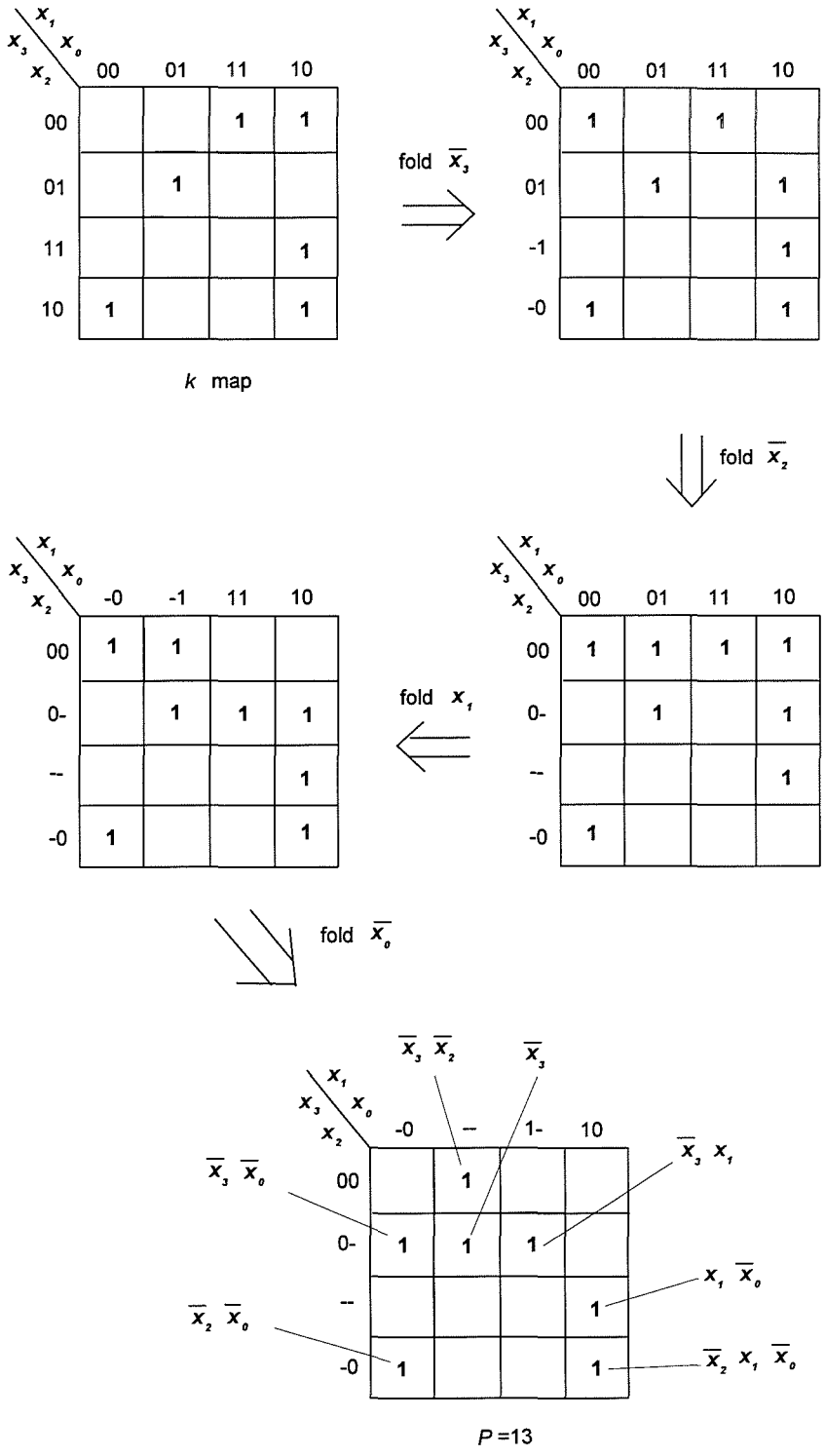


Fig. 2.11.4. A tri-state map to transform fixed polarity coefficients.

The folding technique can be readily extended to an incompletely specified function, in which, it includes some terms called don't care. Even though a don't care can be assumed to be an arbitrary value, "0" or "1", the result of a XOR operation for two

don't care values does not have an arbitrary value but depends on those two don't care values that were assumed before the operation[Tran 87].

2.11.2 Map Minimization

The b_i map can be employed to minimize a RM expansion, which is similar to the Karnaugh map. The major consideration is to obtain the minimum number of products for a given logic function in RM expansion, and the result can lead to a mixed polarity form.

Similarly to the Karnaugh map, the grouping is allowed for any group of 2^n adjacent cells. A significant difference between b_i map and k map is that each "1" entry can be grouped only an odd number of times, and a "0" entry can also be grouped with "1" entries if it is grouped for an even number of times.

This minimisation procedure is

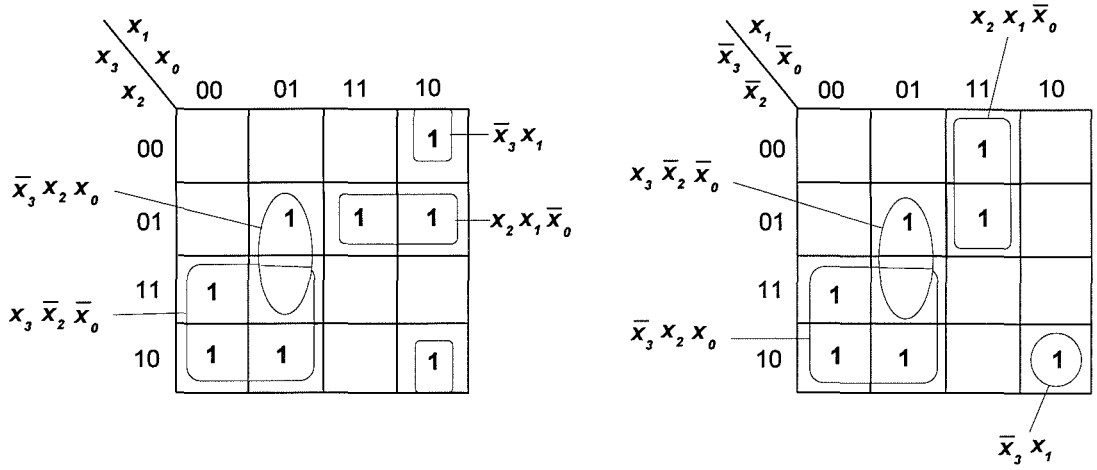
(*a*). If a group is within the true domain of a variable $\dot{x}_i$, the true form $\dot{x}_i$ appears in the product.

(*b*). If a group is within the complemented domain of a variable $\dot{x}_i$, this variable does not appear in the product.

(*c*). If a group spans both the true and complemented domains of a variable $\dot{x}_i$, the complemented form $\overline{\dot{x}_i}$ appears in the product.

Where $\dot{x}_i$ is x_i or $\overline{x_i}$.

Two patterns of Fig. 2.11.3 are used to illustrate the minimization in Fig. 2.11.5



(a). $P=0=(0000)_2$. (b). $P=13=(1101)_2$.

Fig. 2.11.5. Two examples to illustrate map minimization.

From the map (a)

$$f(x_3,x_2,x_1,x_0)=\overline{x_3}x_2x_0 \oplus x_3\,\overline{x_2}\,\overline{x_0} \oplus x_2x_1\,\overline{x_0} \oplus \overline{x_3}x_1 \tag{2.11.6}$$

From the map (b)

$$f(x_3,x_2,x_1,x_0)=x_3\,\overline{x_2}\,\overline{x_0} \oplus \overline{x_3}x_2x_0 \oplus x_2x_1\,\overline{x_0} \oplus \overline{x_3}x_1 \tag{2.11.7}$$

It should be noted that the true forms of variables $\dot{x_3}$, $\dot{x_2}$, $\dot{x_1}$ and $\dot{x_0}$ in the map (b) are explained with $\overline{x_3}$, $\overline{x_2}$, x_1 and $\overline{x_0}$, respectively, because of the polarity number $P=1101$. Equations (2.11.6) and (2.11.7) can be easily checked from equations (2.11.2) and (2.11.5).

When minimization is carried out in a tri-state map, the minimization procedure stated previously should be modified to cover a non-existent state "-". Thus, the minimization procedure is:

(*a*). If a group is within the true domain of a variable x_i, the true form x_i appears in the product.

(*b*). If a group is within the complemented domain of a variable x_i, the complemented form $\overline{x_i}$ appears in the product.

(*c*). If a group spans both the true and complemented domains of a variable x_i, the complemented form $\overline{x_i}$ appears in the product.

(*d*). If a group spans both the true and non-existent domains of a variable x_i, the complemented form $\overline{x_i}$ appears in the product.

(*e*). If a group spans both the complemented and non-existent domains of a variable x_i, the true form x_i appears in the product.

This minimization procedure covers the cases not only for a tri-state map, but also for a transition map. In the following, an example for a tri-state map is given, see Fig. 2.11.6. This map is the last map in Fig. 2.11.4

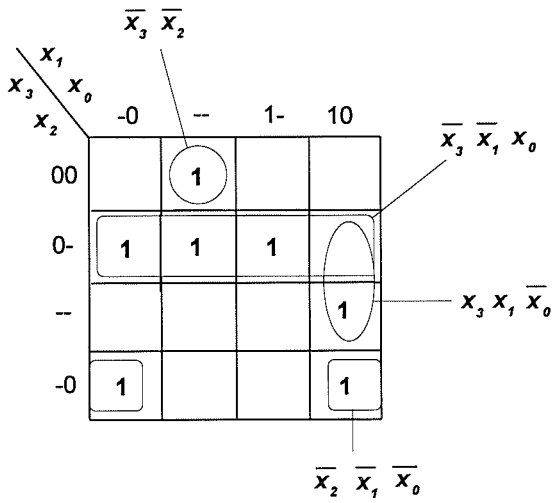


Fig. 2.11.6. A minimization example in a tri-state.

From Fig. 2.11.6

$$f(x_3, x_2, x_1, x_0) = \overline{x_3}\ \overline{x_2} \oplus \overline{x_3}\ \overline{x_1} x_0 \oplus x_3 x_1\ \overline{x_0} \oplus \overline{x_2}\ \overline{x_1}\ \overline{x_0} \qquad (2.11.8)$$

A RM function can be minimized in a k map. According to equation (2.2.12), if two terms are mutually exclusive, inclusive OR (+) can be replaced by exclusive OR ($\oplus$). Therefore, equation (2.3.1) can be rewritten as follows

$$f(x_{n-1},\cdots,x_1,x_0)=\sum_{i=0}^{2^n-1}a_i m_i = a_0 m_0 \oplus a_1 m_1 \oplus \cdots\cdots \oplus a_{2^n-1} m_{2^n-1} \qquad (2.11.9)$$

Equation (2.11.9) reveals that XOR operation can be a connective between all adjacent cells in a k map. This means that, for a given logic function, a k map can be employed to minimize its ESOP form or SOP form. The k map also directly generates a mixed form that contains both inclusive OR (+) and exclusive OR ($\oplus$) in a two level representation. Fig. 2.11.7 shows an example for minimizing a logic function in ESOP in a k map.

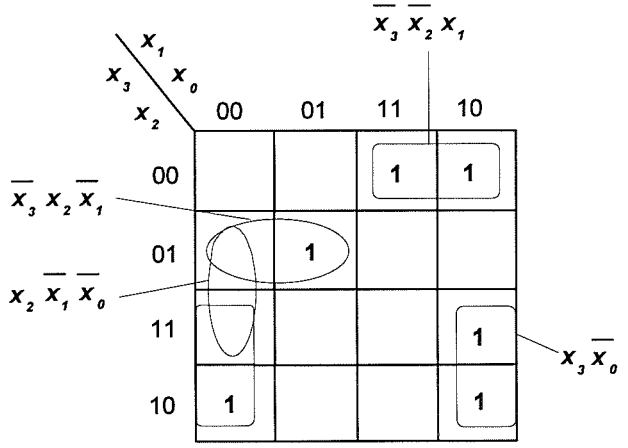


Fig. 2.11.7. Using k map to minimize a logic function in ESOP.

From Fig. 2.11.7

$$f(x_3,x_2,x_1,x_0)=\overline{x_3}x_2\,\overline{x_1}\oplus x_2\,\overline{x_1}\,\overline{x_0}\oplus x_3\,\overline{x_0}\oplus\overline{x_3}\,\overline{x_2}x_1 \qquad (2.11.10)$$

Equation (2.11.10) can be also written as

$$f(x_3, x_2, x_1, x_0) = (\overline{x_3} x_2\ \overline{x_1} \oplus x_2\ \overline{x_1}\ \overline{x_0} \oplus x_3\ \overline{x_0}) + \overline{x_3}\ \overline{x_2} x_1 \qquad (2.11.11)$$

This is because the term $\overline{x_3}\ \overline{x_2} x_1$ circled conforms not only to RM rules, but also to Boolean rules.

As a k map can be used to minimize a logic function in the RM domain, the other maps may seem to be perplexing. When the "1" entries are lumped together or there are few "1" entries in a k map, a minimal expansion may be easily obtained directly from the k map. For a logic function with a high content of XOR operation, the "1" entries may be scattered in the k map, and patterns of XOR operation are not easily recognized, especially because "0" entries can be circled with "1" entries. Tran shows an example for this[Tran 89].

2.12 Tabular Method

Although the map method does provide a powerful and compact means of representation for the display and manipulation of logic functions, it has its practical limitations. In general, the maps can only handle logic functions of six or less variables. In order to overcome this problem, similar to the Quine-McCluskey method in the Boolean domain, a tabular technique for RM logic was introduced by Almaini et al[Almaini 91, 94]. The tabular technique can convert a_i coefficients to b_i coefficients and c_i coefficients, and it can be used manually or programmed on a computer. In theory, it can be used for any number of variables. The technique is illustrated by means of an example. Assume that a logic function in the Boolean domain is

$$f(x_2, x_1, x_0) = \sum(0,2,3,5) \qquad (2.12.1)$$

(*a*). List all the minterms in binary form.

(*b*). Select any variable x_i. For every term containing a "0" in position i ($\overline{x_i}$), generate an additional term with a "1" in position i (x_i).

(*c*). Compare newly generated entries with the ones that already exist cancelling pairs whenever they occur since $\dot{x_i} \oplus \dot{x_i} = 0$.

(*d*). Repeat (*b*) and (*c*) for all other variables. The resulting uncancelled terms are the RM product terms in zero polarity.

Minterms	Terms generated by x_2
$x_2 x_1 x_0$	
0 0 0	1 0 0
0 1 0	1 1 0
0 1 1	1 1 1
1 0 1	

	Terms generated by x_1
$x_2 x_1 x_0$	
0 0 0	0 1 0×
0 1 0×	1 1 1×
0 1 1	1 1 0×
1 0 0	
1 0 1	
1 1 0×	
1 1 1×	

	Terms generated by x_0
0 0 0	0 0 1
0 1 1	1 0 1×
1 0 1×	
1 0 0	

From the remaining terms, the RM expansion is

$$f(x_2, x_1, x_0) = 1 \oplus x_0 \oplus x_1 x_0 \oplus x_2 \qquad (2.12.2)$$

The procedure can be reversed to convert b_i coefficients back to a_i coefficients. The tabular method has been developed to deal with don't care conditions[McKenzie 93]. It can be seen that the basic principle of the tabular technique is similar to that of the map folding technique. Therefore, the tabular technique can be developed to deal with such situations, in which, any c_i coefficients are transformed from the b_i coefficients, any c_i coefficients are found directly from a_i coefficients in $n+1$ steps for any n variable function[Almaini 91], and so on.

In addition, like the Quine-McCluskey method in the Boolean domain, the tabular technique can be readily developed for minimizing a RM function according to the rules in the RM domain[Helliwell 88, Green 93].

2.13 Summary

This chapter introduced the background theory of RM logic that underlies many algorithms of RM logic in aspects of coefficients conversion, manipulation and minimization. There are a great deal of RM canonical expansions for a given logic function, and there is no known method for predicting the best polarity except for exhaustive search[Wu 82]. This makes it more complex for the manipulation and minimization in the RM domain than that in the Boolean domain. Sometimes, it may be difficult to decide the minimum expansion of a logic function unless all the canonical expansions of this function have been considered.

The map method in the RM domain is an efficient tool not only for minimizing a logic function, but also for transforming between RM coefficients and Boolean coefficients. The maps described in this chapter are the most commonly used maps in the RM domain, but not the only ones. For example, a so called ternary map[Green 90] has different format, which can be used to deal with the KRM Expansions. A ternary map, in principle, is similar to the maps discussed in this chapter. Like a Karnaugh map, a map in the RM domain is restricted to handle a logic function of not more six variables.

A tabular method, similar to the Quine-McCluskey method in the Boolean domain, can be employed to resolve this problem. In practice, in order to treat logic functions with more variables, near minimal solutions, namely heuristic algorithms, are employed instead of absolutely minimum algorithms[Sasao 93A]. Owing to the

complexity of exclusive-ORing "1" terms with "0" terms in minimization, no single method can guarantee a global minimum[Tran 93A].

RM expansions usually are two-level logic representations. Based on similar logic algebraic principle in the Boolean domain, it is not difficult to develop them in a multi-level structure[Saul 91]. For multi-output functions, algorithm strategies in the Boolean domain may be adopted to the RM domain[Lin 93, Saul 93, Sasao 93A].

Chapter 3

CMOS Implementation of Logic Circuits

3.1 Introduction

In the earlier days of logic design, a logic function was implemented by only using discrete components, such as INV, NAND, NOR, AND, OR, XOR and XNOR gates. Therefore, counting the number of gates used was often employed to estimate a hardware implementation cost. With the advance of integrated circuit (IC) technology, some complex gates, such as AOI (AND-OR-INV), OAI (OR-AND-INV), or even a single complex gate constructed of a series and a parallel circuit, can be used to realize a logic function. This is specially true in MOS circuit technology[Wu 85]. It is well known that a circuit implemented by complex gates is often more compact and faster, when compared to a logic function realized by individual gates.

The representation and minimization of logic functions, sometimes considered to be technology-independent, are in practical applications technology-dependent. For example, the implementation based on NAND gates for a given logic function is preferred in TTL technology, and the implementation based on NOR gates is better than other gates in NMOS technology. Therefore, a minimized SOP form for a logic function is often changed into NAND-NAND form for TTL circuit, and NOR-NOR form for NMOS circuit. This can be readily achieved by employing De Morgan's theorem.

For a given circuit technology or circuit style, an efficient method of logic design is required not only to concisely represent a logic function and easily minimize the function, but also to allow a simpler hardware realization.

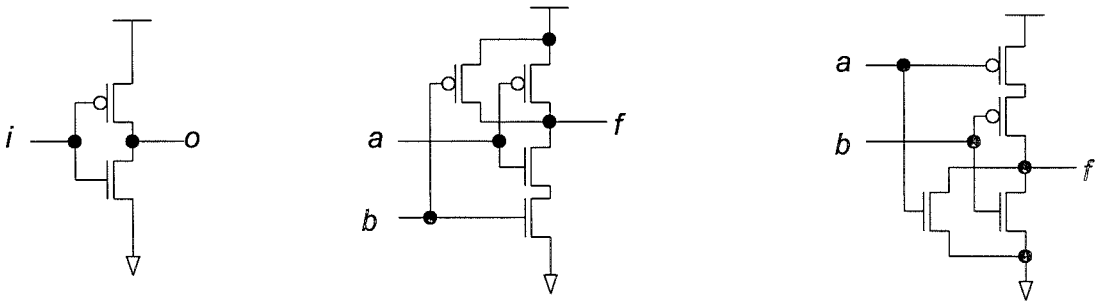
CMOS technology offers many advantages over other IC technologies due to its high speed, high component density, low power requirement and low cost. It is thus dominant in digital circuits[Wu 87, Hurst 92, Weste 93]. Therefore, this work is concerned mainly with CMOS circuits.

In this chapter, basic CMOS circuits are first described, then *switching network theory*, which was derived from the earlier Boolean logic design and now is a powerful tool in MOS circuit design[Wu 85], is reviewed. The circuit implementations based on Boolean logic and RM logic are investigated and compared. This investigation is related to gate level design, transistor level design and layout level design, but the comparison at the transistor level is emphasized. The implementations in PLA and FPGA, which are often considered to favor RM logic[Sasao 90, Csanky 93], are also studied. The mixed representations based on RM logic are introduced to minimize the implementation for RM logic. The main results about testing RM circuits are reviewed.

3.2 CMOS Circuits

CMOS circuits may be divided into two categories, dynamic circuits and static circuits, according to whether they require a clock, or not. Dynamic CMOS circuits and static CMOS circuits can be further divided into different circuit styles, which may be suitable for different applications. Weste and Eshraghian describe their advantages and disadvantages[Weste 93].

The complementary CMOS circuit (also called the true CMOS circuit) is a typical static circuit, and also, its design theory underlies other circuit styles. In applications, the complementary CMOS circuit is one of the most commonly used circuit styles, it is widely employed to construct standard cells in the gate library of a CAD suite, gate array, functional cells, and so on[Hurst 92, Weste 93], because it is reliable and simple for design. All complementary gates may be designed as ratioless circuits. That is, if all transistors are the same size the circuit will function correctly (compared to some other MOS logic families where this is not the case)[Weste 93]. Therefore, this section is concerned mainly with this circuit style.



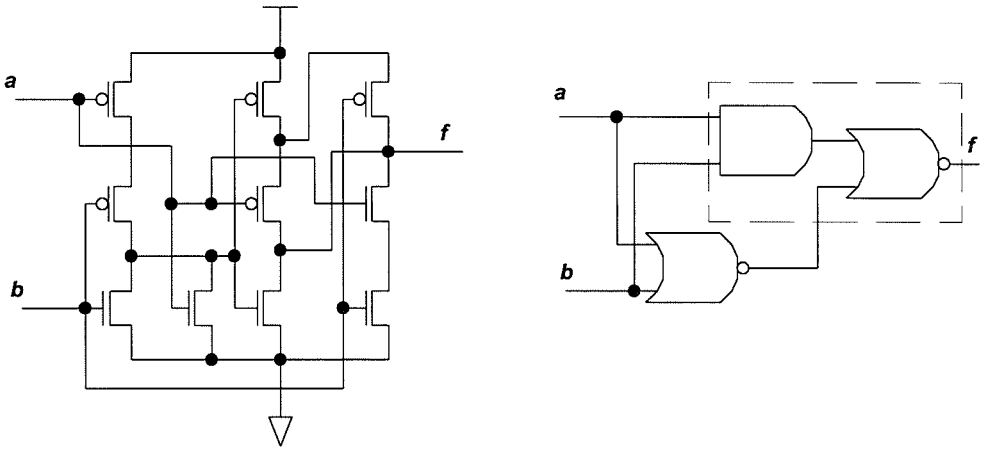
(a). Inverter. (b). 2 input NAND gate. (c). 2 input NOR gate.

Fig. 3.2.1. Three basic CMOS circuits.

In the complementary CMOS circuit, the minimum basic gates are composed of three gates, INV, NAND and NOR, which are shown in Fig. 3.2.1. The so-called minimum basic gates, here, mean that these three gates can't be decomposed further, and also, any kind of gates can be constructed by only using these three basic gates. An AND (OR) gate is constructed by a NAND (NOR) gate with an INV. According to the series and parallel principle, (N)AND gate and (N)OR gate can be easily extended for multi-input gates, but in practice, the number of inputs is restricted to four if high-speed circuits are desirable.

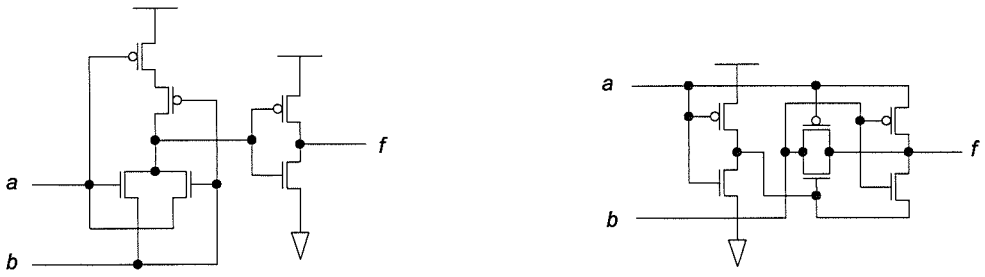
Since electrons have a higher mobility than holes, N type transistors are inherently faster by a factor of about 2.5 than P type transistors. Therefore, the NAND gate is better than the NOR gate in respect to speed, because of the slower P type transistors in parallel and faster N transistors in series for a NAND gate.

XOR and XNOR gates are often classified as complex gates. The reason for this is that, firstly, a single two input X(N)OR gate is usually more complex to realize than a single two input (N)AND gate or (N)OR gate; secondly, unlike (N)AND and (N)OR gates, X(N)OR gates can't be extended for the multi-input gates by simply employing the series or parallel principle. Therefore, in many situations, a multi-input X(N)OR gate is constructed by several two input gates. This problem also occurs in many other circuit technologies.

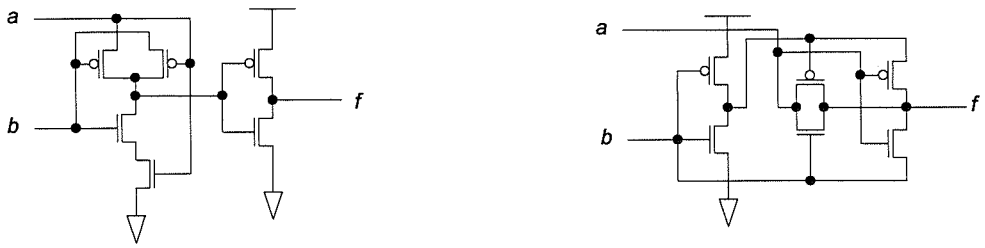


(a). Transistor form. (b). Gate form.

Fig. 3.2.2. Complementary CMOS XOR gate.



(a). XOR gate with buffer output. (b). XOR gate without buffer output.



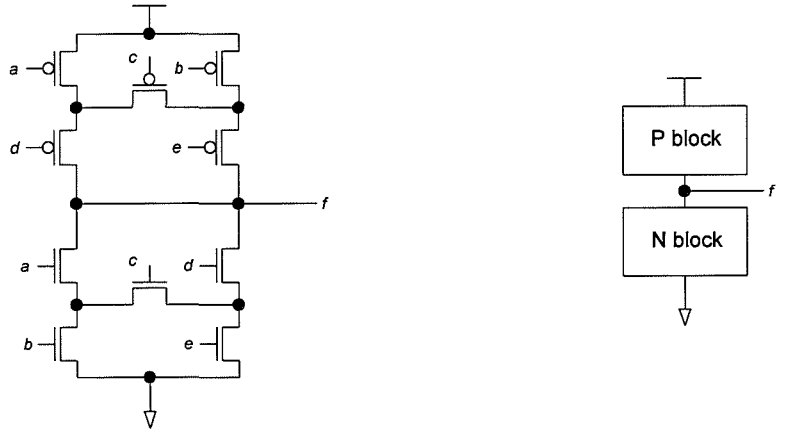
(c). XNOR gate with buffer output. (d). XNOR gate without buffer output.

Fig. 3.2.3. Pass transistor XOR and XNOR gates.

The simplest two input XOR gate known in complementary CMOS requires ten transistors, it is shown in Fig. 3.2.2. In many practical applications, two kinds of X(N)OR gates with pass transistors are widely employed, which are shown in Fig.

3.2.3. If both true and complemented forms of each variable are available, this is called double rail logic, less transistors can be used to construct an X(N)OR gate[Weste 93]. In the following, it is assumed that single rail logic is available, i.e., only the true form for each variable is available as a primary input unless otherwise stated.

Both AOI gate and OAI gate are a combination of AND and OR with an INV, which are not simply constructed at the gate level, but at the circuit level (transistor level). The former is used to realize a logic function in SOP form, and the latter is used to realize a logic function in POS form. A more complex gate can be constructed by a combination of transistors in series and parallel with bridging connections as shown in Fig. 3.2.4 (a)



(a). A complex gate. (b). Block diagram.

Fig. 3.2.4. A complex gate implementation in CMOS circuit for the function in equation (3.2.1).

In Fig. 3.2.4 (a), the transistors controlled by variable *c* perform bridging connections. The function of the circuit in (a) is

$$f = \overline{ab + ace + bcd + de} \tag{3.2.1}$$

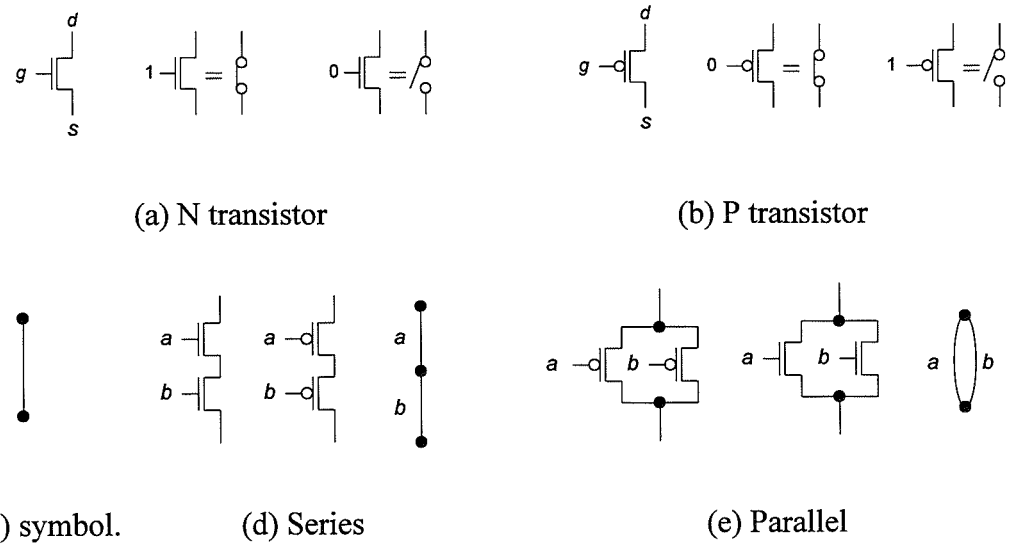
From this example, it is easily seen that the implementation of a logic function by using a complex gate, sometimes, is much simpler than that by using individual gates.

3.3 Switching Network Theory

Switching network theory[Wu 85, Maziasz 87, Dagenais 91, Zhu 93], a graphical form of Boolean logic, is a systematic approach to the design of MOS circuits. The resulting circuits contain a network of transistors in parallel and series; sometimes, they may generate bridging connections. This means that any single complex gate can be designed by this theory.

Switching network theory was developed from the earlier relay circuits[Kohavi 78], in which, a relay can be considered as an ideal bilateral switch. Since MOS circuits have been used in logic circuits, the switch concept is accepted in the area of logic circuit design because a transistor in MOS circuit is similar to a bilateral switch[Wu 85].

In *switching network theory*, a transistor in MOS circuits, the non-ideal switching component, is modelled as a voltage-controlled switch, which is illustrated by Fig. 3.3.1



(a) N transistor (b) P transistor

(c) symbol. (d) Series (e) Parallel

Fig. 3.3.1. Representations of *switching network theory*.

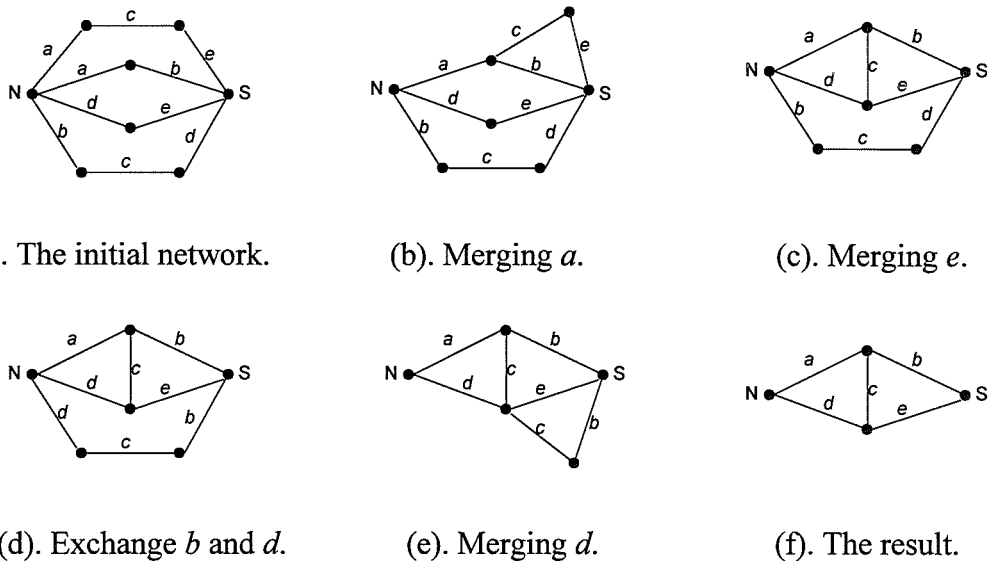
Switching network theory is a graph-oriented method. It employs a topology approach to simplify a MOS circuit. In this theory, each transistor (N type or P type) is

represented by one edge corresponding to a literal in a Boolean function. Each edge is labelled by the name of the input variable on the gate of the transistor it represents.
Initially, a network graph is derived from a Boolean function which, in general, is in SOP form. This graph is composed of edges and nodes, each edge represents a transistor with its gate control variable, and each node indicates that: (1) two transistors connection, (2) Transistor to VDD or GND, (3) transistor to a variable.
An edge-merging procedure is used to merge any pair of edges that have the same variable and the same node in the initial network. A check is then conducted to see whether this merging is valid or not. One way of checking is by tracing all paths from one terminal to the other[Kohavi 78]. A valid merge should not change the logic function represented by the initial network. If the merging is valid, then the pair of edges become one; otherwise, the merging is not performed. Clearly it is a *trial and error* procedure.
For many circuit styles, such as NMOS, dynamic CMOS, or pseudo NMOS, etc., a network for the N block will be processed. For a logic function to be implemented in the complementary CMOS circuit, two networks, one for the N block and its complemented network for the P block, are required. These two networks correspond to two logic functions that are complementary to each other.
An example is employed to illustrate this procedure,

Example 3.3.1:
Equation (3.2.1) is rewritten in standard SOP form, it is

$$\overline{f} = ab + ace + bcd + de \tag{3.3.1}$$

its switching network is represented in Fig. 3.3.2. N and S in Fig. 3.3.2 are called terminals, which correspond to VDD, GND, or a common point for output. In (a) of Fig. 3.3.2, each product term forms a path between the terminals N and S, this is an initial network. For a path, the order of edges can be changed, this is because the order of literals in a product term can be changed. In (b), two *a* edges are merged, and in (c), two *e* edges are merged. In (d), *b* and *d* are exchanged, and in (e), two *d* edges are merged. After two *c* edges and two *d* edges are merged, the final network is shown in (f).



(a). The initial network. (b). Merging *a*. (c). Merging *e*.

(d). Exchange *b* and *d*. (e). Merging *d*. (f). The result.

Fig. 3.3.2. An example for edge-merging procedure.

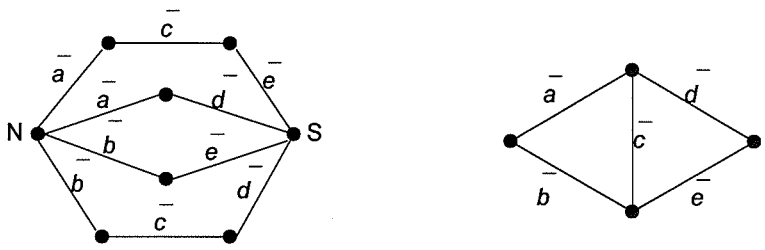
It is easily seen that it is impossible for the final network to be further minimized, since it contains a single edge for each variable. This means that it is the minimum network. The final network, then, is converted into a MOS circuit by substituting a transistor for each edge in the network, which corresponds to the N block in (a) of Fig. 3.2.4, it contains a transistor controlled by variable *c* in a bridging connection.

As stated above, if the circuit under consideration employs NMOS, dynamic CMOS, or pseudo NMOS, etc., the minimization is finished. For the complementary CMOS circuit, another network for the P block should be obtained. This network for the P block is derived from the complementary function of a logic function. Since the circuit under consideration is in single-rail logic, i.e., only the non-complemented form for each variable is available, thus, the complementary function of equation (3.3.1) is

$$f = \bar{a}\bar{d} + \bar{a}\bar{c}\bar{e} + \bar{b}\bar{c}\bar{d} + \bar{b}\bar{e} \qquad (3.3.2)$$

It can be found that in equation (3.3.1), each variable appears in its true form, and in equation (3.3.2), each variable appears in its complemented form. In this way, it can reduce the number of INVs required for the primary inputs, because the same working state for P type transistor and N type transistor is controlled by the opposite voltage.

Similarly, the initial network is shown in (a) of Fig. 3.3.3, and the final result is in (b), which is converted into the P block in (a) of Fig. 3.2.4.



(a). The initial graph. (b). The result.

Fig. 3.3.3. The complementary graph.

A complex circuit can be always decomposed into some simpler series-parallel circuits. A bridging connection may be considered a special series circuit, i.e., a transistor is shared by more than one path. A transistor shared by more than one path is not equal to a bridging connection. For example, in (f) of Fig. 3.3.2, *a* is shared by two paths, one is *a-b*, the other is *a-c-e*, and *a* is not a bridging connection.

From *switching network theory*, it is seen that Boolean logic can be applied to the design at the circuit (transistor) level in MOS circuits, in which the AND operation is explained by a series circuit and the OR operation is explained by a parallel circuit.

Another advantage of Boolean logic can be seen from *switching network theory*, that is, although the Boolean expansion may be minimal, the corresponding series-parallel network may still be further simplified, i.e., generating a bridging connection (or so called non-series-parallel connection). This can be illustrated by continuing the previous example. The minimal expansion of equation (3.3.1), in terms of the number of literals, is written in multi-level form, thus

$$\bar{f} = ab + ace + bcd + de = a(b + ce) + d(e + bc) \tag{3.3.3}$$

It requires eight transistors in N block to implement equation (3.3.3), but in practice, it only requires five N transistors to implement the final network in Fig. 3.2.4.

X(N)OR operation can not be simply explained by one of the two basic circuit principles, series and parallel, this is why an X(N)OR gate is more complex than (N)AND and (N)OR gates in nearly all circuit technologies. Therefore, *switching*

network theory is not very suitable for RM logic. This means that, RM logic can not be used in transistor level design to the same degree in which Boolean logic can.

3.4 Some Techniques for Fast MOS Circuits

In order to design a high speed MOS circuit, there are some techniques that should be considered. Although this work is concerned mainly with the issue of logic circuit minimization which attempts to eliminate the unnecessary components for correct logical operation, the logic circuit design can be better understood by briefly reviewing some of these main techniques.

(a). Redundant Logic

Some extra components may be added to a logic circuit to improve its speed of operation. Because these extra components are not necessary for correct logical operation, they are termed redundant logic. For instance, the carry lookahead circuit, which can speed up the operation speed of a carry ripple adder, is redundant logic. That is, for some logic circuits, redundant logic increases the number of components, but reduces the delay.

(b). Dynamic style

MOS circuits have associated capacitance which gives the MOS gate a storage capability, which can be exploited if a design can guarantee that all storage nodes in the gate are refreshed within a defined time period. This is the so called dynamic CMOS circuit. Dynamic CMOS circuits can reduce the number of P transistors (pull-up) and increase speed, because only the faster N transistor network is used to estimate logic values.

Since clock signals are introduced in dynamic circuit design, it may be more complex as compared to static circuit design.

(c). Inverting Logic

A single MOS gate is usually inverting. For example, an AND gate can be divided into a NAND gate with an INV. Therefore, the delay of an AND gate is roughly equal to the delay of the NAND gate plus the delay of an INV. This also applies to OR, AOI, OAI, and many other complex gates. It can be readily seen that employing inverting logic, in many situations, not only increases the performance speed of a logic circuit, but also reduce the number of components required.

In general, RM logic can handle *inverting logic* better than Boolean logic, because the electrical polarity of a X(N)OR gate is easy to adjust, as described in chapter 2.

(d). Transistor Size Optimization

Optimizing transistor widths in a circuit can increase the speed of the circuit. The effect of the width of a certain transistor depends on the position of the transistor in the circuit and on the sizes of all other transistors in the circuit, and this effect also varies depending on the fabrication technology used. The size of each transistor can be calculated to satisfy the requirements in a design. This technique requires careful and computationally intensive simulation because the parameters can not easily be isolated[McAuley 92].

(e). Transistor Reordering

The order of transistors in a MOS logic gate can have a significant effect on the propagation delay of the gate. Therefore, a discretional placement of devices in a MOS logic gate could lead to poor performance or unnecessary transistor sizing. In a MOS gate, finding a good transistor order can decrease the delay of the gate. Unlike transistor size optimization which normally improves speed at the expense of some additional area, this technique achieves significant reduction in propagation delay with little effect on layout area[Carlson 93].

A general rule is that, the most capacitive nodes are positioned as near as possible to VDD and GND.

(f). Asynchronous Logic Circuit

In synchronous circuit design, the capacitive load of the clock buffer increases as the size of a VLSI chip increases, which not only slows the system, but also increases the power consumption. The improvement of the speed in synchronous circuit design has become complex since a large chip area may yield a significant skew for the clock signal. An asynchronous circuit technique, sometimes called self-timed circuit, has been developed to solve these problems. It is worth mentioning that *Asynchronous Logic Circuit* is not as well known or accepted[McAuley 92], because few successful examples can be found in the literature.

3.5 PLA Implementation

The Programmable Logic Array (PLA), which is specifically designed for random logic applications[Hurst 92], is a kind of structured circuit and widely used in practical applications. For PLA implementation, RM logic, in theory, may generate more economical circuits for many logic functions as compared to Boolean logic[Sasao 90, 93A, 93B].

For PLA implementation in Boolean logic, the criterion of minimizing a logic function is often considered to be the number of products. The number of literals is not important for a PLA implementation. This is because in a PLA, the implementation area of the AND plane is directly proportional to the number of products, and the implementation area of the OR plane is directly proportional to the number of outputs. This can be seen in Fig. 3.5.1, which is a simplified PLA architecture. For a fixed sized PLA to realize a logic function, absolute minimization may not be necessary provided the number of products is not greater than the device capacity.

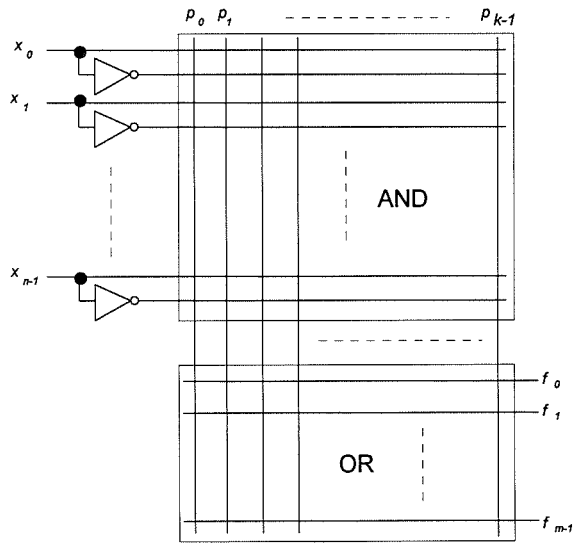


Fig. 3.5.1. A simplified PLA architecture for SOP.

Similarly, a simplified PLA architecture for ESOP can be drawn in Fig. 3.5.2., this kind of PLA is often called XPLA, i.e., XOR PLA.

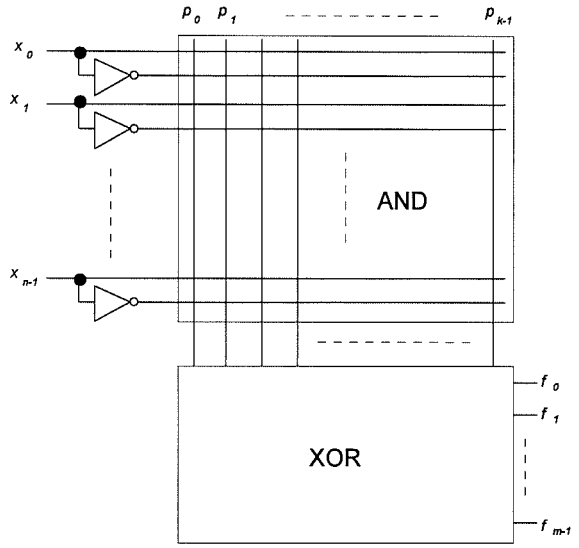


Fig. 3.5.2. A simplified XPLA architecture for ESOP.

When compared, the PLA and XPLA have identical architectures for realizing the AND plane, but the realization of the OR plane is different from the realization of the XOR plane. The former has a simple and regular layout structure corresponding directly to physical realization, it is hard to construct the latter as efficiently as the former. One possible layout strategy for the XOR plane is described by Saul et al[Saul 93], but it is not easily generalized, particularly for a larger circuit with multi-outputs. According to current technologies known in (X)PLA implementation, because the XOR plane for RM logic is not as simple and regular as the OR plane for Boolean logic, RM logic may be restricted in its application to a smaller range, in which, for a logic function, the number of products in ESOP form is obviously bigger than that in SOP.

Although PLA provides an excellent means to implement a logic function, large PLA structures are not necessarily desirable. Folded PLA can reduce implementation area. Unfortunately, the folding of different groups of variables interacts so that optimal PLA folding is a difficult problem[Geiger 90].

Note that even though PLA appears in a AND-OR form, in NMOS circuits, its practical implementation usually employs NOR-NOR logic, and in CMOS, it usually uses dynamic CMOS circuits or pseudo NMOS circuits. The complementary CMOS circuits are not feasible for PLA implementation.

3.6 Gate Implementation

Logic functions based on gate implementation are widely employed in practical logic design. The vendors supply various standard gate based chips, and these standard gates also appear in the standard cell environment in ECAD packages. So called standard gates mean that these gates are designed for general purpose, e.g., the ability of fan-out, and also, the gate types and their inputs are limited, e.g., a AOI or a OAI gate.

In most ECAD suites, each standard gate library corresponds to a specified fabrication technology, and an arbitrary gate like that in (a) of Fig. 3.2.4 can rarely be found in the library. It is possible for an arbitrary gate to be involved in several libraries[Detjens 87] or to be generated dynamically[Maziasz 87], which should be classified into the design of transistor level or layout level.

Because logic design is always technology-dependent, it is hard to compare without considering the technology used. In the following, some of the most commonly used technologies are discussed.

(a). Implementation based on binary tree form:

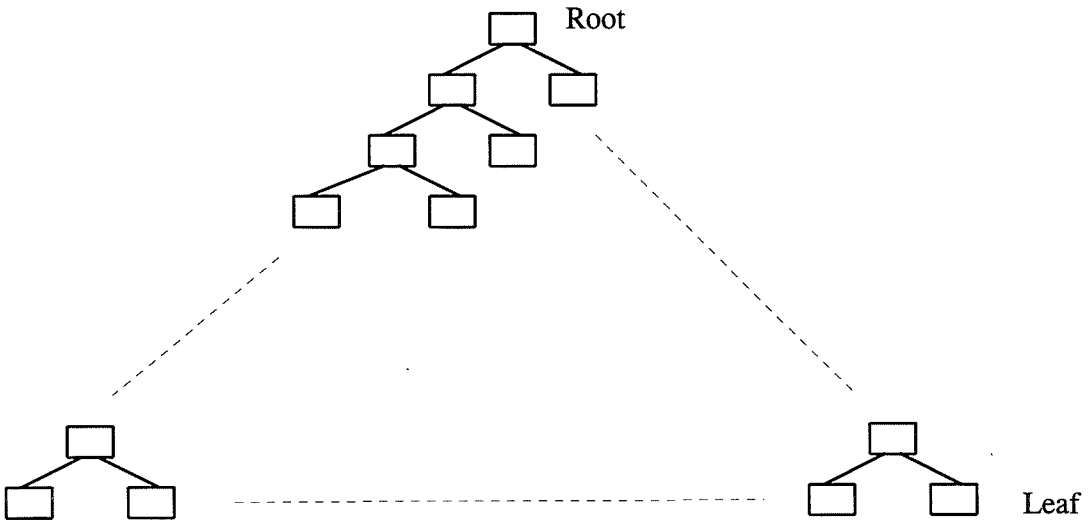


Fig. 3.6.1. Implement logic functions in binary tree form.

This is a simple technology mapping strategy, and can be described by a binary tree. A binary tree is a special case of directed acyclic graph (DAG), in which, the fan-in of each component is restricted to two, and it is shown in Fig. 3.6.1. This kind of implementation is also called a circuit tree[Green 86]. In RM logic, three types of circuit elements and their costs are used to estimate the implementation complexity for a given function[Green 86, 91B], they are:

1. the inverter	1 cost unit
2. the two input AND gate	2 cost units
3. the two input XOR gate	3 cost units

In practice, the leaf cells in Fig. 3.6.1 can be realized by employing multi-input AND gates. Based on the cost for the two input AND gate, the cost for a three input AND gate may be defined as 3 units and the cost for a four input AND gate may be defined as 4 units respectively. This kind of implementation can be categorized into the conventional individual gate implementation. If this conventional implementation method for RM logic is used to realize RM functions in two level representation with fixed polarity, the resulting circuits are considered very easy to test[Reddy 72].

As RM logic in the conventional implementation is limited to using few kinds of gates, for many functions, it may make RM logic hard to compare with Boolean logic in terms of the gate costs, since Boolean logic can use more kinds of gates, including some highly efficient gates, such as AOI and OAI. When a RM function is only restricted in a two level form with fixed polarity, it usually needs more hardware to realize the function than that in a multi-level form with mixed polarity. Consequently, a trade-off between the implementation complexity and the testability should be decided in a practical application.

It should be mentioned that because the logic functions in RM logic are usually more compact[Sasao 90, 93A, 93B], it is believed that there are still some functions that are realized in the conventional implementation of RM logic better than that in Boolean logic, in terms of the implementation complexity.

Because the gate costs discussed here can't be used to estimate the implementation cost accurately without regard to actual target technology, the comparison of implementation cost between RM logic and Boolean logic will be carried out in the next section.

(b). Implementation based on gate array and SOG:

Gate array and Sea of Gates (SOG) are popular styles in use for implementation of general logic functions. Both of them have a common point, that is, their structures are

the core of the chip that contains a continuous array of N and P transistors. A circuit is achieved by using design-specific metalization and contacts. The main difference between gate array and SOG is that they use different strategies to connect these fixed transistors. For example, wiring in an SOG chip occurs over the top of unused transistors, while in a gate array the routing is constrained to a routing channel[Weste 93].

In gate array and SOG, the logic synthesis, in fact, is carried out more at the transistor level or the layout level rather than at the gate level. An arbitrary complex gate without a break in diffusion, termed sharing diffusion, may be constructed. Therefore, the comparison of implementation cost between RM logic and Boolean logic in gate array and SOG can be also carried out in later sections.

(c). Implementation based on FPGA:

Field programmable gate array (FPGA) combines the advantages of both programmable logic and the gate array[Almaini 94], and is a common and fast means to implement logic functions. There exist some different kinds of FPGA products that may have different characteristics.

Consider the architecture of the Xilinx 2000 logic cell array (LCA)[Detjens 90] as an example. Unlike the fixed PLA structure, the structure of Xilinx 2000 is flexible and multi-level. In the Xilinx 2000 logic cell array, a four-input XOR gate uses the same space and is as fast as a four-input AND gate, because it is a table look-up based FPGA[Csanky 93]. It is believed that realizing a function in RM logic is usually more economical than that in Boolean logic. Other products that favor the implementation in RM logic are mentioned by Csanky et al[Csanky 93].

Another example of FPGAs is the product of Signetics/Philips Components Ltd., which uses an AND gate based structure[Hurst 92]. It is not suitable for the implementation in RM logic, but for a NAND-NAND implementation in Boolean logic.

3.7 Transistor Implementation

Implementation cost of logic circuits at the circuit level is usually estimated by counting the number of transistors required, for this reason, the circuit level is often termed the transistor level. Although the most compact logic functions, which are often measured by the number of products or the number of literals, are always desirable in logic synthesis, sometimes, the most compact logic functions do not guarantee generating the simplest circuits measured by the number of transistors. This

is mainly because the complexity of some kinds of gates in a logic representation doesn't exactly correspond to the complexity of circuits. This issue varies depending on the logic functions under consideration. In the following, two typical examples are used to illustrate this and explain the difference between RM logic and Boolean logic. Assume that the functions in their mixed polarity have been minimized before they are implemented, CMOS static circuits are used, and the XOR gate in Fig. 3.2.3 is employed, i.e., a single XOR gate with six transistors. As stated earlier, it is hard to find a highly efficient multi-input XOR gate, therefore, it is assumed that several two input XOR gates in a tree connection are employed to implement a multi-input XOR gate. Before showing and discussing these examples, for convenience, meanings of some symbols should be given first.

f_B function representation based on Boolean logic

f_R function representation based on RM logic

N_p is the number of products;

N_l is the number of literals;

N_g is the number of gates;

N_t is the number of transistors in two-level implementation;

N_{tm} is the number of transistors in multi-level implementation;

$T_{\max}$ is the worst performance time (simulated using Hspice simulator with Cadence 2.4μ library).

Example 3.7.1.

A majority function (the carry function of a full adder) is a three variable function, it is expressed as

$$f = \sum m(3,5,6,7) \tag{3.7.1}$$

its map pattern is

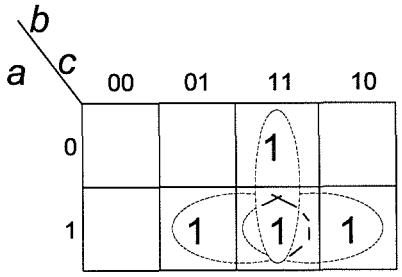


Fig. 3.7.1. Majority function.

From Fig. 3.7.1, the absolute minimal Boolean expansion and RM expansion in two-level representation are

$$f_B = ab + ac + bc \tag{3.7.2}$$

$$f_R = ab \oplus ac \oplus bc \tag{3.7.3}$$

It is interesting to find these two expansions are similar, and the difference is that one uses OR operators to connect products and the other uses XOR operators to do that. This is because all circles in Fig. 3.7.1 conform to both the Boolean rules and the RM rules.
The implementations are:

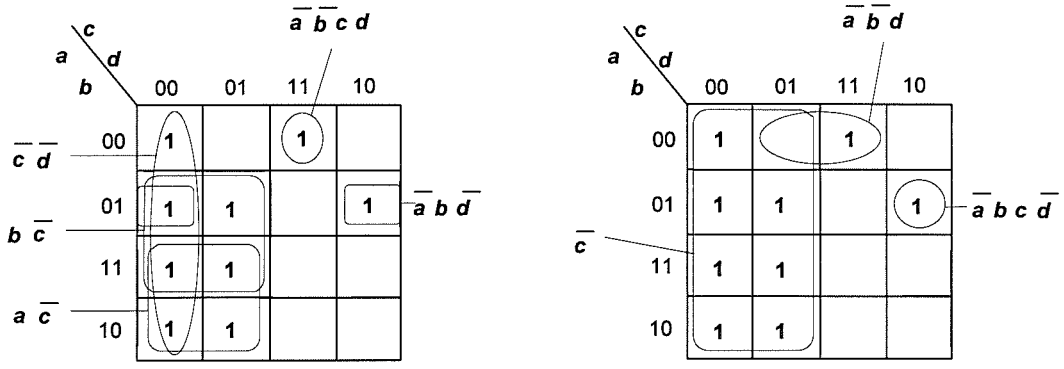
Boolean domain	RM domain
N_p 3	N_p 3
N_l 6	N_l 6
N_g 1 AOI, 1 INV	N_g 3 AND2s, 2 XORs
N_t 14	N_t 30
N_{tm} 12	N_{tm} 24

From this example, it is seen that a logic function has the same number of products and the same number of literals in both Boolean logic and RM logic, the implementation in Boolean logic is much more economical than that in RM logic, because Boolean logic can employ an arbitrary complex gate, and RM logic only uses individual gates with a complex XOR gate.

Example 3.7.2.
A logic function from[Tran 89] which was used to illustrate a good example for RM logic to minimize logic functions, it is a four variable function and is expressed as

$$f = \sum m(0,3,4,5,6,8,9,12,13) \tag{3.7.4}$$

its map patterns are



(a). Boolean domain. (b). RM domain.

Fig. 3.7.2. An example from [Tran 89].

From Fig. 3.7.2, it is not difficult to find that this function is more easily minimized in the RM domain than in the Boolean domain. Their logic representations in two level form are

$$f_B = \bar{c}\,\bar{d} + b\,\bar{c} + a\,\bar{c} + \bar{a}b\,\bar{d} + \bar{a}\,\bar{b}cd \tag{3.7.5}$$

$$f_R = \bar{c} \oplus \bar{a}\,\bar{b}d \oplus \bar{a}bc\,\bar{d} \tag{3.7.6}$$

The implementations are:

Boolean domain	Reed-Muller domain
N_p 5	N_p 3
N_l 13	N_l 8
N_g 1 AOI, 5 INVs	N_g 1 AND3, 1 AND4, 2 XORs, 4 INVs
N_t 36	N_t 38
T_{max} 3.5 ns	T_{max} 2.4 ns

From the implementation in the Boolean domain, it is seen that the performance time $T_{\max}$ becomes much slower as the number of transistors in series increases, because a single AOI gate is used to realize this function. Therefore, a logic function with products (or literals) of more than a certain value should be partitioned into groups and realized. In general, the number of transistors in series is restricted to three or four if high speed circuits are desirable.

In addition, equation (3.7.5) can be expressed in a multi-level form to reduce N_l. Thus, equation (3.7.5) is rewritten as

$$f_B = \overline{(\bar{c}\bar{d} + b\bar{c} + ac)} \cdot \overline{(\bar{a}b\bar{d} + \bar{a}bcd)} = \overline{\bar{c}(\bar{d} + b + a)} \cdot \overline{\bar{a}(b\bar{d} + \bar{b}cd)} \tag{3.7.7}$$

From equation (3.7.7), it can be seen that the number of literals N_l is 10, i.e., it is reduced by 3 in the multi-level representation. Equation (3.7.7) is decomposed into two parts which can be realized by two complex gates instead of one, and a NAND gate is employed to connect these two complex gates. The circuit is shown in Fig. 3.7.3. The total number of transistors used is N_{tm}=32, in which, 8 transistors are used for 4 INVs that are not shown in Fig. 3.7.3

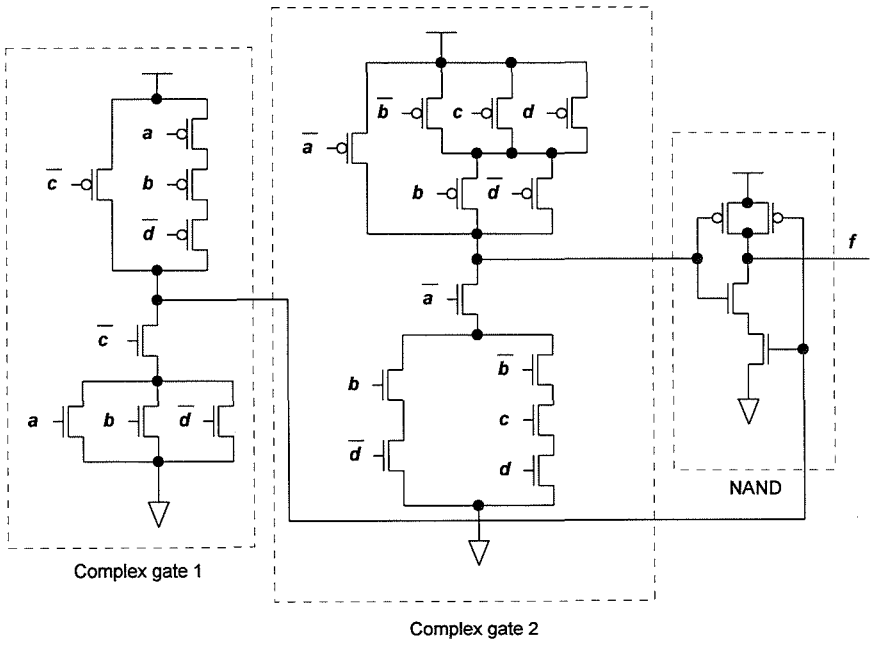


Fig. 3.7.3. Implementing equation (3.7.7).

This circuit was also simulated and T_{max} was 2.1 *ns*. It can be seen that this realization not only reduces the number of transistors, but also improves the speed.

In contrast, the implementation for equation (3.7.6) is illustrated in Fig. 3.7.4, in which, only logic gates are shown, because the conventional implementation method in RM logic is generally gate based implementation. The number under each gate indicates the number of transistors to realize this gate. In total, 38 transistors are required.

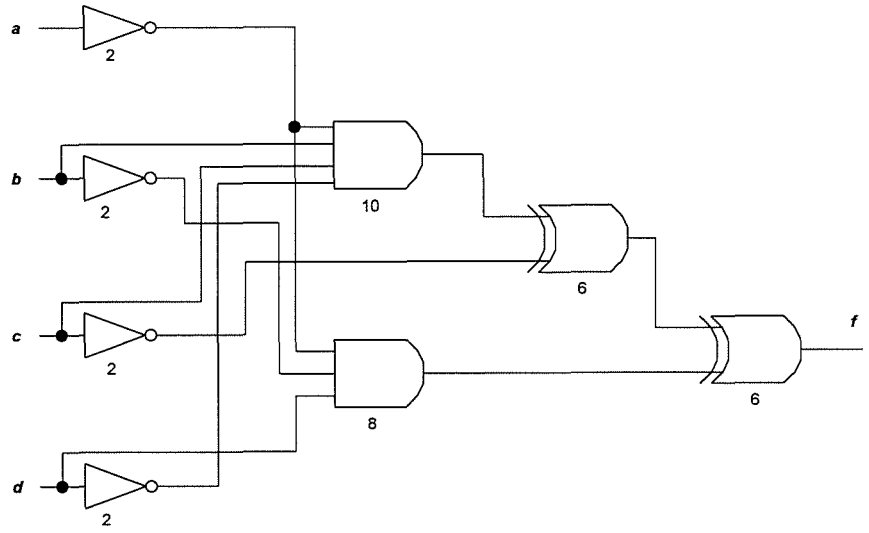


Fig. 3.7.4. The implementation for equation (3.7.6).

Similarly to the case in the Boolean domain, the multi-level representation for equation (3.7.6) is also considered, which can reduce the number of literals. Therefore, equation (3.7.6) is rewritten as

$$f_R = \bar{c} \oplus \bar{a}(\bar{b}d \oplus bc\bar{d}) \tag{3.7.8}$$

It is seen that, 40 transistors are required to implement the function in multi-level form. Although the multi-level representation contains less literals than the two level representation does, see equation (3.7.6), the implementation for the multi-level representation requires more transistors. The implementation in multi-level form is illustrated in Fig. 3.7.5

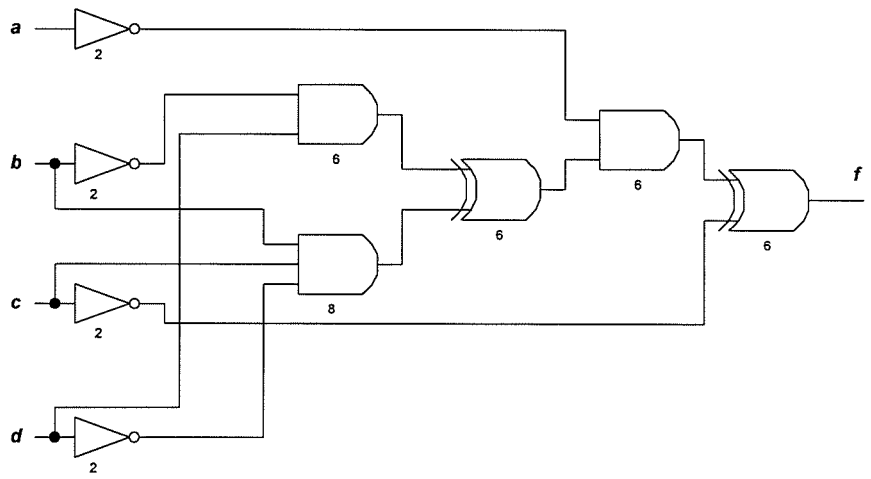


Fig. 3.7.5. The implementation for equation (3.7.8).

When the two circuits in Figs 3.7.4 and 3.7.5 were simulated T_{max} for the two circuits were 2.4 *ns* and 3.0 *ns,* respectively.

3.8 Mixed Representations Based on RM logic

From *example 3.7.2*, it is interesting to know that, although a function in the RM domain looks significantly more compact than its counterparts in the Boolean domain, if it is realized by using the conventional implementation method, the resulting circuit is still more complex than that in the Boolean domain. This is firstly because more complex XOR gates are required in RM logic, and secondly because the conventional implementation in RM logic is normally restricted to using a smaller gate set that only includes INV, AND, and XOR due to testability considerations.

In practice, more economical NAND gates can be employed to replace AND gates. This is because in a RM function, even number of terms can be complemented at the same time and the validity of the function is not changed. This property is easily derived from equation (2.2.8). The resulting circuit can retain the important characteristic of RM logic, the good testability. This will be further discussed in section 3.10.1.

Furthermore, if only the minimum circuit is desirable, all gates used in Boolean logic, even including an arbitrary complex gate, may be employed to realize a RM function. This is based on equation (2.2.12), in which, it is shown that if two terms are relatively exclusive, then, an XOR operator can be replaced by an OR operator. This

can be illustrated by one of the previous examples. For instance, equation (3.7.8) is rewritten as

$$f_R = \bar{c} \oplus \bar{a}(\bar{b}d \oplus bc\bar{d}) = \bar{c} \oplus \bar{a}(\bar{b}d + bc\bar{d}) = c \oplus \overline{\bar{a}(\bar{b}d + bc\bar{d})} \tag{3.8.1}$$

From equation (3.8.1), it is readily seen that, firstly, an XOR operator is replaced by a OR operator because the term $\bar{b}d$ and the term $bc\bar{d}$ are relatively exclusive; and secondly, the bar above variable c is moved onto $\bar{a}(\bar{b}d+bc\bar{d})$ according to equation (2.2.9). This also illustrates that it is easy to adjust the electrical polarity of a circuit in RM logic. Realizing equation (3.8.1) only requires 24 transistors, this is shown in Fig. 3.8.1. Six transistors are used for INVs, not included in Fig. 3.8.1.

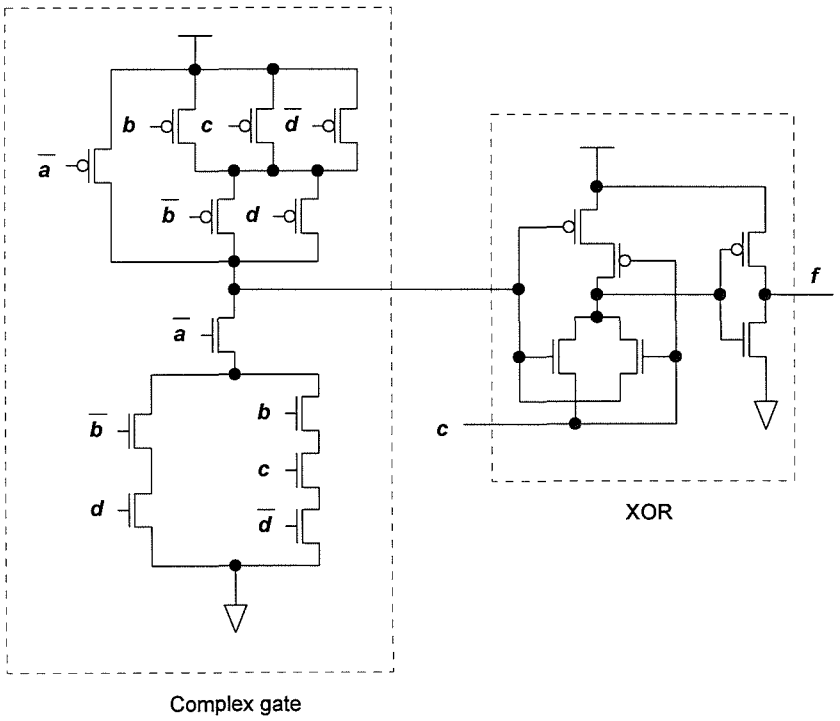


Fig. 3.8.1. A circuit to perform equation (3.8.1).

Compared with the previous implementation in Fig. 3.7.4, which requires 38 transistors, it significantly reduces the number of transistors. The circuit is even simpler than that based on Boolean logic, which needs 32 transistors, see Fig. 3.7.3,

and it also has roughly the same speed as its counterpart realized in Boolean logic, since the gates used by these two circuits in their critical paths are similar.

The author's experience shows that, in many RM logic functions, at least 50% XOR operators can be replaced by OR operators. In this way, many RM functions may be realized more economically than their counterparts in the Boolean domain.

Now, it is not difficult to find that, a RM function can be implemented by employing all gates used in Boolean logic if the testability is not considered. Because the function is first minimized in the RM domain and then the Boolean techniques are employed to minimize the final implementation, equation (3.8.1) is called the mixed representation based on RM logic[Guan 94A]. The mixed representation implementation is more economical than the conventional method based on the individual gate implementation.

It should be mentioned that a more general mixed representation was suggested by Saul[Saul 92], that is, a logic function is originally minimized by combining both Boolean techniques and RM techniques. Its result should be better than that based only on Boolean techniques or that based only on RM techniques. In practice, a systematic and efficient algorithm for minimizing logic functions by combining both Boolean techniques and RM techniques is still unknown, even for some special form of logic functions[Dubrova 95].

3.9 Layout Evaluation

Layout synthesis can only reduce the area for a given circuit, and it can't simplify the circuit further. Therefore, before layout, circuits should be simplified as much as possible.

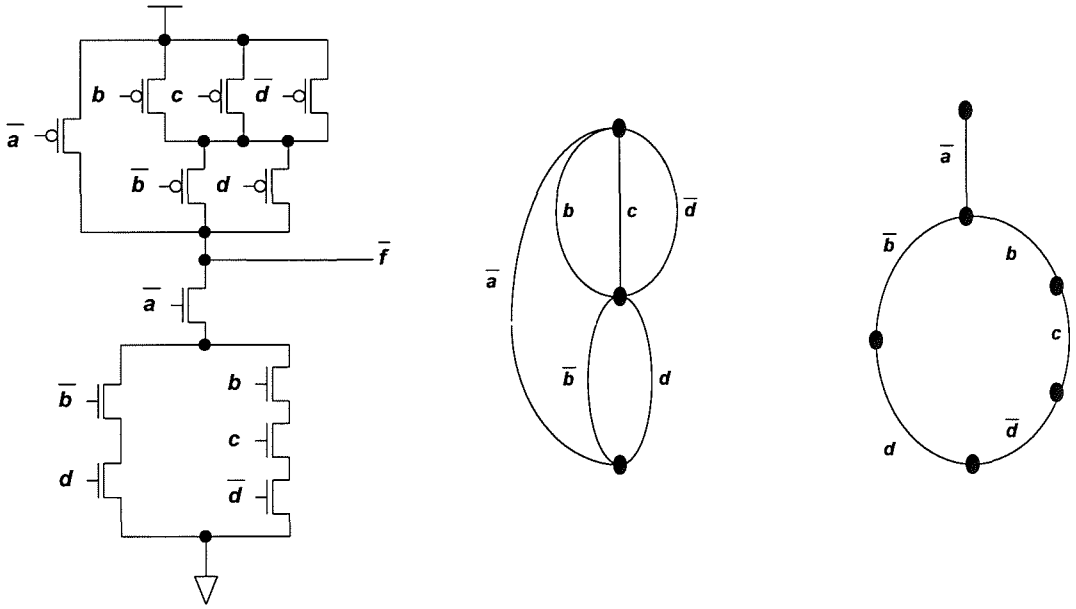
Layout synthesis is a very complex procedure and it varies depending on application. For instance, the channel width of a transistor on a non-critical path can be reduced to a smaller size, and multi-metal layers are used to improve the ability of global connections, etc., these are generally independent of logic representation. In here, only the factors related to logic representation will be considered. A well known factor concerned with this problem is sharing diffusion, which is discussed in the following.

In general, in order to achieve minimum layout, the designer has to share as much diffusion area as possible[Chen 87], and this is achieved by finding the same layout sequence for P type transistors and N type transistors in a complex gate. A similar graphical method to *switching network theory* is employed to find a good sequence of transistors for layout, and the starting point is two dual graphs for two dual minimized networks, which correspond to a Boolean function and its dual function.

An example is used to illustrate sharing diffusion for layout synthesis. A given Boolean function is

$$\overline{f} = \overline{a}(\overline{b}d + bc\overline{d}) \tag{3.9.1}$$

This function is a sub-function of equation (3.8.1). The circuit to perform equation (3.9.1) is the complex gate of Fig. 3.8.1. In Fig. 3.9.1, the circuit and its graphs for P network and N network are shown



(a). Circuit. (b). P network. (c). N network

Fig. 3.9.1. the circuit to perform equation (3.9.1) and its graphs.

In order to reduce the layout area, it is necessary to find a pair of paths on the dual graphs with the same sequence of labels which represent transistors. This path is often termed an Euler path[Maziasz 87], and the pair of paths are called a dual Euler path. An Euler path means that a path goes through each edge once and only once in a graph. From the dual graphs in Fig. 3.9.1, the sequence of labels in a possible dual Euler path is

$$\bar{a}-b-c-\bar{d}-d-\bar{b}$$

According to this sequence of transistors, a so called functional cell layout style[Maziasz 87], which is an efficient approach to realizing the layout of a logic function in automatic design in the standard-cell style, is employed to demonstrate sharing diffusion. In a functional cell, the cell's height is assumed to be fixed by technology consideration, its width can be minimized by ordering the transistors in the layout so that chains of transistors can share common diffusion regions[Uehara 81, Chen 87, Maziasz 87]. Complementary transistor pairs are vertically aligned in the layout. This allows their gate terminals to be connected by vertical polysilicon columns without the use of crossovers. The symbolic layout is seen in Fig. 3.9.2

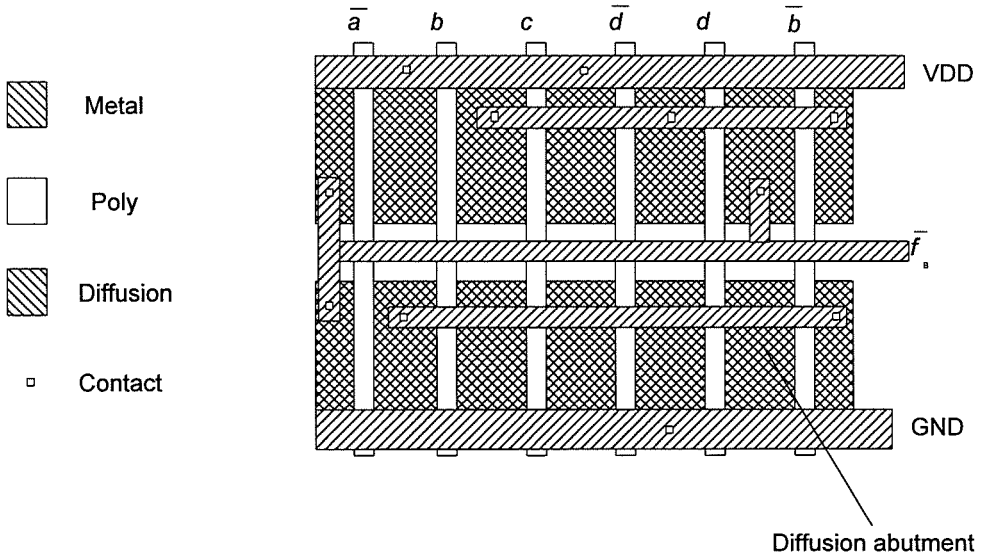
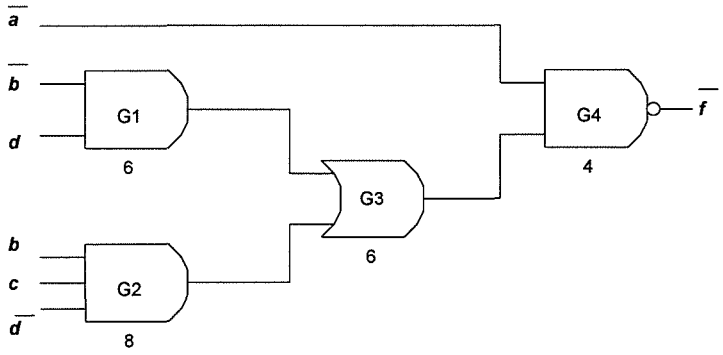


Fig. 3.9.2. Layout in a functional cell.

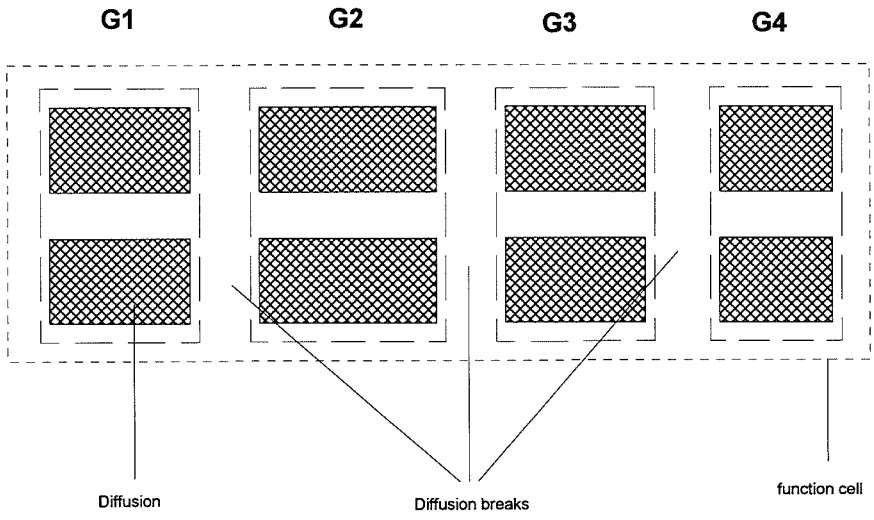
From Fig. 3.9.2, it is seen that the source/drain connections are achieved by sharing diffusion. In result, there is no break in the diffusions, which leads to a smaller layout area, because a break in diffusion requires more space. Although only the layout in a functional cell is shown here, the basic principle of sharing diffusion is suitable for many situations, for example, the layout of gate array or SOG for a given gate is similar to that of functional cells[Weste 93]. In addition, full custom design is also classified into this category. The main difference is that the restriction to fixed height is also relaxed and the size of diffusion can be decided depending on application.

Here, only a simple logic function in Boolean logic is employed to show sharing diffusion. In practice, a more complex gate than that in Fig. 3.9.1 can be realized by sharing diffusion without a break[Maziasz 87]. For some complex gates, there may not be a single Euler path, then the graph can be decomposed into several sub-graphs which have Euler paths. In such cases, a minimum set of dual Euler paths should be found to achieve optimal layout.

In order to understand the layout synthesis of sharing diffusion better, a layout based on individual gate implementation for the same function as above is shown in Fig. 3.9.3



(a). Circuit.



(b). Layout, only diffusions are shown.

Fig. 3.9.3. A layout based on individual gate implementation.

From Fig. 3.9.3, it can be easily seen that the implementation based on individual gates not only requires more transistors, but also yields some diffusion breaks which need more area for the layout.

From the above analysis, it can be seen that, because the conventional implementation in RM logic is similar to that based on individual gates, the approach to sharing diffusion is hardly applied to it unless the mixed representations are used, as described previously. In this way, RM logic may lose one of its main advantages, ease of testing. In contrast, a logic function realized in a Boolean expansion can have a better sharing of diffusion. This is because, to design logic circuits in a Boolean expansion, an arbitrary complex gate can be employed, and realizing a given logic function with a complex gate usually has better sharing potential than realizing the same function with individual gates. Moreover, a single dual Euler path may be found for a single complex gate. For instance, Fig. 3.9.3 shows the same function implemented by individual gates as that in Fig. 3.9.1, implemented by a single complex gate.

Implementing logic functions employing complex gates, compared to traditional individual gate implementation, can reduce the layout size by 20%[Chen 87].

3.10 Testing of RM circuits

It is well known that good testability is one of the most important merits in RM logic. Perkowski and Jeske state that the main advantage of a logic function realized with the RM expansion is its essentially improved testability[Perkowski 90]. In addition, Helliwell and Perkowski conjectured that the gains from easy testing may even exceed possible disadvantages in such cases where the XOR realization is more costly than SOP[Helliwell 88].

Reddy is considered to be the first author to present the testability properties of RM circuits[Reddy 72]. The basic concept used by Reddy is applied to many other researchers' work[Bhattacharya 85, Damarla 89, Sarabi 93]. Reddy investigated two level RM circuits in fixed polarity form and presented a simple approach to test these circuits. The results obtained by Reddy for testing single stuck-at-faults, stuck-at-0 (s-a-0) or stuck-at-1 (s-a-1), are summarized as follows

(a) If the primary input leads are fault-free, then there exists a realization for an arbitrary n-variable logic function that requires a fault detection test set with only n+4 tests and this test set is independent of the function being realized.

(b) If the primary input leads could be faulty, then only n+4+2 n_e tests are required for detecting faults, where n_e is the number of primary inputs appearing in an even

number of product terms in the Reed-Muller expansion for the function being realized.

(c) If the primary input leads could be faulty, then by adding an extra observable output and an extra AND gate, n+4 tests of (a) will be sufficient and these tests will again be independent of the function being realized.

Assume that an arbitrary n variable logic function $f(x_{n-1},...,x_1,x_0)$ is realized in circuit and its primary input leads are fault-free, the following four test patterns are applied to the inputs of the circuit

$$T_1 = \begin{array}{c} \begin{matrix} x_{n-1} & .. & .. & .. & ... & x_0 & y \end{matrix} \\ \begin{bmatrix} 0 & . & . & . & . & 0 & 0 & 0 \\ 1 & . & . & . & . & 1 & 1 & 0 \\ 0 & . & . & . & . & 0 & 0 & 1 \\ 1 & . & . & . & . & 1 & 1 & 1 \end{bmatrix} \end{array} \qquad \text{4 test patterns} \qquad (3.10.1)$$

where y is fixed to "0" or "1" as the circuit is in normal mode[Reddy 72]. [0...000] or [0...001] is used to detect a s-a-1 at the output of any AND gate. [1...110] or [1...111] is used to detect a s-a-0 fault at any AND gate input or output. A s-a-1 fault at any one of the inputs to the AND gates is detected by one of the n test patterns in the set T_2

$$T_2 = \begin{array}{c} \begin{matrix} x_{n-1} & . & . & . & . & . & x_0 & y \end{matrix} \\ \begin{bmatrix} 1 & . & . & . & . & 1 & 1 & 0 & d \\ 1 & . & . & . & . & 1 & 0 & 1 & d \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ 0 & 1 & . & . & . & . & 1 & d \end{bmatrix} \end{array} \qquad n \text{ test patterns} \qquad (3.10.2)$$

where d means "don't care", i.e., "0" or "1". Then a fault test set T is

$$T = T_1 + T_2 \qquad (3.10.3)$$

i.e., T=(n+4) is independent of the function $f(x_{n-1},...,x_1,x_0)$.

In order to detect the faults of primary input leads, $2n_e$ test patterns are required, where n_e is the number of variables appearing in an even number of product terms, which is not independent of the function $f(x_{n-1},...,x_1,x_0)$. For more detail, refer to [Reddy 72].

Later, some researchers developed Reddy's theory and extended it to bridging faults[Bhattacharya 85, Damarla 89]. Sarabi and Perkowski present a good survey for the testability of RM circuit[Sarabi 93].

In fact, it is found that the XOR gate plays a critical role in test, that is, a stuck fault at one input always propagates through a XOR gate, regardless the value on the other input. As stated in section 3.8.1, NAND gates can be employed to replace AND gates in the conventional implementation of RM logic, and the resulting circuit can retain the good testability. In the following, a simple example is used to demonstrate this.

A RM function is

$$f = 1 \oplus b \oplus ab \oplus bc = 1 \oplus b \oplus \overline{ab} \oplus \overline{bc} \tag{3.10.4}$$

its implementation is shown in Fig. 3.10.1

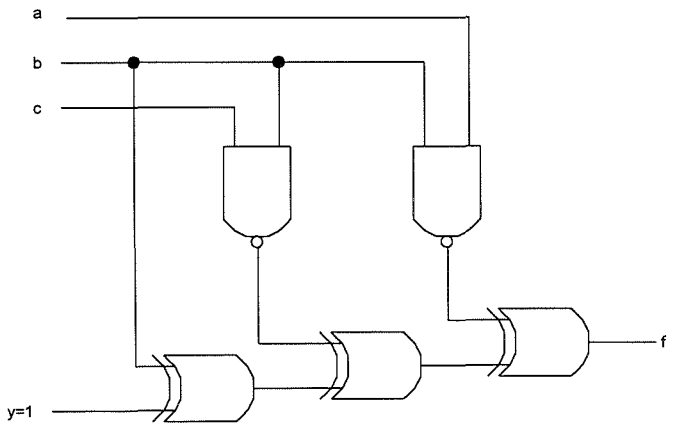


Fig. 3.10.1 RM circuit for equation (3.10.4).

its test patterns are:

$$T_1 = \begin{array}{c} \begin{matrix} a & b & c & y \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array} \tag{3.10.5}$$

where [0000] or [0001] is used to detect a s-a-0 at any NAND gate output; [1110] or [1111] is used to detect a s-a-0 at any NAND gate input and a s-a-1 at any NAND gate output.

$$T_2 = \begin{array}{c} \begin{matrix} a & b & c & y \end{matrix} \\ \begin{bmatrix} 1 & 1 & 0 & d \\ 1 & 0 & 1 & d \\ 0 & 1 & 1 & d \end{bmatrix} \end{array} \tag{3.10.6}$$

T_2 is used to detect s-a-1 at any one of the inputs to the NAND gates.
Because each variable in this example appears in an odd number of product terms, T_1 and T_2 are enough to test any single stuck fault in the circuit.

From the above example, it can be seen that it doesn't change the testability of an RM circuit when even number of AND gates are exchanged with NAND gates at the same time, and even the test patterns do not need to be modified. This conclusion is believed to be also suitable for the work done by Bhattacharya et al and Damarla and Karpovsky[Bhattacharya 85, Damarla 89], in which a bridging fault is considered for testing an RM circuit. This is because nearly all the strategies for testing an RM circuit are based on the basic principle that one change at any input of an XOR gate will affect its output.

3.11 Summary

In this chapter, the CMOS circuit implementations for both Boolean logic and RM logic are studied, ranging from a logic function to be realized in PLA form, to a logic circuit to be mapped into layout.

In general, the theory and application of Boolean logic is wider and more mature than RM logic. Boolean logic can be employed from logic level design to layout level design. At a lower level synthesis, such as transistor level or layout level, the minimization may not be directly achieved by the Boolean algebraic form, but by its graphical form, e.g., *switching network theory*. Boolean logic based on two basic operations, AND and OR, underlies its main advantage, that is, these two operations are directly mapped to series and parallel circuits. In contrast, the XOR operation can not be explained by simply employing series or parallel circuits, therefore, unlike Boolean logic, RM logic based on AND and XOR operations is not easily applied to a circuit level or layout level design.

Although RM logic can't be easily applied to a lower level design, it can be found that RM logic produces many efficient solutions that Boolean logic does not. Particularly, some mapping technologies based on look-up table may be very suitable for RM logic to implement. In addition, if the testability of a circuit is not considered at logic (gate) level, but at a high level, e.g., a functional level, the implementation cost may be largely reduced by substituting some of the XOR operators to OR operators and employing highly efficient complex gates. In this way, the realization of many logic functions in the RM domain is comparable with that in the Boolean domain.

100 four variable randomly generated functions are tested, the results are listed in Table 3.11.1

Table 3.11.1 Minimization results of randomly generated functions.

A	B	C
41	30	28

A-RM two level expansions with mixed polarity (from a RM logic synthesis program developed at Napier University.);

B-the mixed representations based on A;

C-Boolean expansions.

In Table 3.11.1, the numbers indicate the average number of transistors required in the CMOS circuit for a function. In this test, it is found that 33 of 100 functions in the

mixed representations require less transistors than their counterparts in Boolean logic, and only 8 of 100 functions in RM two level expansions with mixed polarity need less transistors than that in Boolean logic.

The basic principle of testing RM logic circuits is reviewed. Using more economical NAND gates to replace AND gates in RM circuits is discussed, and this result doesn't change testing a RM circuit.

From this section, it can be seen that an easily tested circuit in RM logic is obtained at the expense of hardware cost. In practical applications, a trade-off between hardware cost and test should be decided. Because testing a circuit in RM logic is based on the gate fault model, this is difficult to apply to a large circuit[LaLa 85]. In addition, most arithmetic circuits are regular and iterative circuits[Hwang 79], a cell fault model[Parthasarathy 81, Wu 90] for testing these circuits is considered to be preferable[Cheng 87, Chatterjee 87, Bhatia 91]. For these reasons, it is assumed that the cell fault model is used for test in this project. In this way, no knowledge of actual realization of the cell is needed[Parthasarathy 81]. Therefore, a loose definition of RM logic is used in later chapters, that is, a minimized RM function can be realized in the mixed representations, since only the minimum circuits measured by the number of transistors are required.

Chapter 4

Number Systems and Two Operand Adders

4.1 Introduction

High performance adders are used not only for addition, but also for subtraction, multiplication, and division. For most more complex arithmetic processors, such as matrix multiplication, convolvers, FFTs, etc., the adder is one of the most critical building blocks. The speed of an arithmetic processor depends heavily on the speed of the adders used in the system.

An adder circuit, compared with other arithmetic circuits, is related more closely to a lower level design, e.g., a logic level design or circuit level design in which, logic representation and its implementation are emphasized. In contrast, the design of a complex arithmetic processor is mainly emphasized at a higher level, e.g., a structured level or functional level where, the basic building blocks and their relationship are emphasized. Because this project is mainly related to logic minimization and its implementation in arithmetic design, various adders are comprehensively studied.

In addition, the fixed-point structure will be discussed later. A floating-point structure is more complex than the fixed-point structure, but its implementation is based on the implementation for the fixed-point structure. A floating-point structure is usually studied at a higher level of abstraction.

An important factor that will largely influence the algorithm and implementation in arithmetic design is the number representation used, and different number representations may result in different hardware designs.

In this chapter, firstly, some of the most widely used number systems are described, this gives a necessary background for the later content. Secondly, the previous work about the most commonly used adders and RM logic applied to arithmetic circuit design are reviewed. Thirdly, the design methodology for arithmetic circuits employed in this work is discussed. Then, the most widely used carry lookahead scheme, as a typical example, is investigated in both RM logic and Boolean logic, and a comparative study is carried out. Later, a CMOS carry chain adder architecture is proposed, and developed for realizing various residue additions. Finally, a non-traditional on-line adder, which computes from the most significant bit first, is presented.

4.2 Number Systems for Arithmetic Circuits

There exist many number systems[Scott 85], some of them may be only of academic interest, because they are too complex to be realized in circuit. In the following, three kinds of the most commonly used number systems are described.

4.2.1 2's Complement Number System

Any n bit 2's complement number A may be expressed as

$$A = a_{n-1}a_{n-2}a_{n-3}.....a_1a_0 \tag{4.2.1}$$

where a_i= 0 or 1, $i \in \{0,1,.....,n-1\}$. The value of A can be calculated as

$$\begin{aligned} A &= -a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + a_{n-3} \times 2^{n-3} + ...a_1 \times 2^1 + a_0 \times 2^0 \\ &= -a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i \end{aligned} \tag{4.2.2}$$

In the 2's complement number system, the most significant bit (*msb*) of a number determines its sign, i.e., a number is positive when its *msb* is "0" and it is negative when its *msb* is "1". The range is

$$-2^{n-1} \le A \le 2^{n-1} - 1 \qquad (4.2.3)$$

This is a conventional and also the most widely used number representation. There are two main advantages for 2's complement number system: (1). addition and subtraction can be performed in an adder without increasing any additional hardware cost; (2). the circuit to perform addition, the most fundamental arithmetic operation in computer system, is usually simpler, in comparison with most other number systems. For these reasons, hardware implementations for other number representations are often based on that for the 2's complement number system, and also, the implementation result for the 2's complement number system is often used as a criterion to measure the performance efficiency for other number systems.

It should be mentioned that although a sign-magnitude number is different from a 2's complement number, their hardware implementation techniques are similar. In general, the implementation of the former is relatively simple. In other words, the study of the hardware implementation technique for the 2's complement number system is more general, it can usually cover that for the sign-magnitude number system.

4.2.2 Signed-Digit Number System

Signed-digit (SD) number representation was first introduced by Avizienis[Avizienis 61] to attempt to achieve high speed arithmetic operations, because it allows addition and subtraction operations with a carry (or borrow) restricted to two adjacent digital positions of the operands. In a SD number system, number representations allow redundancy to exist, which makes it possible for addition and subtraction operations without a long carry chain. For this reason, a SD number system is also called a redundant number system.

In the conventional fixed-radix systems, the digit set is restricted exactly to r values $\{0,1,\cdots\cdots,r-1\}$. The sign part and magnitude part are often separated, or the sign information is contained in the *msb*. Therefore, the numbers in these systems are usually considered non-sign numbers. In the SD number system, each digit contains a sign and the most significant non-zero digit of a number determines its sign. For a given radix r, each digit of a number is allowed to be more than r values, thus, the following $2\alpha+1$ values can be taken

$$\{\bar{\alpha},\overline{\alpha-1},.....,\bar{1},0,1,.....,\alpha-1,\alpha\} \tag{4.2.4}$$

where $\bar{x}$ equals $-x$. The maximum value of α can be equal to the radix r, in this case, it will yield a high level of redundancy which may make the implementation too costly, since a large digit set requires a large number of bits to represent each digit. In order to avoid high redundancy, the maximum magnitude of α must be within the following region[Hwang 79, Koren 93]

$$\left\lceil \frac{r-1}{2} \right\rceil \leq \alpha \leq r-1 \tag{4.2.5}$$

where $\lceil x \rceil$ stands for the smallest integer that is larger than or equal to the real number x. For many applications, in order to yield minimum redundancy in a balanced digit set, one can choose the following value for the maximum magnitude[Hwang 79]

$$\alpha = \left\lfloor \frac{r}{2} \right\rfloor \tag{4.2.6}$$

where $\lfloor x \rfloor$ stands for the largest integer that is less than or equal to x. Therefore, $\alpha = \frac{r}{2}$ when r is even and $\alpha = \frac{r-1}{2}$ when r is odd.

In general, the hardware implementation for SD number operations is more complex than that for 2's complement, since each signed digit needs more than one bit to represent it. In many situations, a SD number may be required to convert to or from a natural non-signed number. When a non-signed number is converted to a SD number, it may not need any hardware, because a non-signed number can be generally considered a special case of SD number representations, namely, all digits have the same sign, positive or negative. When a non-signed number is converted from a SD number, it can employ the following equation

$$A = \left| \sum A_{SD}^{+} \right| - \left| \sum A_{SD}^{-} \right| \tag{4.2.7}$$

where A_{SD}^{+} (A_{SD}^{-}) is sum of all positive (negative) digits. The range of SD numbers is

$$-r^n + 1 \le A_{SD} \le r^n - 1 \tag{4.2.8}$$

4.2.3 Residue Number System

Unlike the previous two kinds of number systems, residue number system (RNS) is a non-weighted number system. RNS does not consist of a single radix only, but of a *k*-tuple of integers, m_1, m_2, m_3, ... m_k, where each individual member is termed a *modulus*. *Moduli* should be chosen so that they are pairwise relatively prime. An integer X is represented in RNS by a *k*-tuple $(x_1, x_2, x_3, \cdots\cdots, x_k)$ where x_i is a non-negative integer defined as

$$X = m_i \times q_i + x_i, \qquad i = 1, 2, \cdots\cdots, k \tag{4.2.9}$$

Where q_i is the largest integer so chosen that $0 \le x_i < m_i$. x_i is called the residue of X *modulo* m_i, $|X|_{m_i}$ and X *mod* m_i are commonly used.

In RNS, addition, subtraction and multiplication are carry-free, therefore, parallel processing can be performed for these three operations. Identities for these three operations are

Addition:

$$|X+Y|_{m_i} = \left| |X|_{m_i} + |Y|_{m_i} \right|_{m_i} = |x_i + y_i|_{m_i} \tag{4.2.10}$$

Subtraction:

$$|X-Y|_{m_i} = \left||X|_{m_i} - |Y|_{m_i}\right|_{m_i} = |x_i - y_i|_{m_i} \tag{4.2.11}$$

Multiplication:

$$|X \times Y|_{m_i} = \left||X|_{m_i} \times |Y|_{m_i}\right|_{m_i} = |x_i \times y_i|_{m_i} \tag{4.2.12}$$

The largest possible range of RNS can be determined by the following equation

$$M = \prod_{i=1}^{k} m_i \tag{4.2.13}$$

For a given M, if only non-negative integers are needed, the range can be set to $[0, M-1]$. If, on the other hand, negative numbers are also desired, then the range can be set to $[-\frac{M-1}{2}, \frac{M-1}{2}]$ if M is odd, or $[-\frac{M}{2}, \frac{M}{2}-1]$ if M is even[Koren 93].
Unlike addition, subtraction and multiplication, division is very complex in RNS[Szabo 67, Koren 93]. Because most logic systems use binary numbers, a number in RNS should usually be converted to or from a binary number, which is described as

(*a*). Binary to RNS:

$$x_1 = |X|_{m_1}, x_2 = |X|_{m_2}, \ldots\ldots x_k = |X|_{m_k} \tag{4.2.14}$$

(*b*). RNS to Binary:

Chinese Remainder Theorem $$|X|_M = \left| \sum_{i=1}^{k} \hat{m}_i \left| \frac{x_i}{\hat{m}_i} \right|_{m_i} \right|_M \tag{4.2.15}$$

where $\hat{m}_i = \frac{M}{m_i}$.

Conversion from a residue number to a binary number is often considered to be a bottleneck problem, which restricts applications of RNS in many situations. In addition, some operations, such as division, comparison, sign detection, and scaling, etc.. are very complex and slow[Szabo 67, Koren 93], therefore, RNS is now rarely used for general-purpose computer systems, but it may be suitable for some digital signal processors, because additions, subtractions and multiplications can operate in a parallel fashion.

In fact, addition and subtraction in the 2's complement number system can be considered a special case of *modulo* operations with *modulus* equal to *n*, the length of a number representation.

4.3 Review of Two Operand Adders

Adders, as one of the most important parts in any computer architecture, have been extensively studied. A ripple-carry adder is very simple and includes a group of full adders (FAs) connected in series. It is also the most regular adder and has no global connection, which make it easily extendible to any length and realisable in VLSI. Unfortunately, a ripple-carry adder may be too slow for many applications, since its speed is linearly proportional to the length of operands, therefore, some faster adders have been developed.

In order to speed up the addition operation, the long carry propagation should be dealt with. The essentially serial nature of carry propagation is the most difficult problem in speeding up addition. There are two main approaches that can deal with this problem: one is to reduce the carry propagation time; the other is to detect the completion of the carry propagation and avoid wasting time while waiting for the worst case delay, which is $n \times T_{FA}$, where n and T_{FA} stand for the length of a ripple-carry adder and the delay of a basic cell FA, respectively. The second approach leads to a variable operation time and requires an asynchronous logic design, which may be inconvenient and increases the complexity in a synchronous design[Hwang 79, Koren 93]. Therefore, in most situations, the first approach is employed. The adders employing the first approach are discussed in the following.

There exist three main schemes to speed-up addition. They are:

(*1*). carry lookahead scheme;

(*2*). pre-computed scheme (for the conditional sum adder and the carry-select adder);
(*3*). carry-skip scheme.

These three schemes underlie most of the previous algorithms developed for designing a two operand adder, and many of these algorithms, in fact, can be found to be a variant of one of these schemes.

The carry lookahead scheme is the most commonly used means for accelerating carry propagation. The main principle of the carry lookahead scheme is an attempt to generate all incoming carries in all stages simultaneously and avoid a ripple-carry operation. This is achieved by adding some extra components which form the carry lookahead circuit. However, in practice the number of stages over which the lookahead can be applied is limited by the complexity of the gating structure, since more significant stages require successively more logic. It is common to construct four stage carry lookahead units. For a long adder, a two level or multi-level carry lookahead scheme is used. Direct realization of a carry lookahead adder, which is derived from the conventional individual gate implementation, is not very efficient in VLSI, due to its irregular layout. Brent and Kung developed a regular layout for implementing a carry lookahead adder[Brent 82]. Another interesting variant of the carry lookahead adder was first suggested by Ling[Ling 81]. In his work, Ling showed that an arbitrary function could be propagated, unlike the conventional adder, where a function propagated has a definite physical significance[Doran 88]. Ling stated that this scheme not only reduces the component count in design, but also requires fewer logic levels in adder implementation. In fact, at present it is unclear whether the advantage of Ling's adder is widely accepted or not, because Ling's scheme reduces the complexity of the carry lookahead unit, but increases the complexity of summation. Consequently, the latter may offset the gain of the former. The best static CMOS implementation for the carry lookahead unit known was proposed by Lee et al[Lee 93]. Their circuit is designed at the transistor level rather than at the gate level, and a four bit high-speed carry lookahead circuit based on transistor sharing in multi-output static CMOS complex gate is presented.

Another scheme for speeding up the addition operation is the conditional sum adder which was presented by Sklansky[Sklansky 60]. The main idea behind this scheme is that, for a given group of operands, say k bits, two sets of outputs are generated. Each set includes a k bit sum and an outgoing carry. The incoming carry of one set is assumed to be zero, while that of the other is assumed to be one. The eventual incoming carry is used to select the correct output value from the two sets which are pre-computed. A variant of the conditional sum adder is the carry-select adder. The

difference between them is how to divide operands into groups. Although the conditional sum adders and the carry-select adders are utilised less often than the carry lookahead adders, the technique of pre-computation is employed in some designs[Srinivas 92, Guan 93].

Another well known adder called the carry-skip adder was invented by Babbage in the 1800's[Kantabutra 93]. Later, a fast carry circuit based on the carry-skip technique was developed by Kilburn et al[Kilburn 60] at Manchester University in 1960. For this reason, the carry circuit is termed the Manchester carry chain. The original Manchester adder was designed in TTL circuitry, and its basic principle can be applied to MOS circuits[Chan 90]. The carry-skip adder is a more generalized adder. A critical factor of designing a carry-skip adder is related to how to decide the sizes of blocks for a given word. Guyot et al[Guyot 87] presented a method of finding near-optimum block sizes for designing a carry-skip adder, they reduced their optimization problem to a geometrical problem. Chan et al [Chan 90, 92]presented algorithms for computing the best block sizes for a carry-skip adder, and also for a carry lookahead adder. Kantabutra thought the methods of Guyot et al and Chan et al are computationally intensive, and do not yield a simple, intuitive understanding of optimum block sizes[Kantabutra 93]. In his work, Kantabutra used a circuit design procedure based on simulating the components to be used in a real adder. Thus , an adder obtained by using this procedure is very likely to be closer to optimum than the previous methods proposed by Guyot et al and Chan et al.

Adders described previously are usually designed in 2's complement representation. Some 2's complement adders can be obtained by employing other number representations. One example was presented by Srinivas and Parhi[Srinivas 92]. They proposed a so-called sign-select adder by making use of binary SD representation. This sign-select adder first uses redundant number addition in parallel, and then, a fast converter is used to transform the binary signed digit number into 2's complement form. The sign-select adder was concluded by their authors to be better than the carry lookahead adder and the carry select adder. Later, it was shown in our work[Guan 93], that the redundant number addition in the sign-select adder is not necessary, and the converter of the sign-select adder can be modified to a carry chain adder. This is because all known algorithms for converting binary signed digit numbers to 2's complement numbers, in fact, are derived from equation (4.2.7). That is, a carry (borrower) problem can not be avoided in the converter. In most situations, the speed-up methods stated above, i.e. the carry lookahead scheme, the pre-computed scheme, and the carry-skip scheme, can be used for the converter.

There is a common point for designing all the adders mentioned above, that is, any arithmetic design project can be divided into two phases, first is to develop efficient

algorithms and the second is to develop their logic implementations[Hwang 79]. The design following these two phases in this project is called the structured design which is discussed in the next section.

Compared with the conventional Boolean logic, many researchers think RM logic can be more economical for arithmetic design, and if RM logic is used in designing arithmetic circuits it will yield better results[Helliwell 88, Perkowski 90, Saul 92, 93, Lester 93, Csanky 93]. In fact, except for the parity check which can be also explained by *mod* 2 sum, it is hard to find a good example in practical applications to demonstrate this.

A survey on RM logic in the literature shows that only Saul has used RM logic to design some arithmetic circuits[Saul 92]. Saul has done much significant work to improve algorithms for minimizing RM functions. He developed a procedure for multi-level RM minimization and used it to design an eight bit counter and a four bit adder using a gate library containing a wide range of gates, which includes AOI and OAI, and the resulting circuits were compared with that designed using *MisII* based on Boolean logic. The circuits designed using RM logic were over 20 percent smaller and between 25 and 50 percent faster.

Saul's work[Saul 92] is not sufficient to prove the superiority of RM logic over Boolean logic in arithmetic design, and it seems only to prove RM logic over Boolean logic in some situations. One problem which restricts Saul's result to a practical application is that, the arithmetic functions used by Saul may be not very practical. For example, a four bit adder has only eight input variables and a carry-in variable is neglected. Another problem is that Saul's designs belong to the unstructured design, therefore, the result is hardly generalized. In addition, the result is not compared with currently existing arithmetic circuits.

In fact, it is not clear whether RM logic can find better applications in arithmetic design or not. The conclusion that RM logic can find better applications in arithmetic may come from two main results: (1). *mod* 2 sum, which is often realized by XOR gates, is widely used in arithmetic circuits; (2). benchmarks consist of many arithmetic functions, it is found that many of these functions in the RM expansion are more compact than the Boolean expansion. In practice, many benchmark arithmetic functions, at present, may be employed to measure a synthesis program rather than to design practical circuits. The main reason for this is that, many benchmark arithmetic functions are established in an unstructured fashion (unstructured design and structured design will be discussed later), and these functions may be not feasible for currently practical arithmetic circuits that are inherently well structured.

One aim of this project is to attempt to improve and develop arithmetic circuits using RM logic, and it focuses on practical circuit modules that are based on the structured

design, rather than on the benchmark arithmetic functions that are based on unstructured design as will be explained in the next section.

4.4 Design Methodology

Arithmetic circuit design, in general, can be classified into two categories: unstructured design and structured design.

In unstructured design, an arithmetic circuit may be implemented as a pure combinational circuit. For instance, the truth table of an arithmetic circuit with n inputs and m outputs is first listed, then, the logic representation is simplified, and finally, the simplified logic representation is realized. In general, the implementation of these circuits is preferred in PLA or ROM, see Fig. 4.4.1, because the layout of these circuits is often irregular.

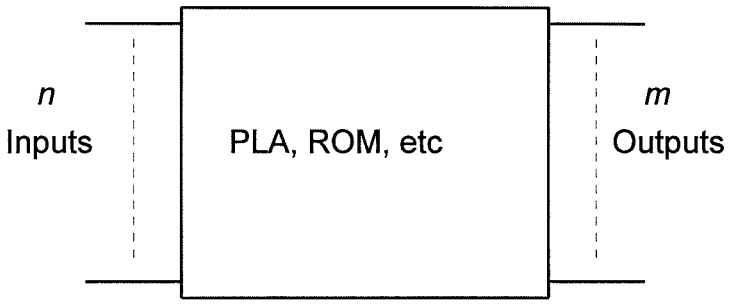


Fig. 4.4.1. Unstructured design.

Unstructured design may be feasible for small size operands, but for large size operands, say, a 64×64 multiplier, it may be impossible. Even though a 64×64 multiplier can be realized in the unstructured design, the resulting circuit may be very complex and its speed may be too slow. For any length of operands, e.g., longer than 500-bit[Takagi 92], this design method is no longer feasible.

The structured design methodology allows us to handle very complex design with two of the most commonly used engineering approaches, *hierarchy* and *abstraction*, and this methodology encourages the use of regular computing structures, and the design is hierarchical[Geiger 90]. This structured design methodology for VLSI was emphasized by Mead and Conway[Mead 80]. In structured design, a design procedure for arithmetic circuits may be generally divided into four steps shown by Fig. 4.4.2,

which is similar to ideal hierarchical design stages for CAD[Hurst 92]. In Fig. 4.4.2, if the second step, the structured design, is eliminated, then, this procedure becomes an unstructured design procedure.

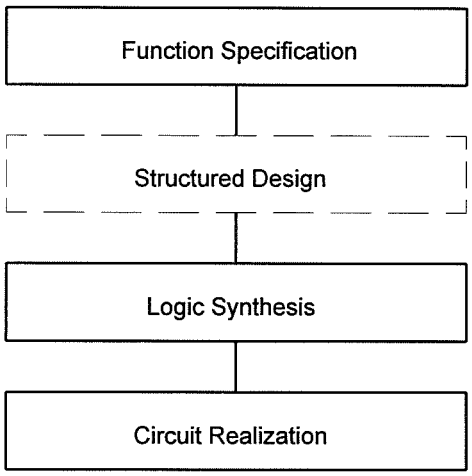


Fig. 4.4.2. A procedure of structured design.

The structured design is briefly described as

Step 1 (Functional Specification):

This seeks to relate a set of system inputs to the desired outputs. Simply speaking, what to design, an adder, a multiplier, even a complex arithmetic processor.

Step 2 (Structured Design):

This is carried out at an architectural or structural level design, which deals with defining blocks and the interconnection of blocks. Furthermore, a block can also be divided into some smaller blocks until getting some blocks with appropriate sizes, these are known as basic blocks, such as, full adder, multiplexers, registers, etc.. The structured design depends mainly on the number representation and its design algorithm, and it may be hardly affected by employing Boolean logic and RM logic. A design at this stage is often (but not always) considered to be independent of circuit technology. For many arithmetic circuits, the bit-slice principle is one of the most commonly used approaches in structured design.

Step 3 (Logic Synthesis):

Logic functions required in the design are synthesized and optimized in terms of certain criteria. There exist some different representations corresponding to a logic function, e.g., Boolean expansions and Reed-Muller expansions. The design based on different logic, Boolean or RM, may lead to a different result.

Step 4 (Circuit Realization):

Various circuit technologies can be chosen, such as TTL, NMOS, CMOS, BiCMOS, etc.; and also various circuit styles can be employed, such as PLA, CMOS, dynamic CMOS, etc.. According to the circuit style chosen in design, this may be further divided into some sub-steps. For example, it can be divided into transistor level synthesis, layout level synthesis, and layout realization. From Chapter 3, it can be seen that the design based on Boolean logic or RM logic may significantly affect transistor level synthesis and layout level synthesis.

Although a structured design needs one step more than an unstructured design, and probably takes more time to generate a circuit, it has some advantages:

(*1*). Circuit complexity, in many applications, grows linearly or near-linearly with the number of inputs. Thus, the number of logic elements needed, when there are many inputs, is likely to be less than that based on an unstructured design in which the rate of increase is nearly exponential.

(*2*) The design process is relatively simple and is essentially independent of the number of inputs. This means that a design result can be generalized. For example, the basic circuit structure of an n bit adder can be suitable for an $n+k$ bit adder, which, in an unstructured design, is not the case. In other words, the result based on a structured design has good *extendibility*.

(*3*) It is relatively easy for a smaller block to be absolutely minimized, and it is very difficult for a large logic function to be absolutely minimized. For a large function, absolute minimization has to be abandoned and heuristic (near minimal) solutions are employed[Sasao 93A], a good result is obtained, but it is unknown how close this result is to the optimum value.

(*4*) The resulting circuits are well formed and their interfaces are well defined, namely, they are modular. Consequently, some of the widely used modules can be elaborately designed and employed repeatedly.

(5) The structured design may generate a regular circuit configuration which is very suitable for VLSI implementation. The resulting circuit is easily pipelined, because in this situation, the circuit is easily decomposed into pipelined stages. In addition, testing a circuit can benefit from a regular circuit.

Because of the advantages described above, the structured design may take a shorter time to achieve a final circuit. In fact, most arithmetic circuits are based on the structured design, particularly for a circuit with a large number of inputs.

As stated above, an unstructured design is only feasible for a small circuit, but a structured design can be suitable for a larger and more complex circuit. Especially a completely structured design, like an iterative network (which is defined as a digital structure composed of a cascade of identical circuits or cells[Kohavi 78]), can be easily realized for an extremely large circuit, where the completely structured design means that all blocks are identical and all interconnection in the circuit is regular. One example of this is a carry-ripple adder, its size is easily extended, in addition, the circuit for a carry-ripple adder can be easily tested and the testing only uses eight patterns independent of the length of the adder[Cheng 87]. In practice, most designs may be not completely structured, the structured design attempts to make a circuit as regular as possible. From the view of VLSI, the interconnections in a circuit are required as locally as possible.

In theory, after a design is processed at a higher level, the result (data) is passed to the next level to be processed, step by step. The boundary for each processing level is clear and well defined. In this way, it favours automatic design. In practice, at present, manual design is often combined into an automatic design, which, sometimes may blur the boundary and definition of an hierarchical design.

It should be noted that structured design and structured circuits have different meanings. A structured design emphasizes the design procedure for a system, that is, a system possesses an inherent regularity for the designers to find. A structured circuit means that, an irregular circuit is implemented employing structured means, i.e. a structured circuit.

The PLA may be the most commonly used structured circuit, and it presents a simple layout process to simplify a random and irregular circuit implementation, rather than optimize the circuit. It is hard to divide all circuits into regular circuits and irregular circuits. Generally speaking, arithmetic circuits are mostly regular, and many control circuits are irregular.

4.5 Carry Lookahead Circuit

In this project, many existing circuit modules in practical arithmetic designs are re-considered in RM logic. In most situations, it is not found that RM logic can improve the previous results. In the following, the carry lookahead circuit is taken as an interesting example to show the effect of utilising different logic representations, because generating this circuit is significantly related to a logic representation.
The carry lookahead adder is one of the most commonly used arithmetic circuits, and the carry lookahead scheme is used not only for the carry lookahead adder, but also for other arithmetic circuits, such as Ling's adder[Ling 81] and the converter from redundant number to 2's complement number[Yen 92]. The structure of the carry lookahead adder can be illustrated by Fig. 4.5.1

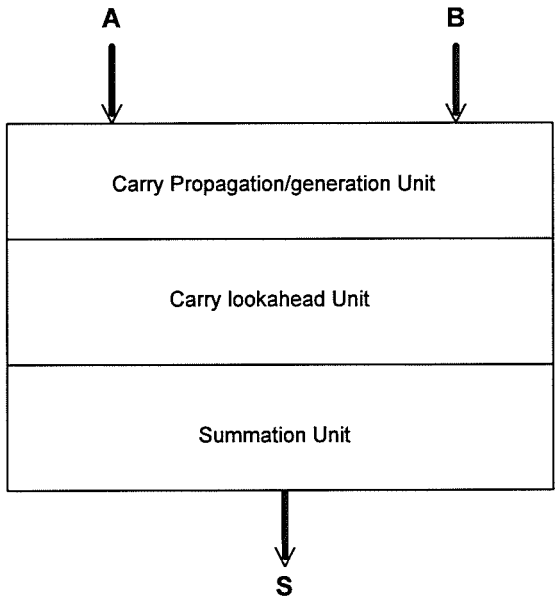


Fig. 4.5.1. The structure of carry lookahead adder.

The carry propagation/generation unit and summation unit are very simple and regular. The difficult part of the design is the carry lookahead unit. A group of equations for a four bit carry lookahead circuit can be derived from equation (3.7.2), they are recursive and expressed as

$$c_0 = g_0 + p_0 c_{in} \tag{4.5.1}$$

$$c_1 = g_1 + p_1 g_0 + p_1 p_0 c_{in} \tag{4.5.2}$$

$$c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{in} \tag{4.5.3}$$

$$c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_{in} \tag{4.5.4}$$

where $g_i = a_i \cdot b_i$ denotes carry generation and $p_i = a_i + b_i$ denotes carry propagation, respectively. It should be mentioned that the definition of carry propagation can be different in the literature. For example, $p_i = a_i + b_i$ is used by Koren[Koren 93], and $p_i = a_i \oplus b_i$ is used by Hwang and Almaini[Hwang 79, Almaini 94]. In equations (4.5.1)-(4.5.4), carry propagation $p_i = a_i + b_i$ and $p_i = a_i \oplus b_i$ can be exchanged with each other, and this will not affect the validity of the result. In some cases, this exchange is not valid, e.g. in a CMOS carry chain adder[Guan 93], $p_i = a_i \oplus b_i$ can not be replaced by $p_i = a_i + b_i$.

From *Example 3.7.1* in Chapter 3, it can be seen that the carry function has the same form in Boolean logic and RM logic without regarding operators + or ⊕. If equation (3.7.3) is initially used instead of equation (3.7.2) and if a carry variable substitution continues to be applied, then, another group of equations for representing a four bit carry lookahead circuit in RM expansion can be obtained.

Assume that the same representation of variables as those in equations (4.5.1~4.5.4) are used. According to equation (3.7.3), a group of equations for the carries of a four bit adder can be expressed as

$$c_0 = a_0 b_0 \oplus a_0 c_{in} \oplus b_0 c_{in} = a_0 b_0 \oplus (a_0 \oplus b_0) c_{in} = g_0 \oplus p_0 c_{in} \tag{4.5.5}$$

$$c_1 = a_1 b_1 \oplus a_1 c_0 \oplus b_1 c_0 = a_1 b_1 \oplus (a_1 \oplus b_1) c_0 = g_1 \oplus p_1 c_0 \tag{4.5.6}$$

$$c_2 = a_2 b_2 \oplus a_2 c_1 \oplus b_2 c_1 = a_2 b_2 \oplus (a_2 \oplus b_2) c_1 = g_2 \oplus p_2 c_1 \tag{4.5.7}$$

$$c_3 = a_3 b_3 \oplus a_3 c_2 \oplus b_3 c_2 = a_3 b_3 \oplus (a_3 \oplus b_3) c_2 = g_3 \oplus p_3 c_2 \tag{4.5.8}$$

Substituting for c_2, c_1 and c_0, c_3 can be written as

$$\begin{aligned} c_3 &= g_3 \oplus p_3 c_2 = g_3 \oplus p_3 (g_2 \oplus p_2 c_1) = g_3 \oplus p_3 g_2 \oplus p_3 p_2 c_1 \\ &= g_3 \oplus p_3 g_2 \oplus p_3 p_2 (g_1 \oplus p_1 c_0) = g_3 \oplus p_3 g_2 \oplus p_3 p_2 g_1 \oplus p_3 p_2 p_1 c_0 \\ &= g_3 \oplus p_3 g_2 \oplus p_3 p_2 g_1 \oplus p_3 p_2 p_1 (g_0 \oplus p_0 c_{in}) \\ &= g_3 \oplus p_3 g_2 \oplus p_3 p_2 g_1 \oplus p_3 p_2 p_1 g_0 \oplus p_3 p_2 p_1 p_0 c_{in} \end{aligned} \tag{4.5.9}$$

Consequently, a group of equations for a four bit carry lookahead circuit are

$$c_0 = g_0 \oplus p_0 c_{in} \tag{4.5.10}$$

$$c_1 = g_1 \oplus p_1 g_0 \oplus p_1 p_0 c_{in} \tag{4.5.11}$$

$$c_2 = g_2 \oplus p_2 g_1 \oplus p_2 p_1 g_0 \oplus p_2 p_1 p_0 c_{in} \tag{4.5.12}$$

$$c_3 = g_3 \oplus p_3 g_2 \oplus p_3 p_2 g_1 \oplus p_3 p_2 p_1 g_0 \oplus p_3 p_2 p_1 p_0 c_{in} \tag{4.5.13}$$

It can easily be seen that these two groups of equations (4.5.1~4.5.4 and 4.5.10~4.5.13) still have the same form without regarding operators + or ⊕. If the conventional implementation based on individual gates[Hwang 79, Chan 92] is employed, it is clear that the circuit to be realized in Boolean logic is better than that in RM logic, since a multi-input XOR gate is more complex and slower than a multi-input OR gate. In practice, the conventional implementation in Boolean logic is also not very efficient in CMOS circuit, because it was designed at gate level and requires 94 transistors[Chan 92], which is much more expensive than a static implementation based on a design at transistor level which requires 38 transistors[Lee 93] (note the authors stated the number transistors was 32, because single-rail logic is considered here, the extra 6 transistors for three INVs should be added for the three variables appearing in both true and complemented forms in their design).

If a dynamic CMOS circuit implementation is considered, then, the circuit in Boolean logic can be further simplified. According to *switching network theory* described earlier, the minimized networks for equation (4.5.1)~(4.5.4) are as follows

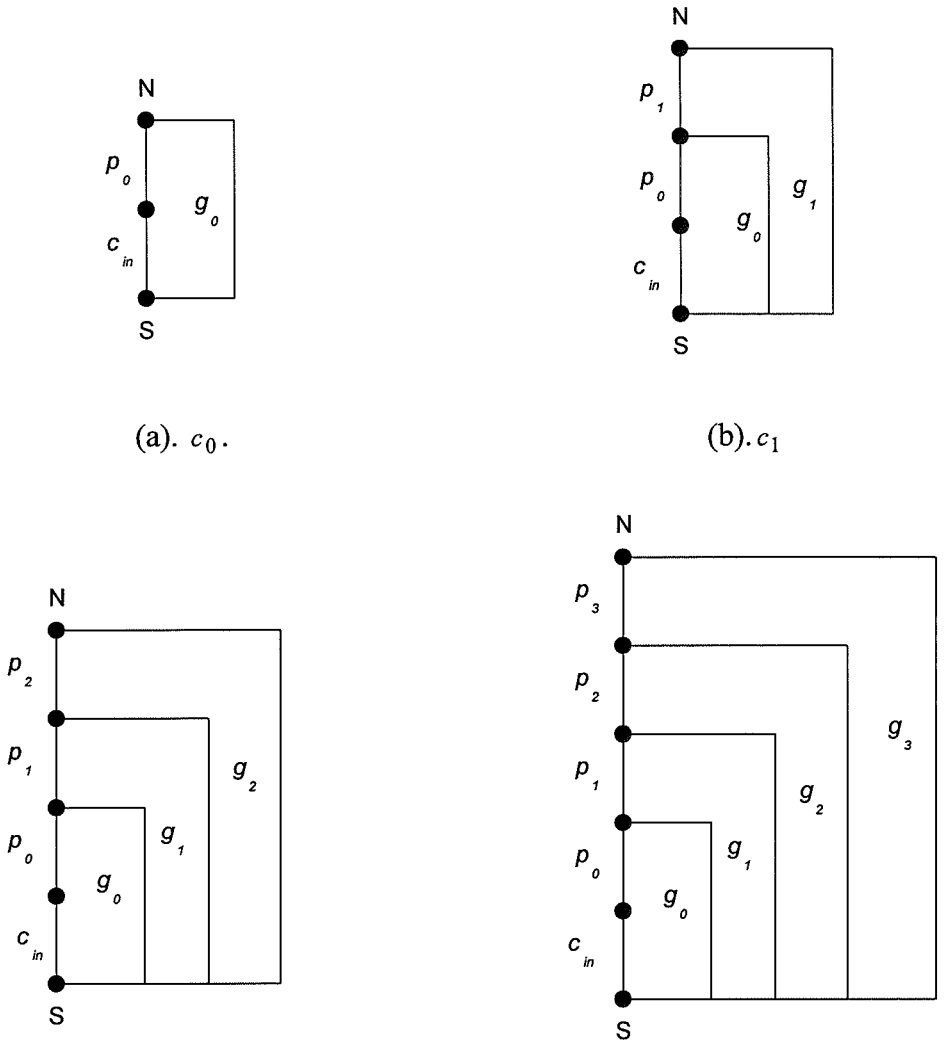


Fig. 4.5.1. The minimized networks.

From Fig. 4.5.1, it can be found that a four bit carry lookahead circuit can share transistors at circuit level design, and a dynamic CMOS circuit for implementing a four bit carry lookahead scheme is shown in Fig. 4.5.2. It requires only 22 transistors. When the design is carried out at gate level, the resulting circuits can only share gates. In contrast, sharing transistors can make a circuit simpler.

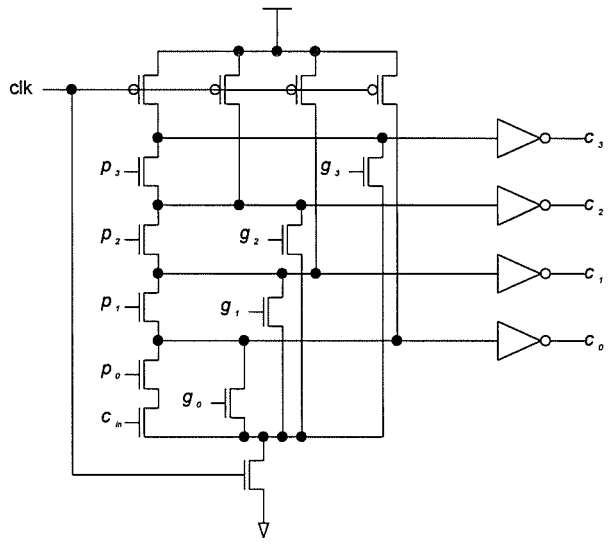


Fig. 4.5.2. A four bit carry lookahead circuit in dynamic CMOS.

In order to obtain a minimum implementation in the RM domain, equation (4.5.13) is expressed in multi-level form, thus

$$c_3 = g_3 \oplus p_3(g_2 \oplus p_2(g_1 \oplus p_1(g_0 \oplus p_0 c_{in}))) \tag{4.5.14}$$

According to equations (2.2.8) and (2.2.9), equation (4.5.14) can be rewritten as

$$c_3 = g_3 \oplus p_3\,(g_2 \oplus p_2\,(g_1 \oplus p_1\,(g_0 \oplus \overline{c_{in} p_0}\,))) \tag{4.5.15}$$

Realizing equation (4.5.15) leads to the minimum implementation for the carry lookahead scheme in the RM domain, this implementation shares gates as much as possible and is shown in Fig. 4.5.3,

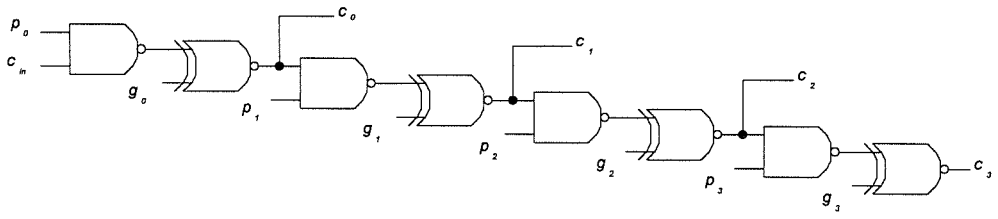


Fig. 4.5.3. The minimum implementation in RM logic.

This implementation will yield the minimum number of transistors, a NAND gate with four transistors and an XNOR gate with six transistors, a total of 40 transistors is required. Unfortunately, this circuit is too slow to satisfy time constraints in practical applications.

In this section, it can be seen that, from the view of the optimum implementation, Boolean logic is usually superior to RM logic for recursive functions, because the XOR operators in recursive RM functions cannot easily be exchanged with OR operators, this lead to their final implementation in individual gate fashion.

4.6 Carry Chain Adder

As stated earlier, structured design is used in this project. That is, a logic function is not established for the whole circuit but for the basic building blocks. In this section, a CMOS carry chain adder architecture is presented, which is from our previous work[Guan 93].

Similarly to the carry lookahead adder, the carry chain adder can be also divided into three parts which are shown in Fig. 4.6.1

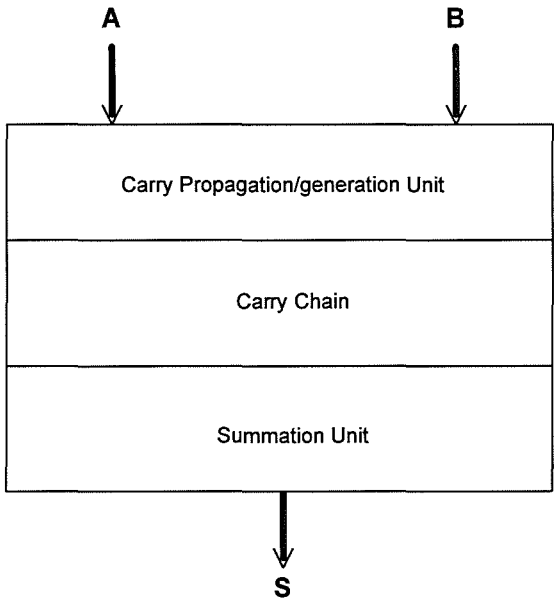


Fig. 4.6.1. The structure of carry chain adder.

In structure, the carry chain adder is similar to the carry lookahead adder. The main difference between them is that, a carry chain circuit is used to replace the carry lookahead circuit for speeding up a carry propagation.

MOS transistors can be employed to construct a highly efficient carry chain[Chan 90]. Chan and Schlag used dynamic CMOS circuits to realize a carry chain. In the following, a static CMOS carry chain is described. The experiment shows that the static CMOS carry chain[Guan 93] is as fast as the dynamic CMOS circuit[Chan 90], and also uses the same number of components as the dynamic version. The resulting static CMOS carry chain adder proposed has two main advantages over the dynamic version, that is: (1). it is easier for implementation because it doesn't need to consider the clock issue; (2). its carry propagation/generation circuit is simpler.

In the design for the static CMOS carry chain, when $a_i b_i = 00$, it is defined as "0" carry generation; and when $a_i b_i = 11$, it is defined as "1" carry generation. This is shown in Table 4.6.1

Table 4.6.1. Carry propagation/generation functions.

a_i	b_i	State	XOR	Carry-in	Carry-out
0	0	$g_i(0)$	0	x	0
0	1	p_i	1	x	x
1	0	p_i	1	x	x
1	1	$g_i(1)$	0	x	1

Where x is a binary variable applied to carry-in, i.e. equal to "0" or "1", $g_i(0)$ and $g_i(1)$ stand for "0" carry generation and "1" carry generation, respectively. From Table 4.6.1, it can be seen that an XOR gate can be used to distinguish these two states, carry propagation and carry generation. When XOR is "1", it indicates the carry propagation state. In this state, a carry-out signal only depends on a carry-in signal and is independent of input variables a_i and b_i. When XOR is "0", it indicates the carry generation state which can be further divided into two sub-states, "0" carry generation state and "1" carry generation state. It is also found that, in "0" carry generation state, both input variables a_i and b_i are "0", and in "1" carry generation state, both input variables a_i and b_i are "1". This means that, in the carry generation state, the carry-out signal can be determined by either a_i or b_i, and is independent of the carry-in signal. In this way, carry propagation and carry generation for each bit can be implemented by a single XOR gate instead of two gates, an XOR gate for carry propagation $p_i = a_i \oplus b_i$ and an AND gate for carry generation $g_i = a_i b_i$, which are used for the conventional implementation[Chan 90, Weste 93]. A complete circuit for a four bit adder is illustrated by Fig. 4.6.2

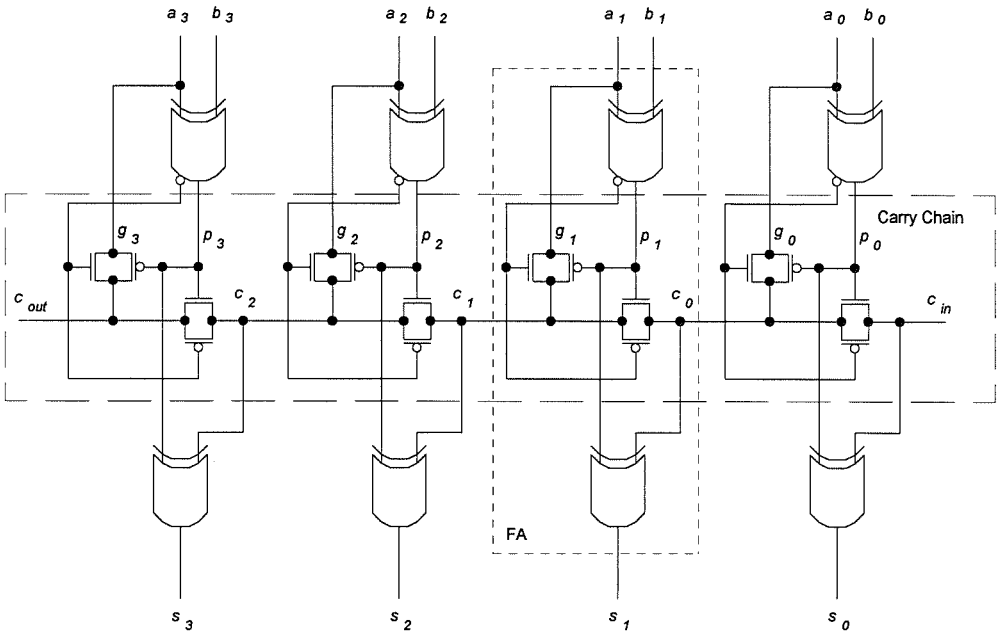


Fig. 4.6.2. A complete four bit CMOS carry chain adder.

It is worth noting that g_i is connected to a_i in Fig. 4.6.2. According to the above analysis, g_i can be also connected to b_i instead of a_i.

A long carry chain may heavily decrease the propagation speed of a carry, because the delay of the carry chain is proportional to n^2 [Pucknell 88], where n stands for the number of pass transistors in series. For solving this problem, a long carry chain should be divided into small sections and a buffer is inserted between every two sections.

In CMOS circuits, an INV can be used as a buffer with the minimum delay. In order to employ one INV as a buffer, a so-called "0" carry adder is introduced. "0" carry adder means that a carry signal is inverted, i.e., using "0" to replace "1" and "1" to replace "0" in the carry chain. It is achieved by adding an INV for the carry generation signal applied to each g_i and c_{in}. Therefore, the inverted carry signal is propagated along the carry chain. Finally, the sum s_i should be non-complemented, this is obtained from the following equation

$$s_i = a_i \oplus b_i \oplus \overline{c_{i-1}} = a_i \oplus b_i \oplus 1 \oplus c_{i-1} = \overline{a_i \oplus b_i} \oplus c_{i-1} = \overline{p_i} \oplus c_{i-1} \qquad (4.6.1)$$

Operation of equation (4.6.1) is based on equations (2.2.7) and (2.2.9). Fig. 4.6.3 shows a four bit "0" Carry adder

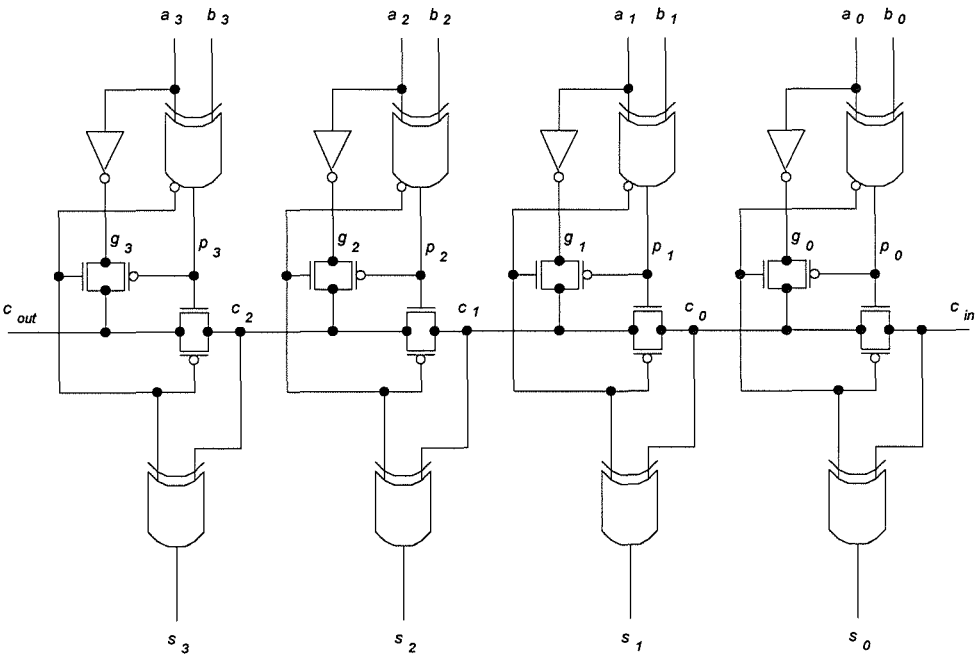


Fig. 4.6.3. A four bit "0" carry adder.

For simplicity, the adder in Fig. 4.6.2 is also called "1" carry adder. In this way, any length of adder can be constructed by alternately connecting a "1" carry adder and a "0" carry adder with an INV as a buffer, see Fig. 4.6.4

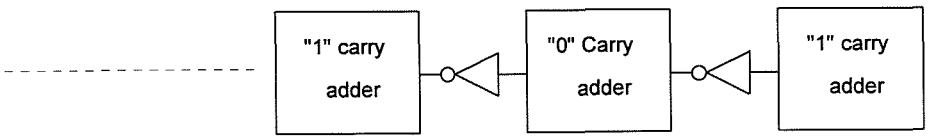
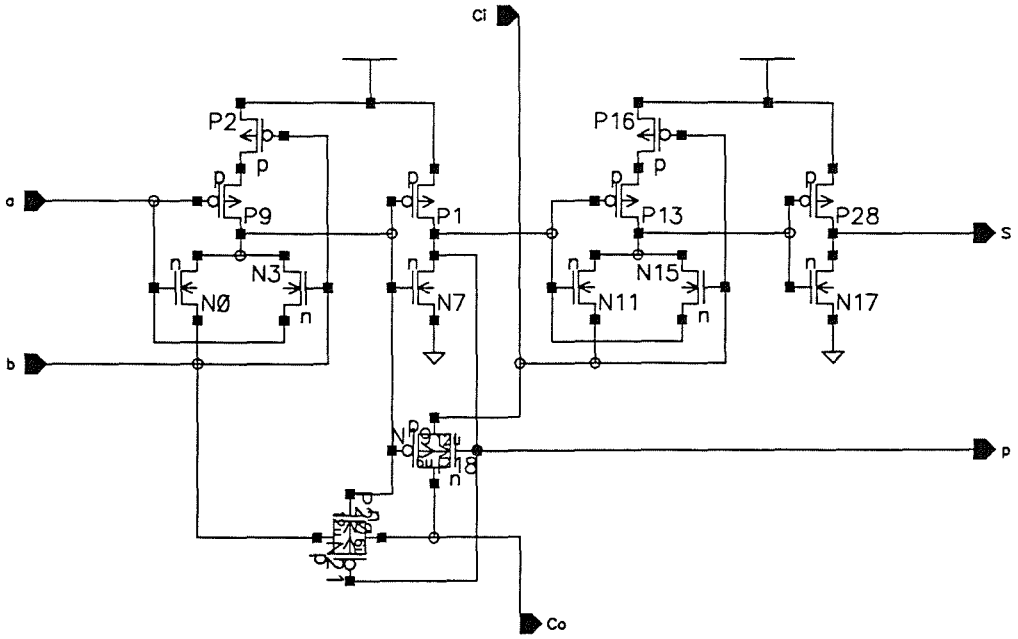


Fig. 4.6.4. A carry chain adder in series structure.

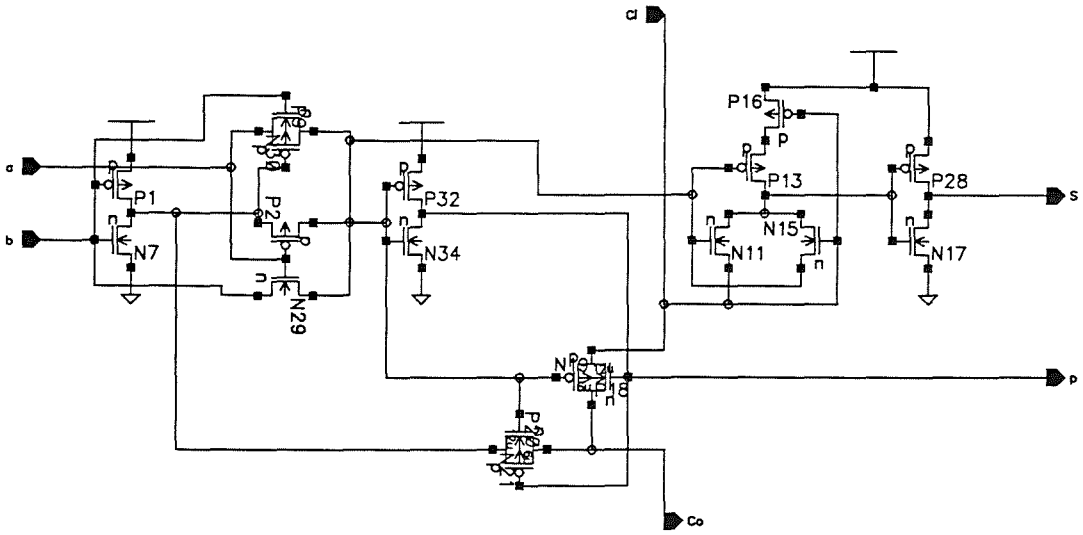
It is possible to use only "1" carry adders and employ buffers to separate the blocks and these blocks are divided in terms of optimum delay[Guan 93]. In this situation, the design is more regular since only one basic cell needs to be designed, but the experiments show that the speed is about 10% slower than that based on two kinds of FAs, "0" carry adder and "1" carry adder. In my experiment using Cadence suite with MIETEC 2.4 μ library, the worst time for a 32 bit adder with only "1" carry adders is 20.1 *ns* , and the worst time for a 32 bit adder with "1" carry adders and "0" carry adders is 17.9 *ns*, assuming that a block consists of a four bit adder.

The circuits of "0" carry adder and "1" carry adder, which are designed and used in the experiment, are shown in Fig. 4.6.5, these two circuits were designed using Cadence suite with MIETEC 2.4 μ library.

The "0" carry adder requires 18 transistors and the "1" carry adder requires 16 transistors. It is believed that both of these circuits are simpler than many existing versions of FAs. It should be mentioned that these two kinds of FAs are only minimised in circuit, measured by the number of transistors, but not optimized in the sizes of transistors, because this work is mainly concerned with the issue of logic minimization. Therefore, most of the circuits in the author's experiments, except buffers, use the standard size of transistors in a library. The circuits may be further optimized if the sizes of some transistors are changed.



(a). "0" carry adder.



(b). "1" carry adder.

Fig. 4.6.5 The circuits of "0" carry adder and "1" carry adder.

In order to obtain higher speed, two identical carry chains can be constructed in a so-called the carry chain select adder, which is derived from the pre-computed technique for the carry-select adder. The difference between them is that, the conventional carry-select adder duplicates two identical ripple-carry adders[Hwang 79, Koren 93] which leads to a requirement for more components, and the carry chain select adder only duplicates carry chains and summation circuits. Consequently, the carry chain select adder uses less components and is as fast as the carry-select adder. Fig. 4.6.6 illustrates a four bit carry chain select schematic

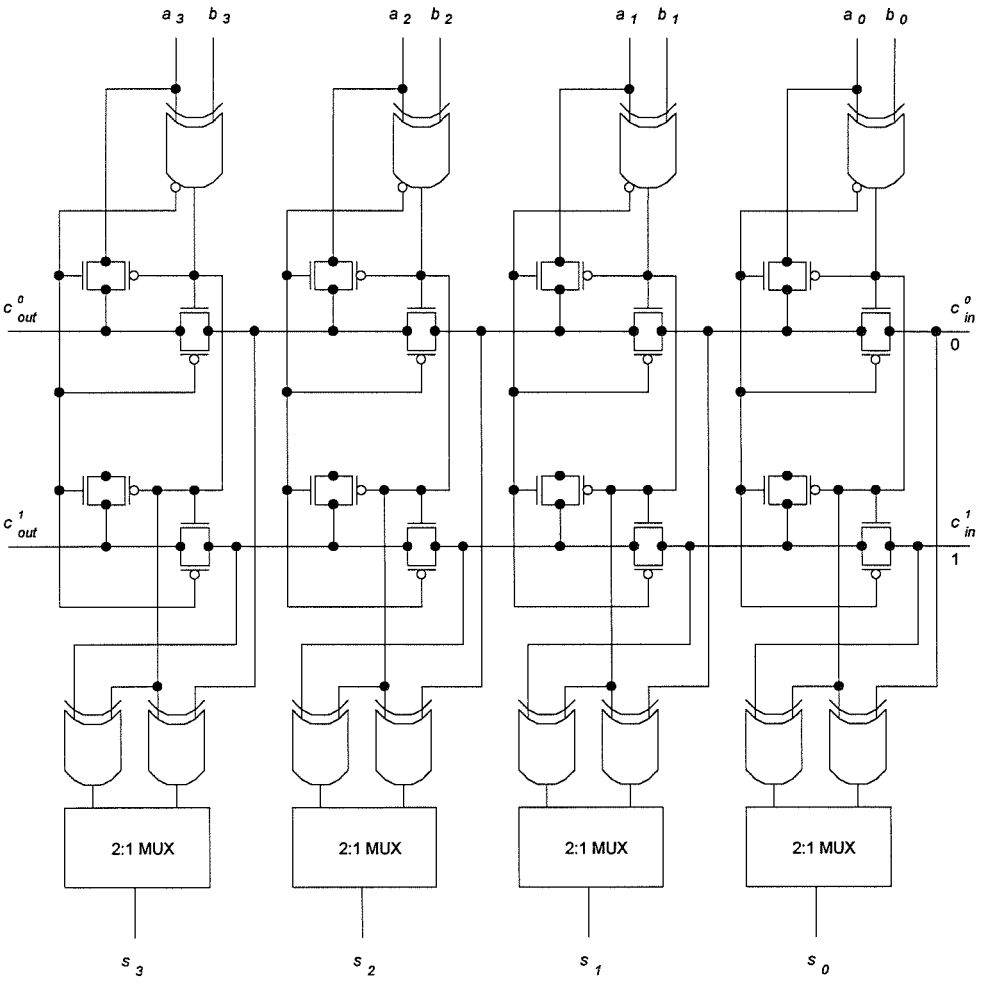


Fig. 4.6.6. Carry chain select schematic.

In Fig. 4.6.6, a superscript indicates that a carry-in signal is pre-set to "0" or "1". The CMOS carry chain adder was compared with the sign-select adder[Srinivas 92] and is found better than it[Guan 93]. It is worth mentioning that the sign-select adder was

considered by its authors to be better than the carry lookahead adder and carry-select adder.

In this section, a CMOS carry chain adder architecture is presented. From this design, it can be found that in logic synthesis, a logic representation and its minimization is not as important as that for the carry lookahead circuit, because the algorithm used is directly concerned with a transistor level design. The only thing concerned with logic representation is equation (4.6.1), in which, it is seen that it is easy for RM logic to adjust an electrical polarity in a design.

Although in the original design both Boolean logic and RM logic are hardly used, the basic building block (a FA shown in Fig. 4.6.2) is explained well by RM logic, which leads to another one bit adder design[Guan 95], i.e., a 5:3 counter which will be discussed in next chapter.

Based on the carry chain adder proposed, two 32 bit carry-skip adders were constructed using the 2.4μ library and simulated by Hspice simulator in the Cadence suite. The first adder, shown in Fig. 4.6.7, is divided into regular blocks (44444444), and the worst time is 11.2 *ns*. The second adder, shown in Fig. 4.6.8, is divided into near-optimal blocks (3458543) according to reference [Guyot 87], and the worst time is 8.5 *ns*.

The results are compared with a 30 bit carry-skip adder constructed and simulated in 2μ CMOS, which was presented by Kantabutra[Kantabutra 93]. Kantabutra' s adder is optimized in blocks (24566421), its worst time is 12 *ns*. Although Kantabutra's adder is optimized in blocks, these two carry-skip adders mentioned above, the first not optimized and the second nearly optimized in blocks, are still faster. This has been demonstrated by an experiment in which, a four bit adder was constructed using the circuit[Kantabutra 93] and simulated using the same conditions as that in [Guan 93]. The worst delay of this four bit adder is 3.8 *ns* which is slower than 2.5 *ns*, the worst delay of the four bit adder of our design[Guan 93].

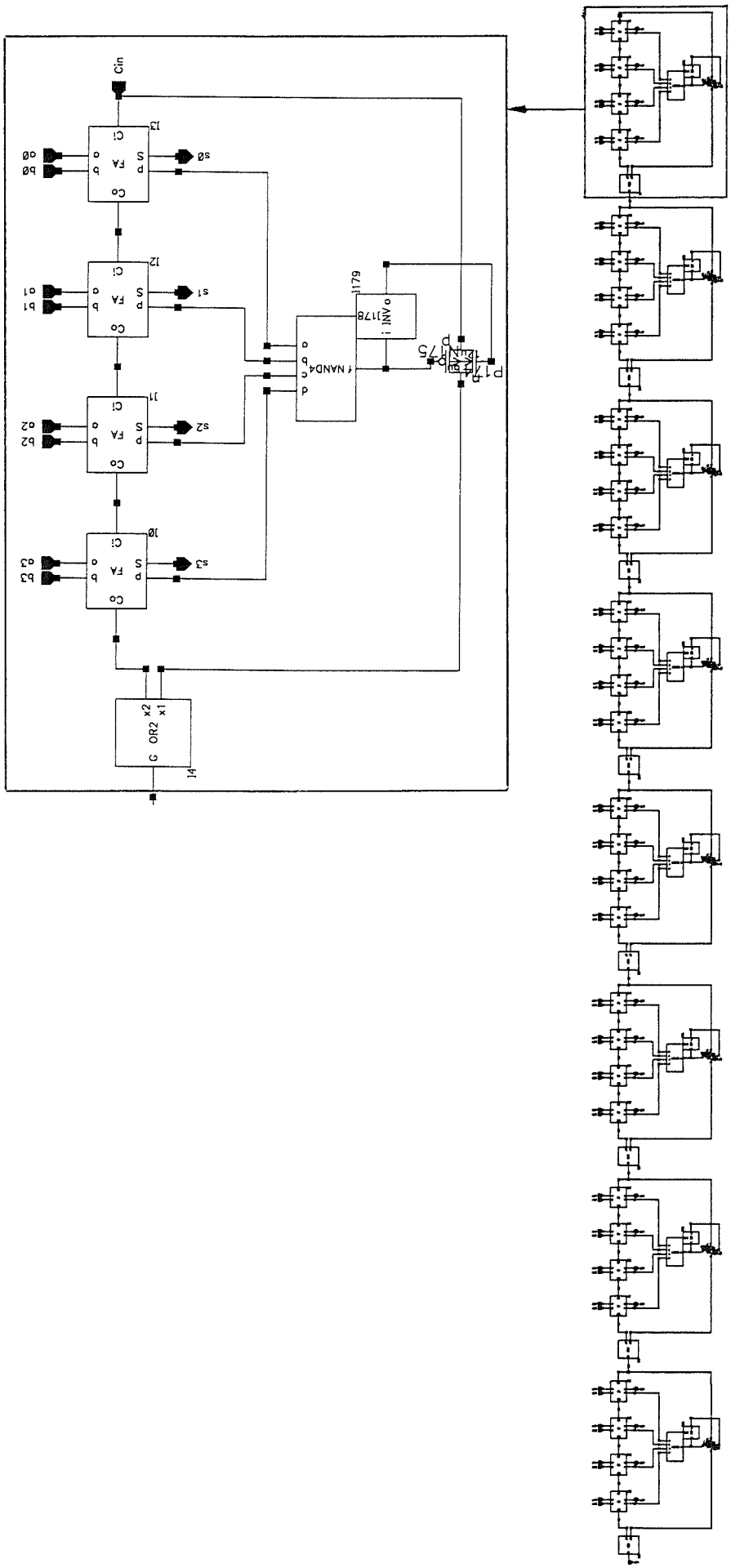


Fig. 4.6.7. A 32 bit carry-skip adder in regular blocks(44444444).

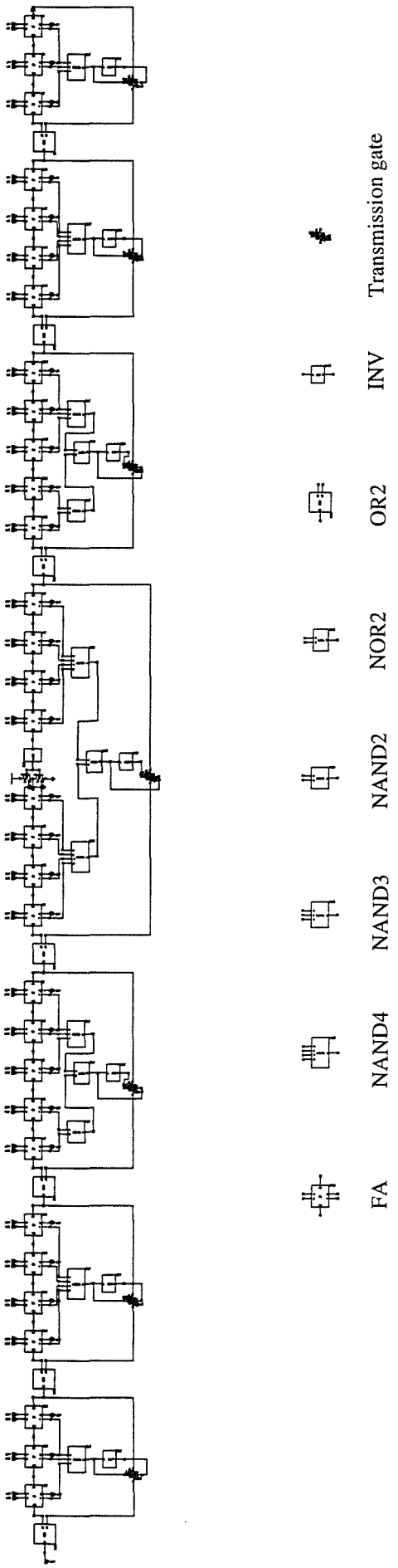


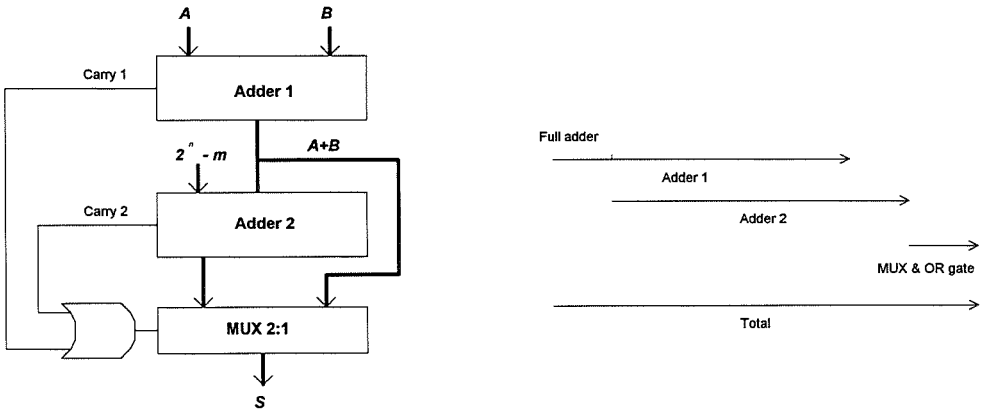
Fig. 4.6.8. A 32 bit carry-skip adder in near-optimal blocks(3458543).

4.7 Residue Adders

The basic principle of the CMOS carry chain adder proposed can be employed to improve the implementation of residue adders, this is discussed in the following.
There are several approaches to implement residue number addition, and the approach based on two binary adders is considered to be the fastest[Dulgate 92]. In the conventional design, two binary adders are used to implement a residue adder, one adder is used to implement the addition operation for *variable+variable* in the first cycle, and the other adder is used to perform the operation for *variable+constant* in the second cycle. This operation can be further described in the following.
In the RNS, if two operands A and $B < m$, then $(A+B)$ *mod m* can be defined as follows

$$|A+B|_m = \begin{cases} A+B & A+B<m \\ |A+B+2^n-m|_m & A+B\geq m \end{cases} \tag{4.7.1}$$

where $A+B\geq m$ is called the overflow condition. Two n bit binary adders may be used to implement this addition operation where $2^n \geq m$, see Fig. 4.7.1



(a). Two adders with MUX. (b). Implementation time (worst delay).

Fig. 4.7.1. Parallel scheme.

In the first cycle, **Adder 1** computes $A+B$, in the second cycle, **Adder 2** computes $A+B+2^n-m$, where 2^n-m is a correction factor, and finally, the correct sum result is selected by a MUX according to the following rule

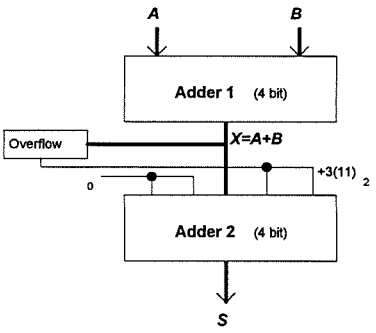
$$MUXoutput=\begin{cases} Sum2 & Carry1=1 \quad or \quad Carry2=1 \\ Sum1 & Otherwise \end{cases} \tag{4.7.2}$$

The example in Table 4.7.1 illustrates this operation. Here, m=5, n=3, the correction factor is $2^3-5=3$, two 3 bit binary adders are employed to implement modulo 5 addition.

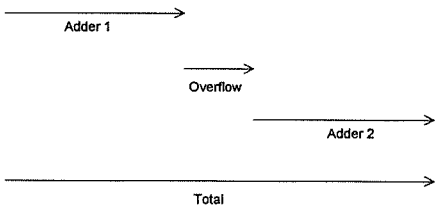
Table 4.7.1. Modulo 5 addition using two binary adders.

A+B	Sum 1	Sum 2	Carry 1	Carry 2	Output
0	0	3	0	0	Sum 1
1	1	4	0	0	Sum 1
2	2	5	0	0	Sum 1
3	3	6	0	0	Sum 1
4	4	7	0	0	Sum 1
5	5	0	0	1	Sum 2
6	6	1	0	1	Sum 2
7	7	2	0	1	Sum 2
8	0	3	1	0	Sum 2

Another possible scheme is that the output of the overflow detection circuit is used to control **Adder 2** to add the correction factor or not. An example of modulo 13 addition is shown in Fig. 4.7.2



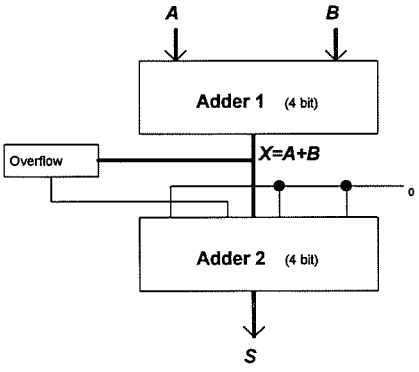
$$Oveflow = x_3 x_2 (x_1 + x_0)$$



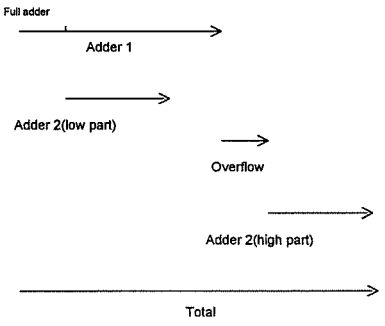
(a). Two adders without MUX. (b). Implementation time(worst delay).

Fig. 4.7.2. Serial scheme for modulo 13 addition.

Implementation of this scheme, generally speaking, belongs to serial operation, and the complexity and delay of the circuit depend heavily on the modulo to be selected. For instance, modulo 12 in this fashion is simpler and faster than modulo 13, because modulo 12 contains partial parallel operation, see Fig. 4.7.3.



$$Overflow = x_3 x_2$$



(a). Modulo 12 adder. (b). Implementation time.

Fig. 4.7.3. The serial scheme for modulo 12 addition.

It is worth mentioning that the decimal adder is a special case in RNS, i.e. the modulo 10 adder. In the following, the parallel scheme will be discussed. The approach is also suitable for the serial scheme.

From Fig. 4.7.1, it can be seen that **Adder 2** is only used to add one variable (the sum for $A+B$) and one constant $2^n - m$ (correction factor), that is , one input of each stage in **Adder 2** is always "0" or "1". In this case, the conventional binary adder can be simplified. Assume that x and i stand for the variable and constant applied to each stage of **Adder 2**, respectively, then the truth table for **Adder 2** is given as follows

Table 4.7.2. The truth table for simplified adder.

One input is always "0"							One input is always "1"						
i	x	c_{in}	s^0	c^0_{out}	g	p	i	x	c_{in}	s^1	c^1_{out}	g	p
0	0	0	0	0	0	0	1	0	0	1	0	0	1
0	0	1	1	0	0	0	1	0	1	0	1	0	1
0	1	0	1	0	0	1	1	1	0	0	1	1	0
0	1	1	0	1	0	1	1	1	1	1	1	1	0

Note that superscripts are used to differentiate between applying "0" and "1" to each stage in **Adder 2**, respectively. In Table 4.7.2, g stands for carry generation and p for carry propagation. Therefore, each stage in **Adder 2** can be expressed as follows

if i=0 for this stage, then

$$s^0 = x \oplus c_{in}, \qquad c^0_{out} = x \cdot c_{in} \tag{4.7.1}$$

if i=1 for this stage, then

$$s^1 = \overline{x \oplus c_{in}}, \qquad c^1_{out} = x + c_{in} \tag{4.7.2}$$

The simplified adders are shown in Fig. 4.7.4

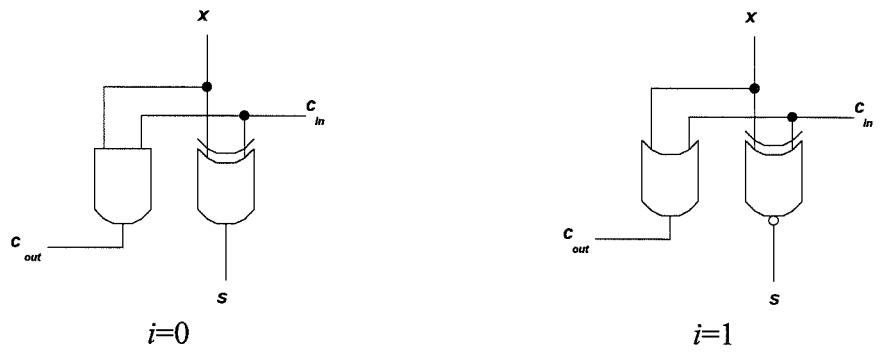


Fig. 4.7.4. Simplified adder cell for *variable + constant.*

It should be noted that no constant *i* appears in these expressions and circuits. That is, the constant *i* is implicitly included in the simplified adder, and only one input (variable *x*) is applied to each stage of the simplified adder. Therefore, this design decreases not only the number of components, but also the connection wires. For example, in order to implement modulo 9 addition, a four bit adder is required, and $2^4 - 9 = 7(0111)_2$ is the correction factor used to apply to **Adder 2** as the constant input, the variable input is *X=A+B* (the sum of the first cycle addition), this can be shown by Fig. 4.7.5

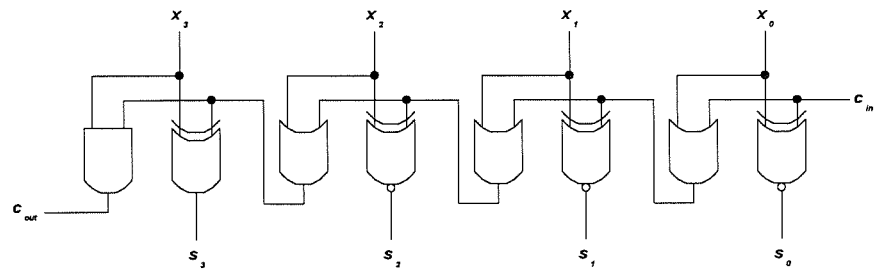


Fig. 4.7.5. The simplified adder for *x*+7.

Although the simplified adder can be used to replace a complete adder, its speed may be still slow. One approach to speed up operation is to employ a carry lookahead scheme. Unfortunately, many residue adders only need shorter words, a larger modulo is rarely used. The carry lookahead scheme is not suitable for shorter words[Hwang 79]. An experiment by the author demonstrates that if the length of operands is less or equal to eight bits, the CMOS carry chain adder is faster than all other high-speed

adders known, including the carry lookahead adder and the carry-select adder. In addition, the CMOS carry chain adder is also simpler than all the other adders, including the ripple-carry adder. In the following, the CMOS carry chain adder is applied to residue adders.

The rule to design a simplified adder for *variable+constant* based on the CMOS carry chain is defined in Table 4.7.3

Table 4.7.3. Rule to design *variable+constant* adder based on CMOS carry chain.

i=0	g=0	p=x
i=1	g=x	p=$\bar{x}$

An example of a complete modulo 9 adder using the CMOS carry chain is shown in Fig. 4.7.6

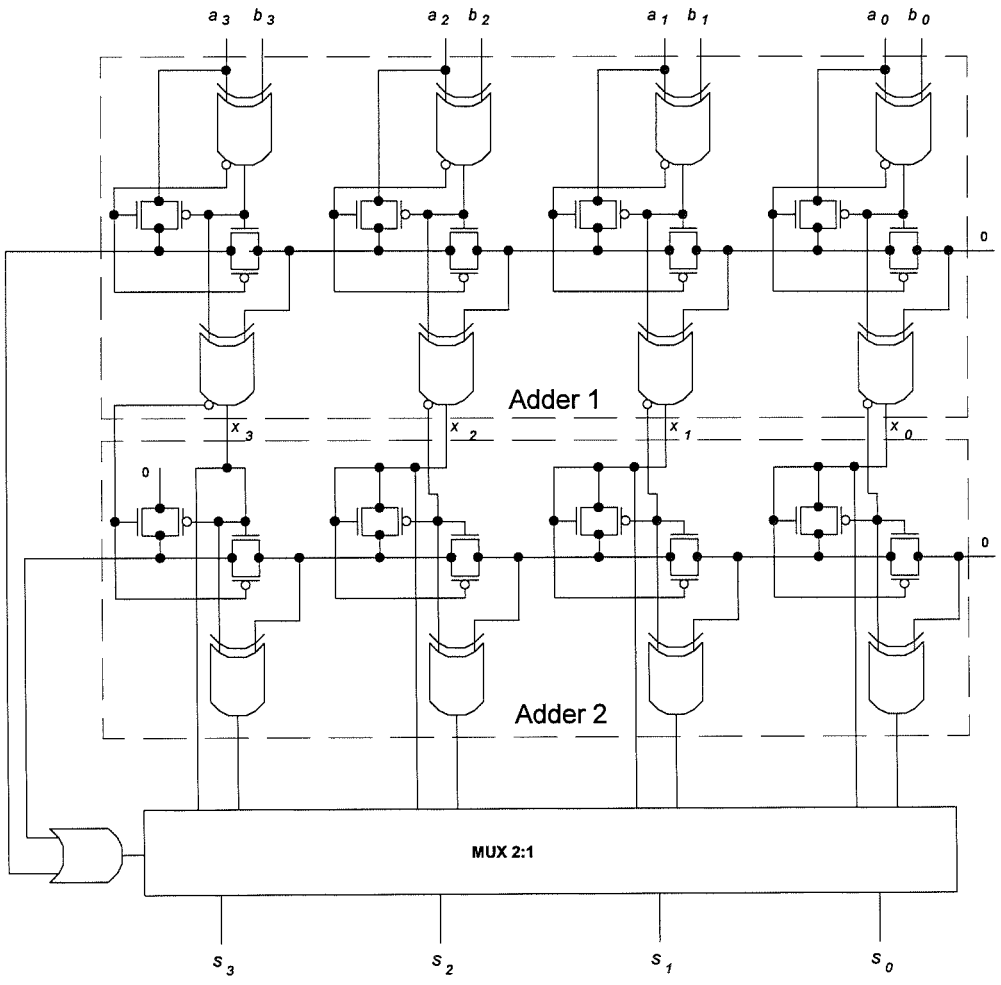


Fig. 4.7.6. Modulo 9 adder.

In this section, residue adders are explored according to the structured design approach. In fact, some residue adders are also studied according to the unstructured design, that is, a logic function is established for a residue adder, and then, the simplified function is used to realize a circuit. In this way, not only can the design not be generalized, but also the resulting circuits are slower and more complex than that based on the structured design. From this section, it can be seen that the structured design, sometimes, is more important than logic representation and its minimization. In the next section, another design to show this is presented.

4.8 On Line Adder

The traditional addition operation is always performed from *lsb* to *msb* due to the nature of the carry scheme. Therefore, the traditional addition operation may be not efficient in some situations. For example, if the output is only expected to have a required precision, the traditional method must compute the digits that will eventually be discarded. Another example is that data are transferred in series with *msb* first.
In this section, an addition algorithm from *msb* to *lsb* is proposed. This algorithm is influenced by Ercegovac and Lang's work[Ercegovac 87], where a conversion of redundant into conventional representations is developed.

Algorithm: Two n bit 2's complement number A and B can be expressed as follows

$$A = -a_{n-1}\,2^{n-1} + \sum_{i=0}^{n-2} a_i\,2^i, \qquad B = -b_{n-1}\,2^{n-1} + \sum_{i=0}^{n-2} b_i\,2^i \tag{4.8.1}$$

The sum of A and B is

$$S = A + B = -(a_{n-1} + b_{n-1})2^{n-1} + \sum_{i=0}^{n-2}(a_i + b_i)2^i \tag{4.8.2}$$

If only the *n-k* bit precision sum of A and B is wanted, equation (4.8.2) can be written as

$$S[k]=-(a_{n-1}+b_{n-1})2^{n-1}+\sum_{i=k}^{n-2}(a_i+b_i)2^i \tag{4.8.3}$$

The procedure of performing equation (4.8.3) can be described by a recurrent equation (4.8.4)

$$S[k]=S[k+1]+(a_k+b_k)2^k \tag{4.8.4}$$

According to equation (4.8.4), the value of a sum is computed starting from *msb*, i.e. from left to right, and the traditional addition operation is from right to left. However, this algorithm requires the propagation of a carry if both a_k and b_k are equal to 1. In order to avoid the propagation of a carry, two conditional forms can be employed, one assumes that there is a carry from the lower significant stage, the other assumes that there is no carry from the lower significant stage. Accordingly, the algorithm stated above can be modified as follows

$$S[k]=\begin{cases}A[k] & \text{if a carry from } k-1 \text{ stage}\\ B[k] & \text{if no carry from } k-1 \text{ stage}\end{cases} \tag{4.8.5}$$

Clearly, these two forms always keep a relationship which is

$$A[k]=B[k]+2^k \tag{4.8.6}$$

The operation of this algorithm is implemented from *msb* to *lsb* step by step. In every step, the truncated sum is kept in B assuming the lower order position to be zero. According to equation (4.8.5), the initial values should be

$$A[n]=1, \quad B[n]=0 \tag{4.8.7}$$

Any other values can be computed from the following equations

$$A[k]=\begin{cases} B[k+1]+2^k & a_k\ b_k = 00 \\ A[k+1] & 01,10 \\ A[k+1]+2^k & 11 \end{cases} \tag{4.8.8}$$

$$B[k]=\begin{cases} B[k+1] & 00 \\ B[k+1]+2^k & 01,10 \\ A[k+1] & 11 \end{cases} \tag{4.8.9}$$

The final result is

$$\begin{aligned} S[0]&=A[0] \quad \textit{if a carry from } -1 \textit{ stage} \\ S[0]&=B[0] \quad \textit{if no carry from } -1 \textit{ stage} \end{aligned} \tag{4.8.10}$$

An example is presented to illustrate the procedure

Example:

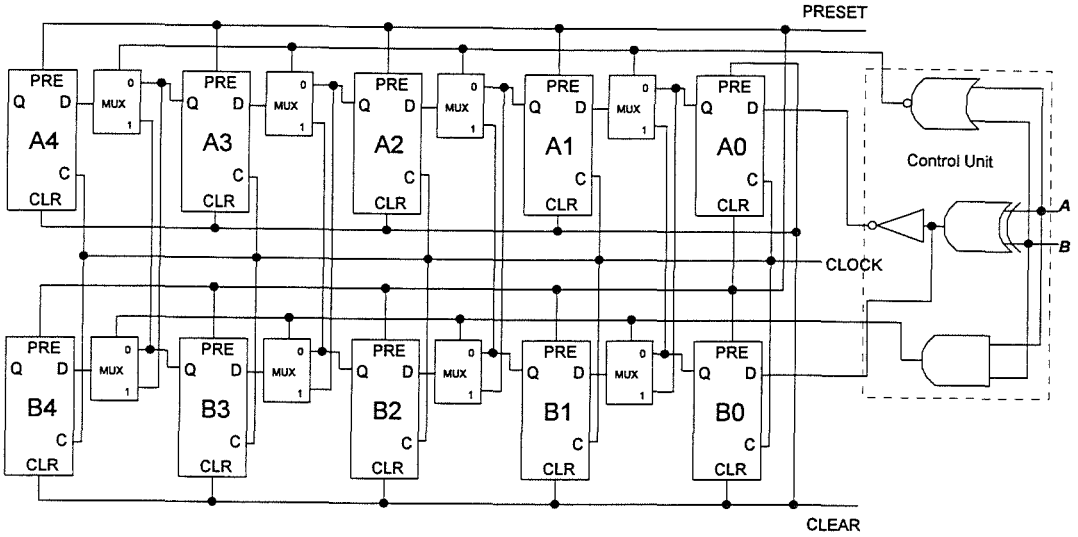
$$\begin{aligned} &876543210 \\ A&=10101101 \\ B&=01100111 \end{aligned}$$

k	a_k b_k	$A[k]$	$B[k]$
8	0 0	1	0
7	1 0	10	01
6	0 1	100	011
5	1 1	1001	1000
4	0 0	10001	10000
3	1 0	100010	100001
2	1 1	1000101	1000100
1	0 1	10001010	10001001
0	1 1	100010101	100010100

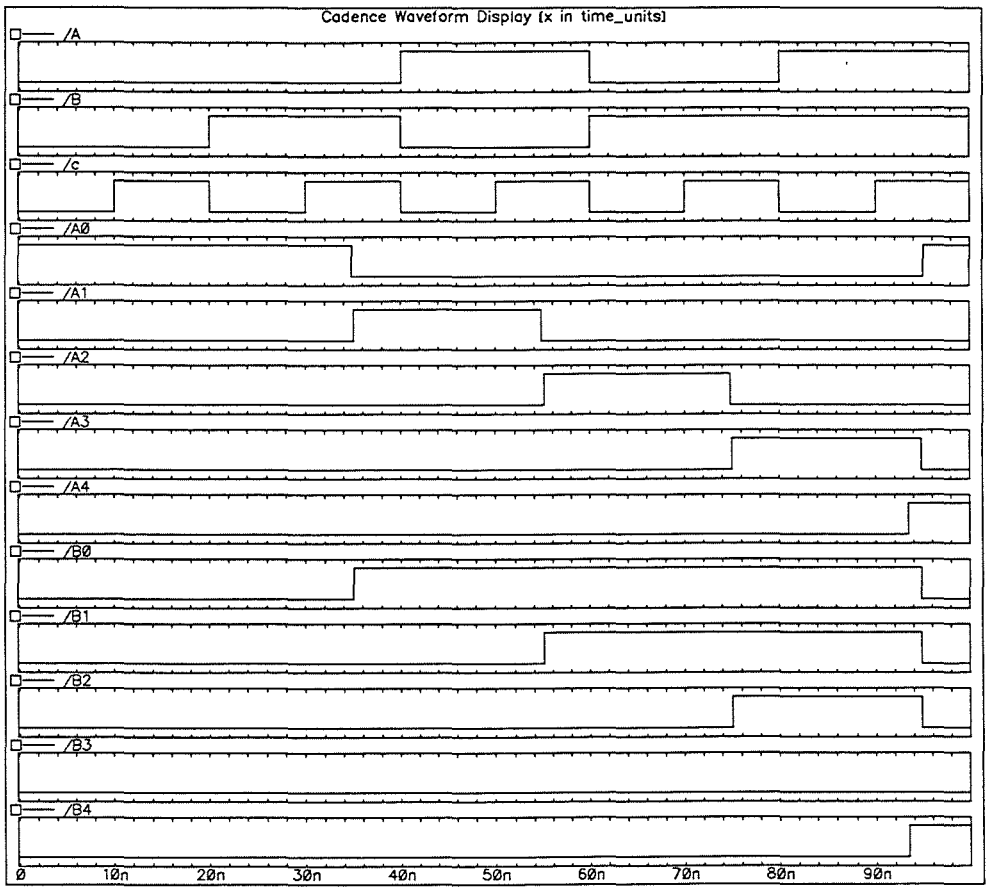
Implementation of the algorithm is simple, it requires two *n+1* bit registers to hold $A[k]$ and $B[k]$ since summing two *n* bit operands may generate a *n+1* bit result. These registers can be shifted one bit left with the insertion of a one or zero in the *lsb* dependent on the variables a_k and b_k. This also requires parallel loading between the registers. Fig. 4.8.1 shows a four bit adder and its simulation generated using the Verilog simulator in the Cadence suite.

In Fig. 4.8.1, the circuit is composed of three parts, shift register A including D flip-flops A0-A4, shift register B including D flip-flops B0-B4, and a control unit. the control circuit is simple and the design is based on equations (4.8.8) and (4.8.9).

Initially, flip-flop A0 is set to one while other flip-flops are set to zero using PRESENT and CLEAR inputs. The operands *A* and *B* are applied in series bit by bit, *msb* first. The final result is stored in B register. Its every step delay is roughly equivalent to the delay of a full adder. The circuit is simple and highly modular, also it has a simple and regular interconnection requirement for data and control. Therefore, it is very suitable for VLSI implementation.



(a). schematic.



(b). The waveform of simulation (test pattern:*A*=0101, *B*=1011).

Fig. 4.8.1. A four bit on-line adder.

4.9 Summary

In this section, the three most widely used number systems are briefly introduced, and a review of two operand adders in the literature is presented. Two design methods in arithmetic circuits, structured design and unstructured design, are discussed.

In the literature, it is not difficult to find that nearly all the highly efficient circuits for two operand adders designed previously are based on structured design, especially, employing the bit-slice principle[Hwang 79, Cavanagh 84, Scott 85, Koren 93]. In these existing circuits, the sizes of their basic building blocks are usually smaller, and also the types of these blocks are few. The most commonly used building blocks include FA, flip-flops, MUXs, etc., and the minimum implementations for these circuits have been extensively explored by many researchers and designers. In fact, further improving these circuits is not easily achieved by simply using RM logic or Boolean logic[Guan 95].

Although it is shown by some researchers that some benchmark addition functions are more compact in the RM domain, their results are not comparable to the existing circuits; the main reason for this is that their designs are based on unstructured design. The carry lookahead scheme is employed to show that again, the implementation based on individual gates that are often used in RM logic is more complex and slower. This result is believed to be applicable to most recursive functions. This is because in most recursive functions, each variable often appears in its true form or complemented form, but not both, in this case, it is hard to replace the XOR operation with the OR operation[Sasao 93A, Guan 94A]. But, it is also seen that it is easy for RM logic to adjust the electrical polarity, which is mentioned in the previous section.

At the same time, a CMOS carry chain adder is presented. This adder is flexible, according to different applications, a simple serial adder, a carry chain select adder, or a skip-adder based on the CMOS carry chain adder can be constructed. These adders are faster and simpler than many previous designs[Srinivas 92, Guan 93]. This adder is also used for residue addition.

An algorithm and its implementation for an on-line adder is proposed. This result can be applied to such situations in which the output of an addition operation is only expected to have a required precision, or a sequence of operations can be speeded up by overlapping the operations[Ercegovac 80]. In these situations, the traditional addition often can not be performed efficiently.

From the adders presented, it is seen that the main basic building blocks are still FA, flip-flops, MUXs, etc.. This is because the logic representations, the Boolean form and the RM form, hardly affect an architectural or structural design.

Chapter 5

Carry Free Adders and Parallel Multipliers

5.1 Introduction

Like an adder, a multiplier is also considered to be an essential part not only for general-purpose computers, but also for various digital signal processors. Adders and multipliers are the most important parts for various logic systems, because many numerical calculations can be ultimately reduced to two essential operations, addition and multiplication. Therefore, like an adder, the speed of a multiplier is also a critical design parameter for many logic systems. A multiplier is considered the limiting factor in both the performance and die size of most chips today[Goto 92]. Thus, multipliers are often designed and fabricated as benchmarks for demonstrating various high-speed technologies[Yano 90].

In design, the basic principle, *shift and add*, still underlies many algorithms and implementations for multipliers. Depending on applications, a multiplier may be implemented in different implementation styles, such as bit-serial, bit-parallel, digit-serial and on-line. In general, the design for a bit-parallel multiplier (simply called parallel multiplier), when compared to the others, is more closely related to the issue of logic minimization. In contrast, the design of the others are more concerned with timing. Since this project is mainly concerned with logic representations and their minimization, parallel multipliers are explored more than the others. It is also related only to fixed point multiplication, because floating point operation is based on fixed point operation and considered at a higher level.

A parallel multiplier is traditionally realized in the 2's complement number system. It also can be found that some non-traditional number systems, such as the SD number system and the RNS, are exploited for implementing a parallel multiplier. The SD number system attracted many researchers' interests recently especially regard to its application on parallel multipliers[Takagi 85, Kuninobu 87, Srinivas 91, Makino 93, Huang 94, Phatak 94].

Firstly, a review of the previous work in the literature is presented. Secondly, a general structure for the parallel multiplier is described, which outlines most designs of parallel multipliers. Thirdly, carry free adders, the core of a parallel multiplier, are briefly introduced. The redundant binary adder and the 5:3 counter, two types of the most widely used means for implementing the carry free adder, are studied separately. The converter from a binary SD number to a binary number is also studied. Later, a unified structure for the redundant binary adder and the 5:3 counter is investigated, and implementation comparison and evaluation for these two schemes are discussed. Finally, a variant of the Baugh and Wooley algorithm is generalized. This algorithm is employed in our previous design[Guan 94B], and also this design is further improved for large operands.

5.2 Review of Multipliers

Compared with an adder, a multiplier is more complex, since multiplication is generally implemented via a sequence of addition and shift operations. In the early days, because the hardware implementation was very expensive, multiplication was often performed by software or by employing an adder with some registers[Hwang 79]. With the advance of electronics, multipliers became a standard part for many logic systems. In design, there are several kinds of multipliers which can be chosen according to different applications.

A bit-serial multiplier (simply called serial multiplier) is the simplest form of multiplier. In most situations, it is assumed that the multiplicand bits are applied to the circuit in a parallel fashion and the multiplier bits are applied to the circuit in a serial fashion, therefore, a serial multiplier is also called a serial-parallel multiplier sometimes. The same assumption is used in the following.

A serial multiplier generates the bits of the product sequentially starting with *lsb*. In each step, one bit of the multiplier is examined to determine the multiplicand to be added or not. A serial multiplier is traditionally designed in carry save addition and shift structure, and for n bit operation, $2n$ clock cycles will be required to complete the process where n clock cycles are used for n row carry save additions, and the other n

clock cycles are utilized only to propagate the remaining carries. Gnanasekaran modified the traditional implementation[Gnanasekaran 85] so that his circuit operates in carry save addition and shift structure for the n first clock cycles and reconfigures itself in n bit ripple-carry addition structure at the $(n+1)$ clock cycle. This design eliminates the delay due to storage elements during the last n clock cycles, and results in about one-third increase in speed for an approximately one-third increase in hardware.

Another interesting design for a serial multiplier was presented by Ait-Boudaoud et al[Ait-Boudaoud 91]. They eliminate the broadcast of data over a long path which may reduce the clock frequency in VLSI implementation. As a result of this, the structure of the circuit is more modular when compared to the previous approaches, where the data moves from one cell to its adjacent cell sequentially. This has reduced enormously the effect of stray capacitance by the use of long paths. This original design is only for unsigned numbers. Recently, Moh and Yoon modified this result to a 2's complement case[Moh 95].

It is not difficult to find that in nearly all designs for a serial multiplier, the emphasis is on a structure and timing design rather than logic minimization, because in the structured design, the resulting basic building blocks are still FAs which are assumed to have been optimized.

Although a serial multiplier is simple, it can't satisfy many applications with high-speed, in this situation, a parallel multiplier is required. An array scheme can be employed for constructing a parallel multiplier in a very regular structure, but it is still considered to be not fast enough for many high-speed applications, since its speed is proportional to the length of operands. In order to further advance the speed, Wallace suggested a scheme for a fast parallel multiplier[Wallace 64], therefore, this scheme was termed the Wallace tree later. In his design, Wallace used FAs and realized a circuit in tree form to reduce n partial products (PPs) to two in a 3:2 ratio, then, a carry propagation adder was used to add these two PPs. An improvement on the Wallace tree has been made by Dadda[Dadda 65], who devised an arrangement that uses fewer FAs, although the same number of levels is required. Dadda found that the last level produces two PPs, the preceding level has at most 3, its predecessor at most 4, and so on. Thus, a series 2,3,4,6,9,13,19,28..... is formed, i.e. $\left\lfloor N \times \frac{3}{2} \right\rfloor$ is used to determine a following number, where $\lfloor x \rfloor$ means the greatest integer not greater than x and N stands for a number in the series (the least number of N is 2). In order to minimize the number of FAs in the total structure, the design is started by using only enough FAs in the first level to reduce the initial n PPs to one having a number of PPs equal to one in the series[Dadda 65, Scott 85]. Dadda also generalizes a parallel multiplier design in

two steps, one is carry free addition and the other is carry propagation addition. In carry free addition, he proposes so-called parallel *n:m* counters, in which, FA is only a special form of them. The carry free adder design based on parallel counters was further refined by Swartzlander et al[Swartzlander 73, Mehta 91]. In addition, the idea of a *n:m* counter was extended by Stenzel et al to include counters with inputs that have different weights[Stenzel 77], namely, these inputs can come from different columns in a PP matrix. These counters are described as $(c_{k-1}, c_{k-2}, \cdots, c_0, d)$ counters, where k is the number of input columns, c_i is the number of inputs in the column of weight 2^i, and d is the length of output as required. Counters like (5,5,4), (2,2,2,3,5) and (3,3,3,3,6) were suggested to reduce PP matrix height to two rows. In Stenzel et al's work, many complex counters are often based on ROM implementation, which is considered to be impractical for a practical application since ROMs are slow and occupy substantial area[Mehta 91]. Besides ROM implementation, it is hard to find an efficient circuit for these counters with more than, say, eight inputs, in the literature. In most practical designs, the inputs for a counter as a basic building block is five or less.

SD number representations were first introduced by Avizienis[Avizienis 61] for fast parallel arithmetic, especially for multiplication and division. Because of the redundancy in SD number representations, a carry propagation can be limited to two adjacent positions of the operands in addition and subtraction. In this way, a SD number adder can be used to perform carry free addition for a parallel multiplier. Avizienis presented a structure of SD number adder; this structure was further explored by Chow and Robertson[Chow 78] in binary SD form, where the SD number adder was called redundant binary adder. Because a number in binary SD representation has three values, i.e. $\{\bar{1}, 0, 1\}$, it is necessary to code it in binary logic, which is often termed format or coding. Chow and Robertson indicated that different formats will lead to different circuit complexity, and they demonstrated this by presenting some formats in logic form. They also presented a procedure for logical design of a redundant binary adder. The basic principle to design a redundant binary adder described by Chow and Robertson is widely employed.

The first design to use redundant binary adders in binary tree fashion for a parallel multiplier, probably, was proposed by Takagi et al[Takagi 85]. The speed of their design is considered to be almost the same as that by a multiplier with the Wallace tree, but the circuit is much more regular than that based on the Wallace tree. Therefore, this design is more suitable for VLSI implementation. Unfortunately, the redundant binary adder used in Takagi et al's design is too complex, so that this multiplier may be not as fast as expected.

Similarly to the case for FA, the minimum circuit for a redundant binary adder is also pursued by many researchers. One version with 42 transistors in CMOS was presented by Kuninobu et al[Kuninobu 87], and it is believed that it is the minimum redundant binary adder circuit known. For the same length of operands, different formats also require different number of levels to implement carry free addition for a parallel multiplier. This was found by Makino[Makino 93] et al and Huang et al[Huang 94] independently. In both of these two designs, a level of redundant binary adders is reduced when compared to the conventional implementation[Takagi 85, Kuninobu 87].

Another interesting application of binary SD representation is the attempt to combine binary SD representation with normal binary representation in design, which is called hybrid (signed-digit) number representations, and this can be found in some researchers' work[Srinivas 91, Phatak 94]. At present, the merit of introducing the hybrid number (system) representation in design is not clear, because the designs proposed by Srinivas et al and Phatak et al, in fact, have not improved previous designs.

Since some basic operations, such as addition, subtraction, and multiplication, are carry free, therefore, using RNS for a parallel multiplier has been of interest to some researchers. A recent design[Razavi 92] shows that 2's complement multiplier based on RNS still cannot compete with existing designs based on 2's complement number or binary SD number. A critical factor for this is that the conversion between a binary number and a residue number is too complex, especially for the conversion from a residue number to a binary number which is still a bottleneck problem for many practical applications.

In many situations, a direct and simple operation for 2's complement numbers is desirable. Baugh and Wooley have done significant work in this area. They presented an algorithm that can generate a PP matrix in which only adders are required instead of adders and subtracters[Baugh 73], and this algorithm is called the Baugh and Wooley algorithm. It is also possible to reduce PPs initially, and this has been demonstrated by a well-known algorithm termed the modified Booth algorithm that was developed by MacSorley[MacSorley 61] from Booth's previous work[Booth 51]. In his work, Booth presented a two bit scanning approach for multiplication, which was modified by MacSorley to three bit scanning, i.e. the modified Booth algorithm, and it has been further developed and generalized as multiple bit scanning technique[Vassiliadis 89, Sam 90]. It should be mentioned that the modified Booth algorithm can also be employed by a serial multiplier.

It can be also found that for some applications, a serial multiplier is too slow and a parallel multiplier is faster than necessary, in this situation, a digit serial multiplier can be employed[Parhi 90].

In multiplier design, the critical task of logic level design is to attempt to generate optimum basic circuits, particularly for the circuits which are used highly repeatedly in the design. The minimum logic implementation for some of the most widely used modules, such as FAs, flip-flops, MUXs, redundant binary adders, always attracts research interests.

5.3 A General Structure for Parallel Multiplier

A general structure for a parallel multiplier can be divided into three functional blocks, it is shown in Fig. 5.3.1

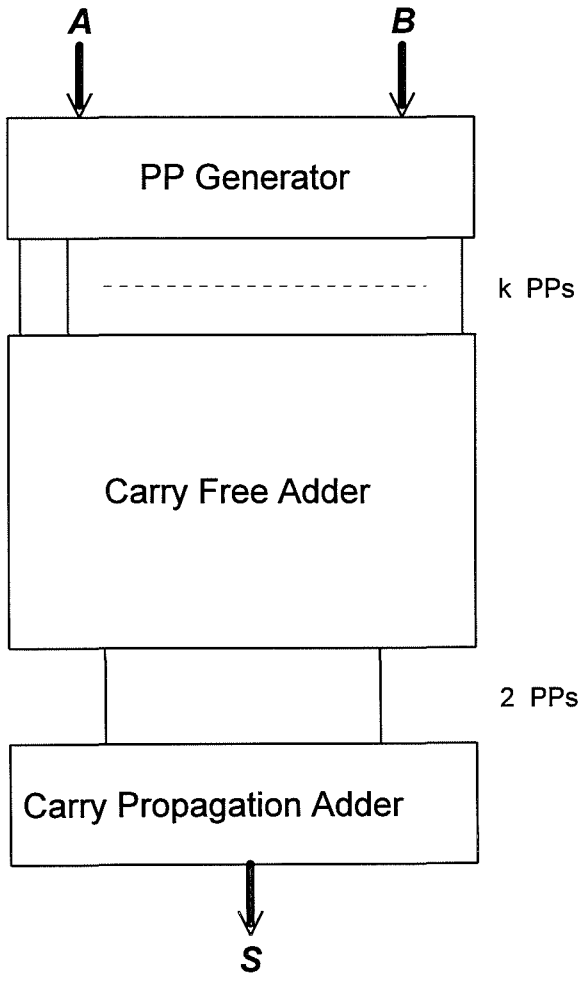


Fig. 5.3.1. A general structure for parallel multiplier.

These three blocks can be often studied separately and described as follows

Partial Product Generator (PPG):

PPG is used to generate a PP matrix. A straightforward method to generate a PP matrix can employ a group of AND gates, the result will include $k=n$ PPs, where n stands for the length of the multiplier. Although this method is simple, it will increase the complexity of CFA if 2's complement number is required. This is because both addition and subtraction are needed in CFA. The Baugh and Wooley algorithm can be used to solve this problem. The Baugh and Wooley algorithm will generate a PP matrix from which, only addition is required in CFA. This will simplify a design. The multiple bit scanning technique[Vassiliadis 89, Sam 90] may be used to reduce PPs initially in the matrix. In this case, the number of PPs is $k=n/(q-1)$, where q stands for the number of bits to be scanned. In general, the complexity of PPG will increase very quickly as q increases. Therefore, a large value of q is rarely employed. In most situations, $q=3$ is used, it is known as the modified Booth algorithm.

Carry Free Adder (CFA):

k PPs generated in PPG are applied to CFA in which, k PPs are handled in carry free addition fashion until two PPs are left, and these two PPs are applied to CPA. CFA can be realized using $n:m$ counters, e.g. 3:2 counter (i.e. full adder), 5:3 counter, 7:3 counter, etc., where n and m indicates the number of inputs and outputs, respectively. It is also possible to employ other number systems to implement CFA. Besides the 2's complement number system, the binary SD number system is commonly used, in which, a binary SD adder (i.e. the redundant binary adder) is required. In this case, a converter from binary SD numbers to 2's complement numbers is needed instead of a CPA. It should be mentioned that conversion from a 2's complement number to a binary SD number is not necessary, because the former can be considered to be a special case of the latter. CFA occupies more than half area of the chip for a parallel multiplier, therefore, many designs focus on how to reduce the area of CFA.

Carry Propagation Adder (CPA):

The two PPs from the CFA are applied to the CPA to calculate the final result. The CPA, which is actually a two operand adder described in Chapter 4, can't avoid a carry propagation problem. When the CFA is realized in the binary SD number system, as mentioned above, the converter is required to replace the CFA. In practice, the converter is quite similar to a two operand adder, this is because, nearly all algorithms

for a converter are derived from equation (4.2.7), which eventually leads to solving the same problem, a carry propagation, as that for a two operand adder.

The structure for a parallel multiplier described above is most widely employed in practical designs. So far, many algorithms and implementations for each one of the three functional blocks have been presented.
Another interesting method to design a multiplier with large operands is that, a multiplier can be first divided into some relatively smaller blocks, which is based on the following idea

$$\begin{aligned} P = A \times B &= (A_h \cdot A_l) \times (B_h \cdot B_l) \\ &= A_h \times B_h + A_h \times B_l + A_l \times B_h + A_l \times B_l \end{aligned} \tag{5.3.1}$$

where the subscripts h and l identify the high part and the low part, respectively, and the dot "$\cdot$" refers to concatenation. A study shows that this design method still leads to a circuit structure similar to that in Fig. 5.3.1[Mekhallalati 92].

5.4 Carry Free Adders

The CFA is usually realized employing n:m counters whose resulting circuits often rely largely on logic optimization methods. A general module of n:m counter is a combinational network with n inputs and m outputs where the outputs express the count of the number of inputs that are 1's. This can be illustrated by Fig. 5.4.1, and the relationship between the inputs and outputs is expressed by equation (5.4.1).

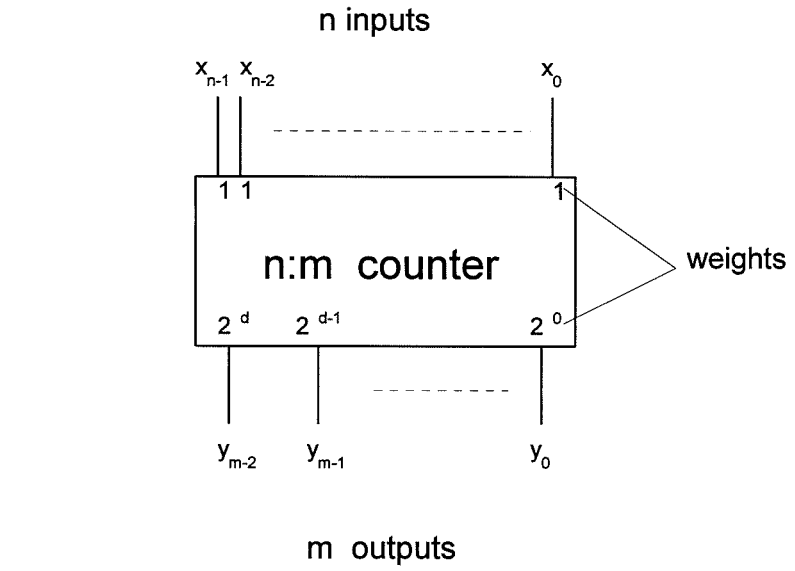


Fig. 5.4.1. A general module of *n*:*m* counter.

$$x_{n-1}+x_{n-2}+\cdots\cdots+x_0=y_{m-1}2^{d}+y_{m-2}2^{d-1}+\cdots\cdots+y_0 2^{0} \tag{5.4.1}$$

Where $x_i, y_i \in\{0,1\}$, d is chosen equal to $\lfloor \log_2 n \rfloor$, i.e., the largest integer that is less than or equal to $\log_2 n$.

An example in Fig. 5.4.2 is employed to show the CFA constructed using *n*:*m* counters, this is a 7:3 compressor with the 7:3 counters. The so-called 7:3 compressor means that the CFA can reduce PPs in a 7:3 ratio. Similarly, other compressors can be named.

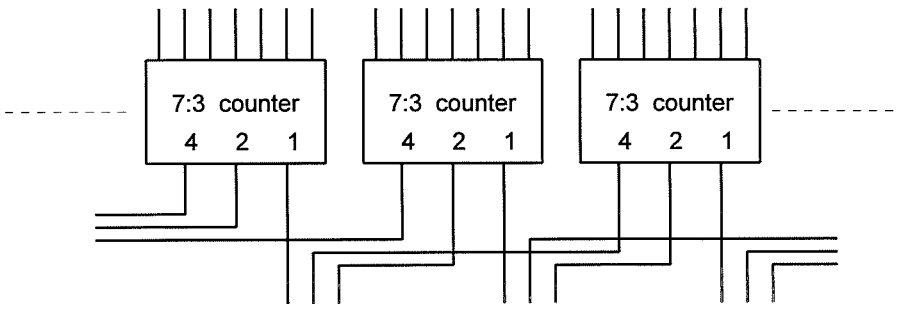


Fig. 5.4.2. A 7:3 compressor with 7:3 counters.

Since a carry free adder is realized using a lot of identical (or similar) modules, a critical factor for designing a CFA is to design the basic building block(s). An *n:m* counter can be implemented using various approaches, such as ROM, PLA, and the combinational circuit design approach. It can be also implemented based on FAs. The former two approaches, ROM and PLA, usually do not yield efficient circuits since both ROM and PLA are slow and occupy substantial area. When the approach based on FAs is used, this actually leads to a circuit similar to the Wallace tree, leaving some room for improvement. The combinational circuit design approach can generate a circuit which usually is faster and simpler than that based on the other approaches, and the result of the combinational circuit design approach may be significantly influenced by the means of logic minimization.

Although an arbitrary *n:m* counter can be employed for CFA implementation, in practice, the counters with smaller variables, such as FA, 5:3 counter and redundant binary adders, are used much more widely than that with larger variables. The main reason for this is that, it is very difficult to find a higher-performance circuit for the counters with larger variables, when compared to FA, 5:3 counter, and RBA.

5.5 Redundant Binary Adder (RBA)

It is well known that the main shortcoming of the Wallace tree is that it uses FAs to construct a CFA in a 3:2 ratio, and this leads to an irregular circuit which is difficult to realize in VLSI. In order to overcome this problem, a 4:2 compressor scheme is suggested, this can reduce a PP matrix to two PPs in a 4:2 ratio. This scheme will generate a regular and symmetric circuit that can be realized in binary tree fashion.

As stated previously, the CFA can be realized not only in the binary number system, but also in the binary SD (BSD) number system. In the BSD number system, a basic building module which is often called the RBA (the redundant binary adder)[Chow 78, Takagi 85, Huang 94] is used. This is shown in Fig. 5.5.1

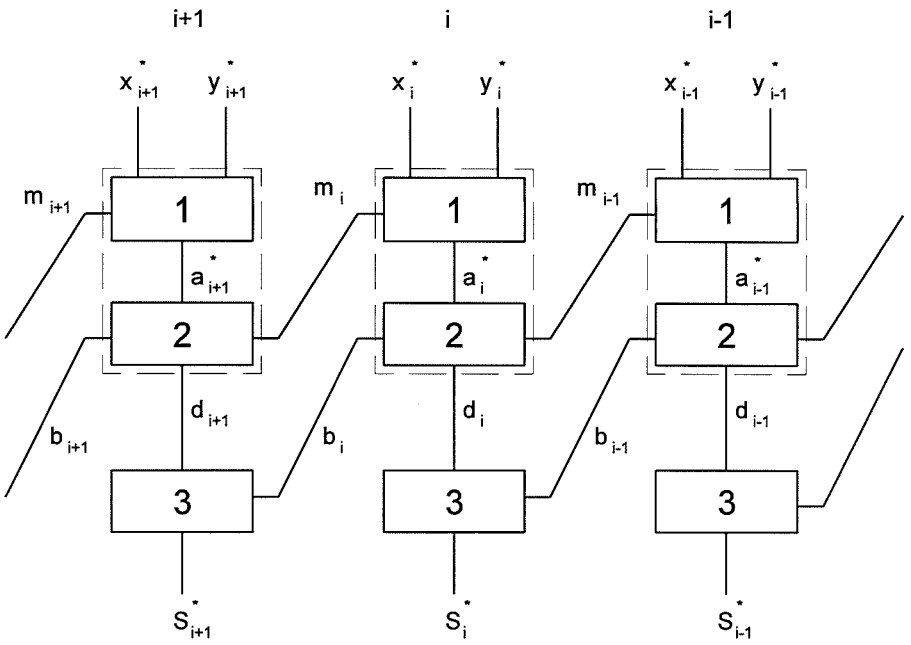


Fig. 5.5.1 The structure of RBA.

Every stage of the RBA has three blocks which are marked 1, 2 and 3. x_i^* and y_i^* are primary inputs, and S_i^* is primary output. m_i, a_i^*, d_i, and b_i are intermediate variables. The variables with "*" indicate that they have three states, for each of these variables, two bits will be required to represent it in binary logic. The RBA structure can be further decomposed into two levels, the upper level and lower level. The upper level is composed of blocks 1 and 2. The lower level consists of only block 3. Therefore, the mathematical relationships of RBA are:

$$x_i^* + y_i^* + m_{i-1} = d_i + 2m_i + 2b_i \tag{5.5.1}$$

$$d_i + b_{i-1} = S_i^* \tag{5.5.2}$$

Equation (5.5.1) is for the upper level and equation (5.5.2) for the lower level. Where, x_i^*, y_i^*, and $S_i^* \in \{\bar{1},0,1\}$, d_i and $m_i \in \{0,1\}$, and $b_i \in \{\bar{1},0\}$. It should be noted that a_i^* is eliminated.

Chow and Robertson studied nine types of codings for the number representations of a RBA and they called them formats[Chow 78]. The logic expansions of these nine formats based on Boolean logic were studied by Chow and Robertson. In this project, all functions of these nine formats based RM logic are studied using the RM logic

synthesis programs developed at Napier University. The study shows that only format 2 has a better result. Table 5.5.1 lists these results, the numbers in the table indicate the number of transistors required for a given format.

Table 5.5.1. RBA comparison between Boolean logic and RM logic.

Format	1	2	3	4	5	6	7	8	9
Boolean	80	50	52	64	52	76	70	74	80
RM	82	40	52	64	54	106	84	84	116

The structure for a parallel multiplier based on RBAs is shown as follows

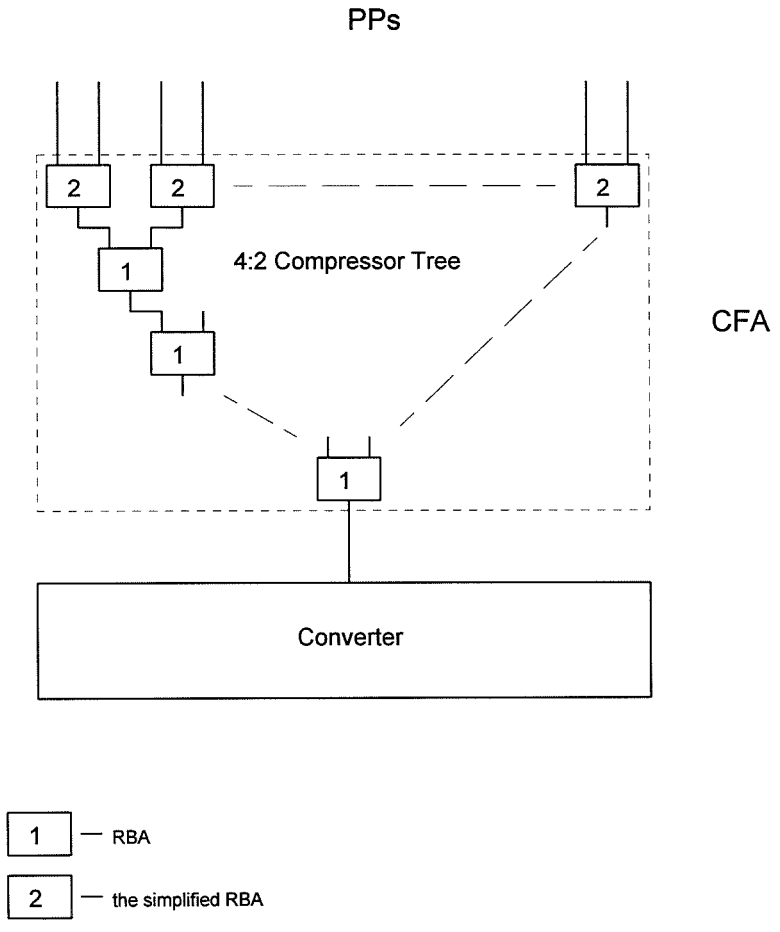


Fig. 5.5.2. A 4:2 compressor structure for parallel multiplier.

In Fig. 5.5.2, block 1 is a normal RBA, and block 2 is a simplified RBA. Because 2's complement number is a special form of BSD number, when PPs are applied to CFA, not only is a conversion from a 2's complement number to a BSD number

unnecessary, but also the adder of the first level can be further simplified[Kuninobu 87]. In general, a simplified RBA requires less than half of the components for a normal RBA and has half the delay of a normal RBA.

5.6 Conversion between 2's Complement and RBSD Numbers

When a multiplier is designed using RBA, the resulting BSD number has to be converted back to a 2's complement number. In practice, CPA and the converter are very similar, the main reason is that both of them require a carry (borrow) propagation operation. This is because nearly all algorithms for the converter are derived from equation (4.2.7), as mentioned earlier. The main difference between them is that the converter may be simplified if some coding, e.g. format 2, is used[Yen 92].
In principle, both CPA and the converter contain three parts, i.e., Carry (or Borrow) Control Logic (CCL), Carry (or Borrow) Logic (CL), and Summation Logic (SL), which is illustrated in Fig. 5.6.1

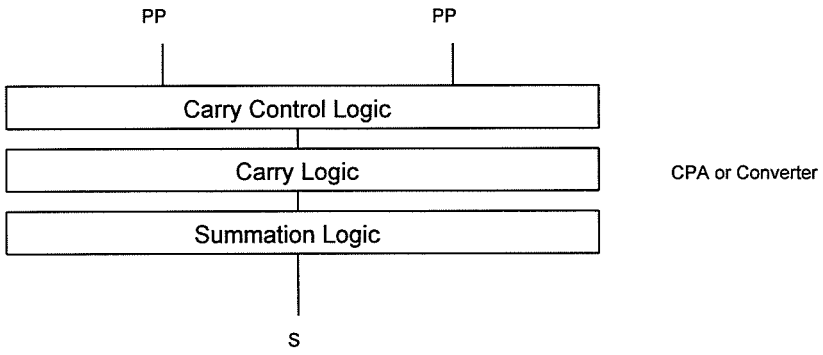


Fig. 5.6.1. The unified structure for CPA and the converter.

The two PPs, generated from the CFA, are applied to the CCL. The CCL consists of two simple functions, carry (borrow) generation g_i and carry (borrow) propagation p_i. The CL can employ many speed-up techniques, such as the carry lookahead, the carry skip, the carry select, the Manchester carry chain, and so on. The SL is very simple, with one bit corresponding to one XOR gate.
Two generalized algorithms for the converter are described in the following, both of them are independent of format used.

Algorithm 1 (based on addition):

carry generation $g_i = 1$ if $x_i^* = 1$, otherwise, $g_i = 0$;

carry propagation $p_i = 1$ if $x_i^* = 0$, otherwise, $p_i = 0$;

carry $c_i = c_{i-1}$ if $p_i = 1$, otherwise, $c_i = g_i$ $(c_{-1} = 1)$;

sum $S_i = c_{i-1} \oplus p_i$;

final result is: $\sum_{i=0}^{k-1} 2^i \times S_i + 1$.

Algorithm 2 (based on subtraction):

borrow generation $g_i = 1$ if $x_i^* = -1$, otherwise, $g_i = 0$;

borrow propagation $p_i = 1$ if $x_i^* = 0$, otherwise, $p_i = 0$;

borrow $c_i = c_{i-1}$ if $p_i = 1$, otherwise, $c_i = g_i$ $(c_{-1} = 0)$;

sum $S_i = c_{i-1} \oplus \bar{p}_i$;

final result is: $\sum_{i=0}^{k-1} 2^i \times S_i$.

These two algorithms are very similar because addition and subtraction of 2's complement numbers are very similar. Note that g_i, p_i, and c_i, in circuit, are identical for the CL and SL circuits, therefore, the same symbols are used irrespective of carry or borrow. Furthermore, CL and SL can be considered format-independent, i.e., they are the same for all formats and CPA.

It should be mentioned that On-the fly algorithm[Ercegovac 87] for a converter used in a multiplier without CPA[Ercegovac 90], in fact, is a sequential operation[Kornerup 94], therefore, this algorithm is probably not very suitable for a general parallel multiplier.

5.7 5:3 Counter

In order to construct a 4:2 compressor in the 2's complement system, a structure similar to that of RBA[Chow 78] is employed. This is shown in Fig. 5.7.1

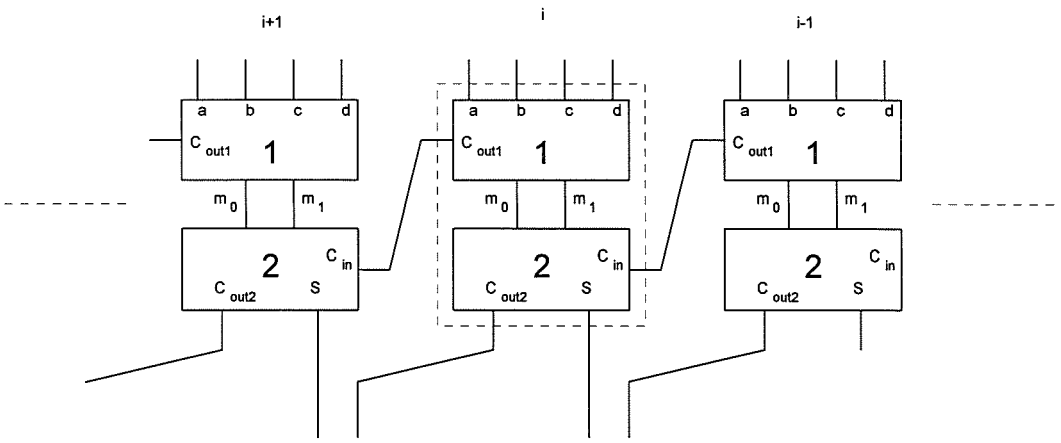


Fig. 5.7.1. A 4:2 compressor based on 5:3 counter.

In Fig. 5.7.1, it can be seen that the basic building module is divided into two blocks and each module includes five input variables, a, b, c, d, and C_{in}, and three outputs, S, C_{out1}, and C_{out2}. m_0 and m_1 are intermediate variables, in the design, they can not be concerned. The relationship between the inputs and the outputs can be expressed as follows

$$a+b+c+d+C_{in} = S+2C_{out1}+2C_{out2} \tag{5.7.1}$$

It can be seen that this is a 5:3 counter, but it is different from the normal one described previously where, every output has a different weight. In this 5:3 counter, there are two outputs which have the same weight. Another important difference from the normal one is that, in this 5:3 counter, one output C_{out1} depends only on a, b, c, and d, which are termed local variables. C_{out1} is connected to C_{in} of the next stage. The outputs S and C_{out2} are connected to the inputs of next level. In this way, a carry is limited within two adjacent stages. The principle of carry free for a 5:3 counter is similar to that for the RBA.

In the design of this special 5:3 counter, S can be easily obtained, it is

$$S = C_{in} \oplus a \oplus b \oplus c \oplus d \tag{5.7.2}$$

One thing should be dealt with, that is, when two or three of five input variables are equal to "1", one of two carries, C_{out1} and C_{out2}, should be set to "1", this is illustrated in Table 5.7.1

Table 5.7.1. The simplified truth table for 5:3 counter.

Value	S	C_{out1}	C_{out2}
0	0	0	0
1	1	0	0
2	0	?	?
3	1	?	?
4	0	1	1
5	1	1	1

In Table 5.7.1, Value stands for the number of the five input variables equal to one, "?" means that either C_{out1} or C_{out2} is set to one, but not both. In other words, it requires the function of one carry to be decided first. Because C_{out1} is restricted more strictly than C_{out2} (C_{out1} depends on the four local input variables and C_{out2} depends on all the input variables), the function of C_{out1} should be determined first.

In principle, C_{out1} is set to one when any two of the four local variables equal one, therefore, C_{out1} can be derived from any two or any three of the four local input variables, or all four. In practice, the number of the variables can only be selected as three or four, this can be verified using a complete truth table. Appropriate selection of the number of input variables for C_{out1} will make a circuit simple.

In this design, three local variables are selected for C_{out1}. Assume that these three variables are *b*, *c*, and *d*. In fact, this is the carry function of the FA known as the majority function, therefore, according to equation (3.7.3), C_{out1} can be expressed as

$$\begin{aligned} C_{out1} &= bc \oplus bd \oplus cd \\ &= bc \oplus bd \oplus cd \oplus d\bar{d} \quad (\textit{adding term } d\bar{d} = 0 \textit{ doesn't affect the validity of equation}) \\ &= b(c \oplus d) \oplus d(c \oplus \bar{d}) \\ &= b(c \oplus d) \oplus d(\overline{c \oplus d}) \quad (\textit{according to equation } (2.2.9)) \end{aligned} \tag{5.7.3}$$

After C_{out1} has been decided, this also decides a complete truth table for the 5:3 counter, it is shown by Table 5.7.2.

Table 5.7.2. The truth table for 5:3 counter.

C_{in}	a	b	c	d	Value	S	C_{out1}	C_{out2}
0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0
0	0	0	1	0	1	1	0	0
0	0	0	1	1	2	0	1	0
0	0	1	0	0	1	1	0	0
0	0	1	0	1	2	0	1	0
0	0	1	1	0	2	0	1	0
0	0	1	1	1	3	1	1	0
0	1	0	0	0	1	1	0	0
0	1	0	0	1	2	0	0	1
0	1	0	1	0	2	0	0	1
0	1	0	1	1	3	1	1	0
0	1	1	0	0	2	0	0	1
0	1	1	0	1	3	1	1	0
0	1	1	1	0	3	1	1	0
0	1	1	1	1	4	0	1	1
1	0	0	0	0	1	1	0	0
1	0	0	0	1	2	0	0	1
1	0	0	1	0	2	0	0	1
1	0	0	1	1	3	1	1	0
1	0	1	0	0	2	0	0	1
1	0	1	0	1	3	1	1	0
1	0	1	1	0	3	1	1	0
1	0	1	1	1	4	0	1	1
1	1	0	0	0	2	0	0	1
1	1	0	0	1	3	1	0	1
1	1	0	1	0	3	1	0	1
1	1	0	1	1	4	0	1	1
1	1	1	0	0	3	1	0	1
1	1	1	0	1	4	0	1	1
1	1	1	1	0	4	0	1	1
1	1	1	1	1	5	1	1	1

From the complete truth table, initially, C_{out2} can be represented as

$$C_{out2} = C_{in}\bar{a} \oplus C_{in}d \oplus C_{in}\bar{c} \oplus C_{in}b \oplus a\bar{b} \oplus a\bar{d} \oplus ac \; * \tag{5.7.4}$$

**This initial expression is got from a RM logic synthesis program developed at Napier University.*

Similarly to C_{out1}, C_{out2} can be derived as

$$\begin{aligned} C_{out2} &= C_{in}\bar{a} \oplus C_{in}d \oplus C_{in}\bar{c} \oplus C_{in}b \oplus a\bar{b} \oplus a\bar{d} \oplus ac \\ &= C_{in}(\bar{a} \oplus b \oplus \bar{c} \oplus d) \oplus a(\bar{b} \oplus c \oplus \bar{d}) \\ &= C_{in}(a \oplus b \oplus c \oplus d) \oplus a(b \oplus c \oplus d) \\ &= C_{in}(a \oplus b \oplus c \oplus d) \oplus a(\bar{a} \oplus b \oplus c \oplus d) \\ &= C_{in}(a \oplus b \oplus c \oplus d) \oplus a(\overline{a \oplus b \oplus c \oplus d}) \end{aligned} \tag{5.7.5}$$

It can be found that equations (5.7.2), (5.7.3), and (5.7.5) include many common factors that can be shared to construct a simpler circuit[Guan 95] as shown in Fig. 5.7.3

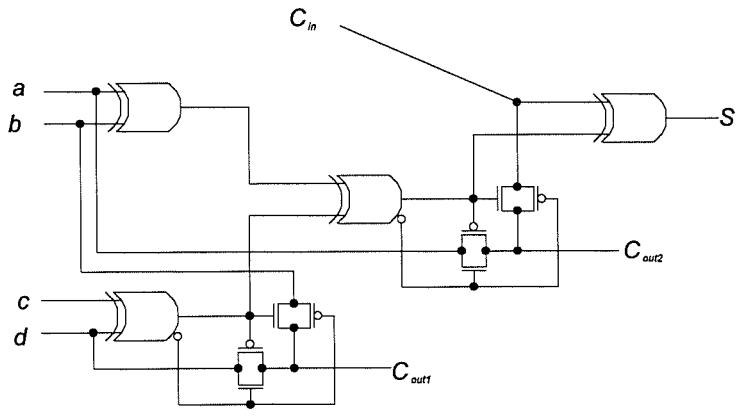


Fig. 5.7.3. A 5:3 counter.

This design is based on RM logic, and it is believed that equation (5.7.2) is the most compact expansion for five variable mod 2 sum, which also leads to the minimum implementation by using four two-input XOR gates. The other two variables, C_{out1} and C_{out2} are handled so that their expansions match that of S to generate common factors as much as possible[Guan 95]. In this case, it can be seen that the design based on RM logic is better than that based on Boolean logic, since it is not easy for the initial SOP forms for C_{out1} and C_{out2} to match equation (5.7.2).
From this design, it can be seen that RM logic, sometimes, can solve a problem which is not easily solved in Boolean logic.

5.8 A Unified Structure for 4:2 Compressor

It can be easily seen that the range of values for equation (5.5.1) is {-2~3}. If equation (5.5.1) is biased (scaled) by a constant 2, namely, a constant 2 is added to both sides of equation (5.5.1), then its range of value becomes {0~5}. In practice, it is not necessary for a constant 2 to appear in equation (5.5.1), because the constant is assimilated by variables, x_i^*, y_i^* and b_i. That is, x_i^*, y_i^* and b_i are biased by 1. As a result, x_i^* and $y_i^* \in \{0,1,2\}$, and $b_i \in \{0,1\}$.
As stated above, in binary logic, two bits will be required to represent x_i^*, y_i^*, or S_i^*. Assume that x_i^*, y_i^*, and S_i^* are represented by pairs $x_i^1 x_i^2$, $y_i^1 y_i^2$ and $S_i^1 S_i^2$, respectively. Consequently, equation (5.5.1) can be rewritten in the following form

$$x_i^1 x_i^2 + y_i^1 y_i^2 + m_{i-1} = d_i + 2m_i + 2b_i \tag{5.8.1}$$

If it is defined that x_i^1, x_i^2, y_i^1 and $y_i^2 \in \{0,1\}$, and $x_i^1 x_i^2 = x_i^1 + x_i^2$, $y_i^1 y_i^2 = y_i^1 + y_i^2$, equation (5.8.1) becomes equation (5.8.2)

$$x_i^1 + x_i^2 + y_i^1 + y_i^2 + m_{i-1} = d_i + 2m_i + 2b_i \tag{5.8.2}$$

Equation (5.8.2) represents a 5:3 counter, and its range of value is {0~5}. Compared with equation (5.7.1), there is nothing different but the symbols used. According to

the analysis above, the upper level of the RBA and the 5:3 counter have a similar circuit structure and similar relationships between inputs and outputs. The difference between them is that, firstly, in the mathematical relationship, the 5:3 counter is a RBA biased by a constant 2. Secondly, in logic design, the number representation of the 5:3 counter is fixed, but for the RBA, there exist many different number representations, which are often called formats or codings as mentioned earlier, which is why block 3 (see Fig. 5.5.1) is required. Block 3 is used to transfer the intermediate variables d_i and b_{i-1} to $S_i^1 S_i^2$ in a given format.

Broadly speaking, the upper level of RBA (block 1 and 2) is a type of 5:3 counter termed biased 5:3 counter, because there is a simple addition relationship between its five inputs and three outputs. In other words, a RBA, in circuit structure, can be considered as a 5:3 counter with a coding circuit, this coding circuit is block 3.

5.9 Implementation Comparison and Evaluation

The circuit complexity for the RBA may vary depending on the format used. Chow and Robertson presented formats 1~9 in Table 5.9.1, and concluded that formats 2, 4, and 5 have the simplest logic representations[Chow 78]. In practice, according to the author's design and a survey of the literature, the circuit for format 2 is simpler and faster than that for formats 4 and 5.

Table 5.9.1. Various formats for RBA and their p_i and g_i functions.

x_i^*	Codings (Formats)											
$x_i^1 x_i^2$	1	2	3	4	5	6	7	8	9	10	11	CPA
00	0	0	0	0	d	0	0	1	-1	0	0	0
01	1	1	1	1	1	1	1	1	1	-1	-1	1
10	-1	d	0	-1	0	1	-1	0	0	1	1	1
11	0	-1	-1	d	-1	-1	-1	-1	-1	d	0	2
p_i	$\overline{x_i^1 \oplus x_i^2}$	$\overline{x_i^2}$	$\overline{x_i^2}$	$\overline{x_i^1 + x_i^2}$	$\overline{x_i^2}$	$\overline{x_i^1 + x_i^2}$	$\overline{x_i^1 + x_i^2}$	$x_i^1 \overline{x_i^2}$	$x_i^1 \overline{x_i^2}$	$\overline{x_i^1 + x_i^2}$	$\overline{x_i^1 \oplus x_i^2}$	$x_i \oplus y_i$
g_i	$x_i^1 \overline{x_i^2}$	x_i^1	$x_i^1 x_i^2$	x_i^1	$x_i^1 x_i^2$	$x_i^1 x_i^2$	x_i^1	$x_i^1 x_i^2$	$\overline{x_i^1 \oplus x_i^2}$	x_i^2	$\overline{x_i^1} x_i^2$	$x_i y_i$

Note: d means don't care.

It should be noted that a function with the simplest logic representation in theory may not always generate the minimum circuit implementation in some circuit technologies. Here, instead of theoretical analysis, a survey of practical designs in CMOS circuits is given, and a comparison and evaluation based on this survey is presented. Table 5.9.1 lists various codings from the survey. The complexities of g_i and p_i are also shown in the table.

Formats 1~9 are those shown by Chow and Robertson, and all their logic representations for RBA can be found in [Chow 78]. Of the nine formats, format 1 is used by Balakrishnan and Burgess[Balakrishnan 92], and is also employed to design a parallel multiplier in hybrid number representation[Srinivas 91]; format 2 is used in [Kuninobu 87, Yen 92]; format 4 is used in [Harata 87, Rajashekhara 90]. Formats 3, 5~9 are hardly used.

Format 10, probably, was the first one used for a parallel multiplier[Takagi 85]. Format 11 is a new one introduced recently by Makino et al[Makino 93] and Huang et al[Huang 94] independently. In the following, only format 2, format 11, and the 5:3 counter will be discussed, because the circuits for format 2 and format 11 are superior to those for the other codings in the literature.

For convenience, some symbols should be defined first. With reference to Fig. 5.9.1

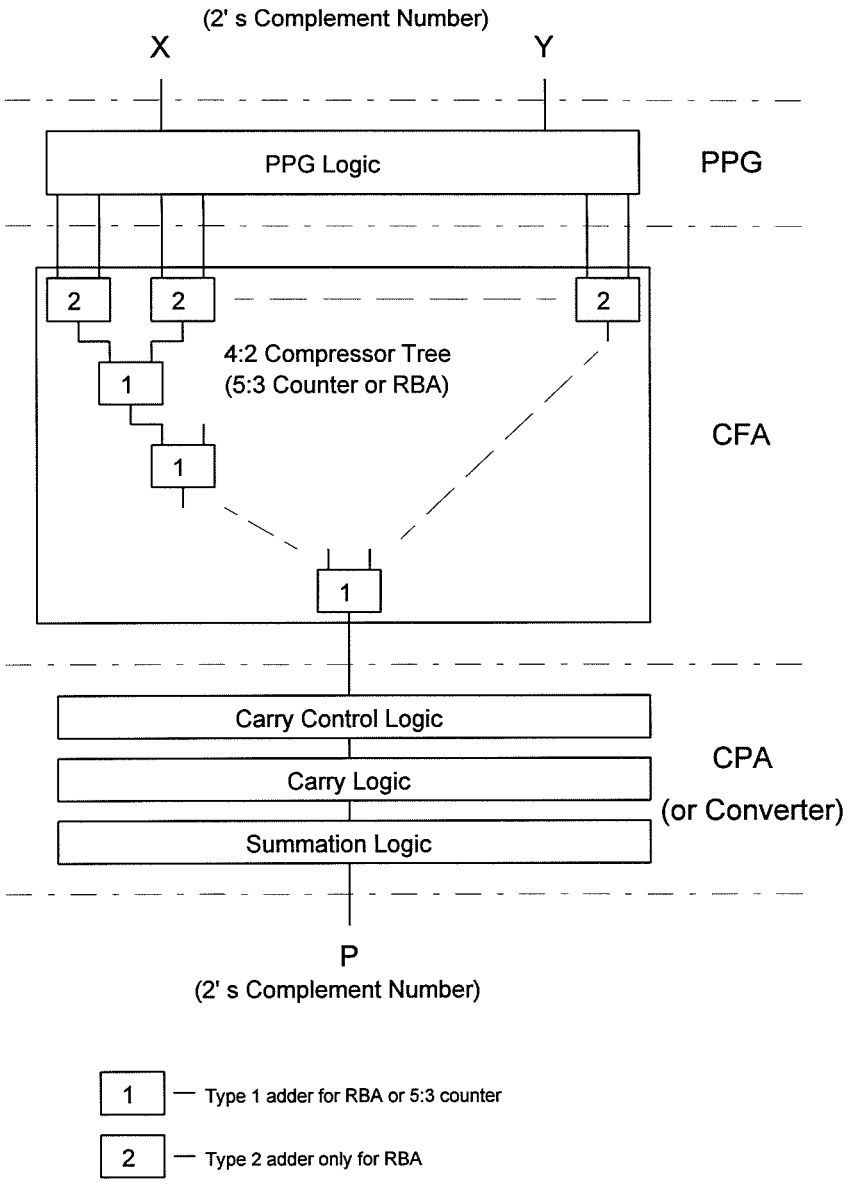


Fig. 5.9.1. A functional structure for parallel multiplier based on 4:2 compressor.

The structure in Fig. 5.9.1 is for both a multiplier based on RBA and that based on 5:3 counter, it comes from a combination of Fig. 5.3.1, Fig. 5.5.2, and Fig. 5.6.1. The operation time for each functional block is defined as follows:

T_{PPG} - the operation time of PPG;

T_{CFA1} - the operation time of type 1 RBA addition tree;

T_{CFA2} - the operation time of type 2 RBA level;

T_{CCL} - the operation time of carry control logic;

T_{CL} - the operation time of carry logic;

T_{SL} - the operation time of summation logic.

Similarly, the implementation costs are: A_{PPG}, A_{CFA1}, A_{CFA2}, A_{CCL}, A_{CL}, and A_{SL}.

(i). format 2:

This is a natural coding for RBA[Kuninobu 87, Yen 92]. That is, the first bit is the sign bit, and the second bit is the magnitude bit. This format may yield a simpler RBA. Another advantage for this format is that, it simplifies the CCL circuit, see Table 5.9.1. A possible circuit to implement the converter is shown in Fig. 5.9.2

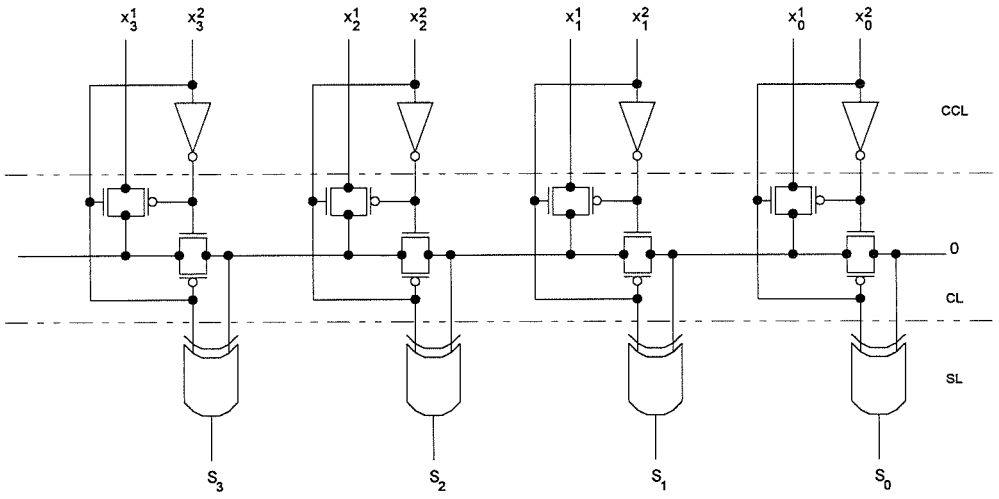


Fig. 5.9.2. A 4 bit redundant to binary converter.

The implementation time of a parallel multiplier based on format 2 is:

$$T_2 = T_{PPG} + T_{CFA1} + T_{CFA2} + T_{CL} + T_{SL} \tag{5.9.1}$$

The implementation cost is:

$$A_2 = A_{PPG} + A_{CFA1} + A_{CFA2} + A_{CL} + A_{SL} \tag{5.9.2}$$

(ii). format 11:

format 11 is based on the following operation in 2's complement number:

$$X + Y = X - (-Y) = X - \overline{Y} - 1 \tag{5.9.3}$$

$\overline{Y}$ is 1's complement of Y, i.e., $\overline{y_i} = 1 - y_i$, $y_i \in \{0,1\}$.

The main advantage for this new coding is that in the CFA, type 2 RBA level is not needed. The RBA circuit based on format 11 may be more complex than that based on format 2.

The implementation time of a parallel multiplier based on format 11 is:

$$T_{11} = T_{PPG} + T_{CFA1} + T_{CCL} + T_{CL} + T_{SL} \tag{5.9.4}$$

The implementation cost is:

$$A_{11} = A_{PPG} + A_{CFA1} + A_{CCL} + A_{CL} + A_{SL} \tag{5.9.5}$$

(iii). 5:3 Counter:

The 5:3 counter and RBA, as stated earlier, have similar circuit structure. In fact, a 5:3 counter and a RBA with format 11 can be considered identical if x_i^2 and y_i^2 in the RBA are complemented. This can be verified by Table 5.9.2. In Table 5.9.2, one point should be mentioned, that is, for 5:3 counter, $b_i = 0$ means 0 and $b_i = 1$ means 2; but for RBA, $b_i = 0$ means -2 and $b_i = 1$ means 0. It can be found that C_{in}, a, b, c, d, S, C_{out1}, and C_{out2} for 5:3 counter are equal to m_{i-1}, x_i^1, x_i^2, y_i^1, y_i^2, d_i, m_i, and b_i for a RBA with format 11, respectively. In other words, a circuit has two different explanations. Therefore, the operation time and implementation cost for a parallel multiplier based on 5:3 counter are the same as equations (5.9.4) and (5.9.5) respectively.

Table 5.9.2. A truth table for both 5:3 counter and RBA with format 11.

m_{i-1}	x_i^1	x_i^2	y_i^1	y_i^2	5:3	RBA	d_i	m_i	b_i
0	0	0	0	0	0	-2	0	0	0
0	0	0	0	1	1	-1	1	0	0
0	0	0	1	0	1	-1	1	0	0
0	0	0	1	1	2	0	0	1	0
0	0	1	0	0	1	-1	1	0	0
0	0	1	0	1	2	0	0	1	0
0	0	1	1	0	2	0	0	1	0
0	0	1	1	1	3	1	1	1	0
0	1	0	0	0	1	-1	1	0	0
0	1	0	0	1	2	0	0	0	1
0	1	0	1	0	2	0	0	0	1
0	1	0	1	1	3	1	1	1	0
0	1	1	0	0	2	0	0	0	1
0	1	1	0	1	3	1	1	1	0
0	1	1	1	0	3	1	1	1	0
0	1	1	1	1	4	2	0	1	1
1	0	0	0	0	1	-1	1	0	0
1	0	0	0	1	2	0	0	0	1
1	0	0	1	0	2	0	0	0	1
1	0	0	1	1	3	1	1	1	0
1	0	1	0	0	2	0	0	0	1
1	0	1	0	1	3	1	1	1	0
1	0	1	1	0	3	1	1	1	0
1	0	1	1	1	4	2	0	1	1
1	1	0	0	0	2	0	0	0	1
1	1	0	0	1	3	1	1	0	1
1	1	0	1	0	3	1	1	0	1
1	1	0	1	1	4	2	0	1	1
1	1	1	0	0	3	1	1	0	1
1	1	1	0	1	4	2	0	1	1
1	1	1	1	0	4	2	0	1	1
1	1	1	1	1	5	3	1	1	1

From the above analysis, it can be seen that comparison between the two schemes (one based on 5:3 counter and the other based on RBA) can, in practice, be reduced to comparison of their CFAs, which relies heavily on their basic cells, since PPG, CL, and SL are the same for both. Additionally, the implementation cost for CCL is very low, compared with CFA. In Table 5.9.3, some best basic cells realized in CMOS are listed.

Table 5.9.3. Comparison of some best basic cells.

Reference	[Kun.87]	[Pha.94]	[Mak.93]	[Nag.90]	[Got.92]	[Gua.94B]
Type	format 2	format 2	format 11	5:3	5:3	5:3
Transistors	42	40*	48	54	52	34**
Delay	4.5	5	4	3.5	5	3
Levels	$\lceil \log_2 n \rceil$	$\lceil \log_2 n \rceil$	$\lceil \log_2 n \rceil$-1	$\lceil \log_2 n \rceil$-1	$\lceil \log_2 n \rceil$-1	$\lceil \log_2 n \rceil$-1

* There are two types of RBA required, their average value is given;

**There are two types of RBA required, their average value is given (including buffers, see later).

In order to achieve high speed circuits, it is assumed that a single XOR gate is realized by six transistors in pass transistor form (see Chapter 3); a two input gate is considered to have 1 gate delay without regarding its type; a three input gate except X(N)OR gate and a four input gate except X(N)OR gate are considered to have 1.5 gate delay and 2 gate delay, respectively. The realization of X(N)OR gate is different from the other types of gates due to its poor extendibility.

The levels of CFA required are given in the bottom row, n is the number of partial products, and the ceiling $\lceil X \rceil$ of a number is the smallest integer that is larger than or equal to X. It can be seen that format 2 requires one level adder more than the others, corresponding to type 2 RBA.

From Table 5.9.3 and equations (5.9.1), (5.9.2), (5.9.4), and (5.9.5), it can be seen that the design of [Kuninobu 87] can eliminate the CCL, but the implementation cost of the first level (type 2 RBA) can't offset the gain. Even though type 2 RBA is simple, the number of type 2 is large due to the CFA in binary tree form. This is particularly true for large operands. The design of [Phatak 94] has the same problem although it is considered by the authors to reduce part of the global wires. It should be noted that the CFA under consideration is in binary form, that is, reducing one level adder not only reduces the number of adders, but also substantially reduces its global wires.

The designs in the fourth column to the last column from left can be evaluated by equations (5.9.4) and (5.9.5), it is obvious that the design of [Guan 94B] is the best, in

terms of speed and cost. In the following two sections, our previous design[Guan 94B] will be further generalized.

5.10 A Variant of Baugh and Wooley Algorithm

In our previous design[Guan 94B], a variant of the Baugh and Wooley algorithm is used to generate the PP matrix. This algorithm for $n \times n$ multiplier has been studied by some researchers[Guan 94B]. This algorithm is superior to the original Baugh and Wooley algorithm because it doesn't yield additional rows for the PP matrix, but this algorithm is only limited to the situation in which the length of multiplicand is equal to that of multiplier. Here, the algorithm is developed to be suitable for the more general situation in which two operands have different lengths. The resultant PP matrix is still without additional rows.

Two operands A and B in 2's complement number can be expressed as

$$A = -a_{m-1}2^{m-1} + \sum_{i=0}^{m-2} a_i 2^i \quad \text{and} \quad B = -b_{n-1}2^{n-1} + \sum_{j=0}^{n-2} b_j 2^j \tag{5.10.1}$$

For convenience, it is assumed that $m \geq n$. In fact, this condition is not necessary, because multiplication is commutative, i.e., $P = A \times B = B \times A$. Therefore, the shorter of two operands is always considered to be the multiplier.

the product P of A and B is

$$\begin{aligned} P = A \times B &= (-a_{m-1}2^{m-1} + \sum_{i=0}^{m-2} a_i 2^i) \times (-b_{n-1}2^{n-1} + \sum_{j=0}^{n-2} b_j 2^j) \\ &= a_{m-1}b_{n-1}2^{m+n-2} + \sum_{i=0}^{m-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} - 2^{m-1}\sum_{j=0}^{n-2} a_{m-1}b_j 2^j - 2^{n-1}\sum_{i=0}^{m-2} a_i b_{n-1} 2^i \end{aligned} \tag{5.10.2}$$

Equation (5.10.2) includes two negative terms. It is desirable to have all terms positive, because this will permit straightforward addition of all summands in each column of the PP matrix. That is, the negation of the negative summands is added while still maintaining a mathematically correct equation for the product P.
An example is first used to show the following substitution:

$$0\ \bar{1}\ 0\ \bar{1}\ \bar{1} = \bar{1} \times (\ 01011\ ) = (\ 10100\ ) + 1 = 10101$$

This means that a non-positive integer composed only of 0 and $\bar{1}$ bits can be transferred to its 2's complement representation by changing $\bar{1}$ to 0 and 0 to 1 and adding 1 to *lsb*. This operation can be expressed as

$$x_{k-1}2^{k-1} + x_{k-2}2^{k-2} + \ldots\ldots x_2 2^2 + x_1 2^1 + x_0 2^0 = \left\lceil \sum_{0}^{k-1} (1 - |x_i|) 2^i \right\rceil + 1 \qquad (5.10.3)$$

where $x_i = \{0, \bar{1}\}$, and k is the length of the number.
It should be noted that the non-positive number under consideration is a non-redundant number, in binary redundant number system, $x_i = \{1, 0, \bar{1}\}$.
In binary logic, one signal line usually corresponds to one bit. For example, in positive logic, high voltage indicates 1 and low voltage indicates 0, which means that $\bar{1}$ can only be implied in signal lines. In other words, differentiating $\bar{1}$ from 1 doesn't depend on voltage value, but on signals. That is, for a_{m-1} and b_{n-1}, high voltage is $\bar{1}$ and low voltage is 0; for all the other bits, high voltage is 1 and low voltage is 0.
According to the above analysis, the operation $1 - |x_i|$, in binary logic, can be achieved by NAND operation. Therefore, the first negative term in equation (5.10.2) can be rewritten as follows

$$\begin{aligned} -2^{m-1} \sum_{j=0}^{n-2} a_{m-1} b_j 2^j &= 2^{m-1} (2^n + 2^{n-1} + 1 + \sum_{j=0}^{n-2} (1 - |a_{m-1} b_j|) 2^j) \\ &= 2^{m-1} (2^n + 2^{n-1} + 1 + \sum_{j=0}^{n-2} \overline{a_{m-1} b_j}\, 2^j) \end{aligned} \qquad (5.10.4)$$

It is worth mentioning that there are two constant terms 2^n and 2^{n-1} in equation (5.10.4), this is because according to 2's complement, the sign bit should be extended to *msb*. The *msb* for product P is bit $m+n-1$.
Similarly, the second negative term is

$$-2^{n-1}\sum_{i=0}^{m-2} a_i b_{n-1} 2^i = 2^{n-1}(2^m + 2^{m-1} + 1 + \sum_{i=0}^{m-2} \overline{a_i b_{n-1}} 2^i) \tag{5.10.5}$$

When the two negative terms in equation (5.10.2) are substituted by equations (5.10.4) and (5.10.5), equation (5.10.2) can be rewritten as

$$\begin{aligned}
P &= a_{m-1} b_{n-1} 2^{m+n-2} + \sum_{i=0}^{m-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} \\
&+2^{m-1}(2^n + 2^{n-1} + 1 + \sum_{j=0}^{n-2} \overline{a_{m-1} b_j} 2^j) + 2^{n-1}(2^m + 2^{m-1} + 1 + \sum_{i=0}^{m-2} \overline{a_i b_{n-1}} 2^i) \\
&= a_{m-1} b_{n-1} 2^{m+n-2} + \sum_{i=0}^{m-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} + 2^{m-1} \sum_{j=0}^{n-2} \overline{a_{m-1} b_j} 2^j + 2^{n-1} \sum_{i=0}^{m-2} \overline{a_i b_{n-1}} 2^i \\
&+2^{m+n} + 2^{m+n-1} + 2^{m-1} + 2^{n-1} \\
&= a_{m-1} b_{n-1} 2^{m+n-2} + \sum_{i=0}^{m-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} + 2^{m-1} \sum_{j=0}^{n-2} \overline{a_{m-1} b_j} 2^j + 2^{n-1} \sum_{i=0}^{m-2} \overline{a_i b_{n-1}} 2^i \\
&+2^{m+n-1} + 2^{m-1} + 2^{n-1}
\end{aligned} \tag{5.10.6}$$

Note that 2's complement number can be considered to be a special case of residue number, namely, $p = |P|_{2^{m+n}}$. Therefore, the constant term 2^{m+n} in equation (5.10.6) can be eliminated.
The PP matrix of equation (5.10.6) is shown in Fig. 5.10.1

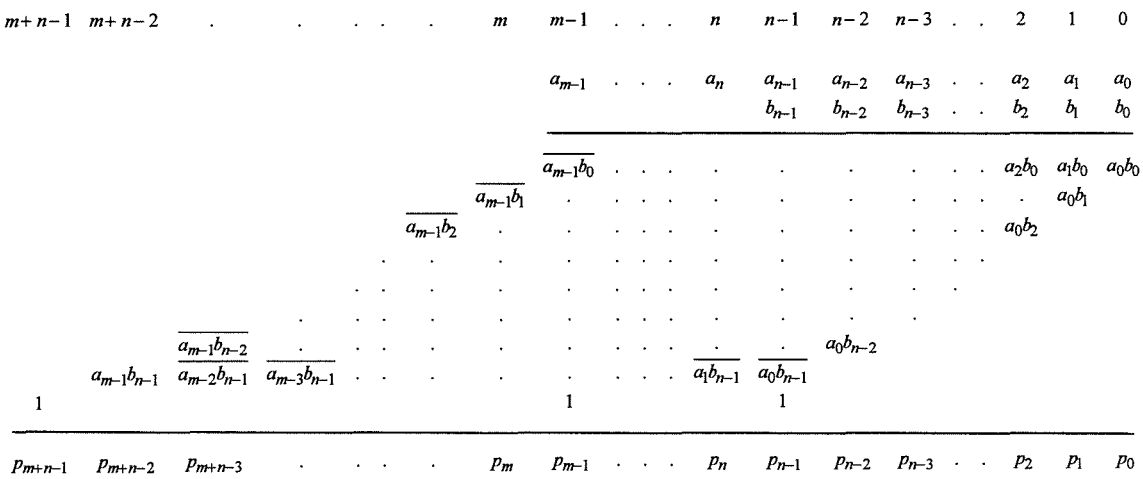


Fig. 5.10.1. PP matrix of equation (5.10.6).

If the two operands are equal, i.e. $m = n$, then equation (5.10.6) can be reduced to the form in equation (5.10.7)

$$P = a_{n-1}b_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} + 2^{n-1}\sum_{j=0}^{n-2}\overline{a_{n-1}b_j}2^j + 2^{n-1}\sum_{i=0}^{n-2}\overline{a_i b_{n-1}}2^i$$
$$+2^{2n-1} + 2^n \qquad (5.10.7)$$

The PP matrix generated by equation (5.10.7) is shown in Fig. 5.10.2

$2n-1$	$2n-2$	.	.	. . .	.	n	$n-1$	.	. . .	2	1	0
							a_{n-1}	.	. . .	a_2	a_1	a_0
							b_{n-1}	.	. . .	b_2	b_1	b_0
						1	$\overline{a_{n-1}b_0}$	.	. . .	a_2b_0	a_1b_0	a_0b_0
						$\overline{a_{n-1}b_1}$	.	.	. . .	.	a_0b_1	
					$\overline{a_{n-1}b_2}$	.	.	.	. . .	a_0b_2		
				.	.	.	.	.	. . .			
				. .	.	.	.	.	. .			
			.	. .	.	.	.	.	.			
		$\overline{a_{n-1}b_{n-2}}$	.	. .	.	.	.	a_0b_{n-2}				
1	$a_{n-1}b_{n-1}$	$\overline{a_{n-2}b_{n-1}}$	$\overline{a_{n-3}b_{n-1}}$	. .	.	$\overline{a_1b_{n-1}}$	$\overline{a_0b_{n-1}}$					
p_{2n-1}	p_{2n-2}	p_{2n-3}	.	. . .	.	p_n	p_{n-1}	.	. . .	p_2	p_1	p_0

Fig. 5.10.2. PP matrix for equation (5.10.7).

It can be readily seen that the two constant terms 2^{2n-1} and 2^n can be placed in row $n-1$ and row 0 respectively, and no additional row is generated.

For $m \neq n$, an added complication is encountered. From Fig. 5.10.1, it can be seen that the constant terms 2^{n-1} and 2^{m-1} cannot be directly assimilated by the other rows. The 2^{n-1} term can be easily handled by rewriting $2^{n-1} = 2^{n-2} + 2 \times 2^{n-3}$ which can be placed in row $n-2$ and row $n-3$. However, 2^{m-1} cannot be dealt with in the same way.

In order to eliminate the additional row, 2^{m-1} and $\overline{a_{m-1}b_0}$ are recoded as follows:

Assume two temporary variables x_1 and x_0, an equation can be written as

$$2^{m-1} + 2^{m-1}\overline{a_{m-1}b_0} = 2^m x_1 + 2^{m-1} x_0 \tag{5.10.8}$$

Table 5.10.1 lists all possible values for equation (5.10.8)

Table 5.10.1. All possible values for equation (5.10.8).

2^{m-1}	$\overline{a_{m-1}b_0}$	x_1	x_0
1	0	0	1
1	1	1	0

From Table 5.10.1, it can be seen that $x_1 = \overline{a_{m-1}b_0}$ and $x_0 = a_{m-1}b_0$. The additional row in Fig. 5.10.1 has been eliminated and the PP matrix for $m \neq n$ can be expressed in Fig. 5.10.3

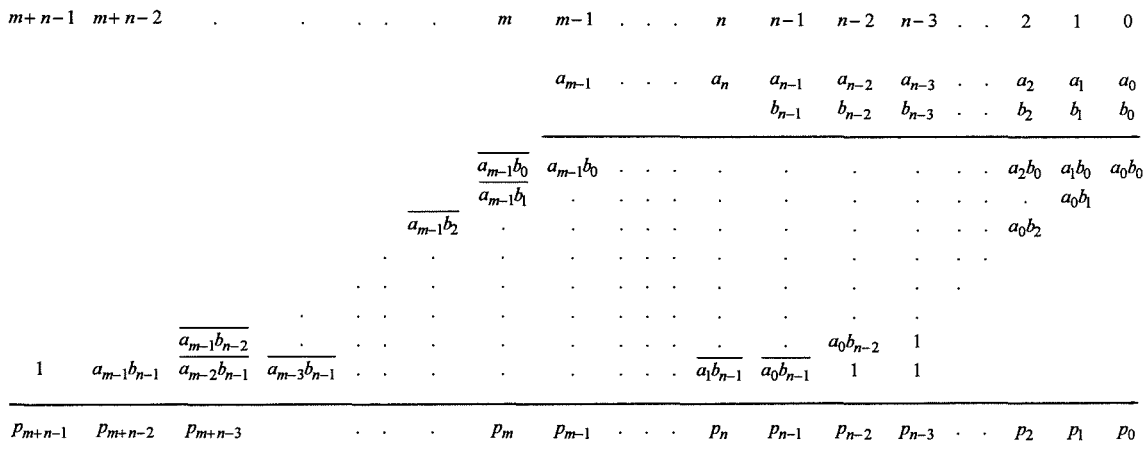


Fig. 5.10.3. PP matrix for a general form with $m \neq n$.

Implementing this algorithm is simple, it only requires an array of AND and NAND gates. Clearly, it has the same implementation complexity as the traditional Baugh and Wooley algorithm, and it generates fewer rows for a matrix of PPs.

5.11 Parallel Multiplier based on 5:3 Counter

Our previous design is a 8×8 parallel multiplier based on the 5:3 counter. The schematics and simulations are shown in Appendix A. In this design, the variant of the Baugh and Wooley algorithm is employed to generate the PP matrix, then the 5:3 counter, as the main building block, is used to construct the CFA. Finally, the CPA is realized using a simple carry skip adder based on the CMOS carry chain. The circuit is simulated and considered better than many previous designs[Guan 94B].

There is one problem in the previous design, that is, the two outputs of the basic cell (Fig. 5.7.3), i.e. C_{out1} and C_{out2}, are not buffered. For a parallel multiplier with larger operands, the performance will degrade. The reason for this is that a signal can only pass two transmission gates at each level adder. For example, a signal from input b or

d via a transmission gate is applied to C_{in} of a neighbor cell, this signal passes another transmission gate and is applied to input *b* or *d* of the next level adder. If this signal continues to pass the same path as the preceding level, then, two transmission gates will be added to a long path, this is shown in Fig. 5.11.1

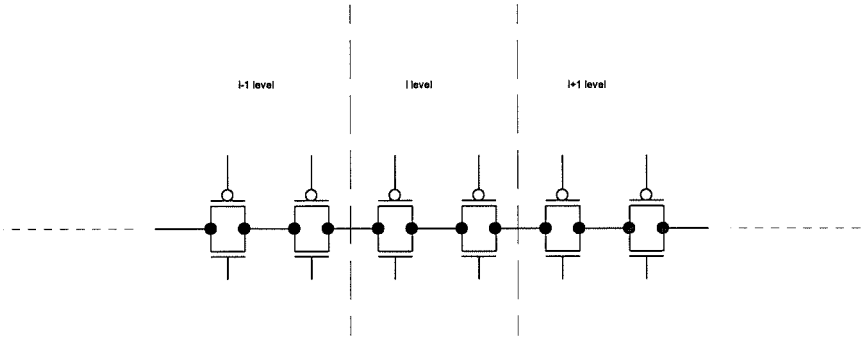


Fig. 5.11.1. A long path of transmission gates.

A multiplier with small operands, say a 8×8 multiplier, doesn't need buffers between two level adders, since only two level adders are required for the CFA, which leads to at most four transmission gates in a long path. For a multiplier with large operands, it is recommended that every two level adders are buffered, that is, the longest path is limited to have at most four transmission gates.

In practice, only C_{out2} needs to be buffered, since there is no long path in the one level adder. It is possible to add buffers between every two level adders directly. Although this approach is simple, an additional delay by adding buffers will be added for the CFA. In order to mininize the additional delay by adding buffers, INV, as a buffer, can be employed. For this purpose, another 5:3 counter is designed, which is shown in Fig. 5.11.2

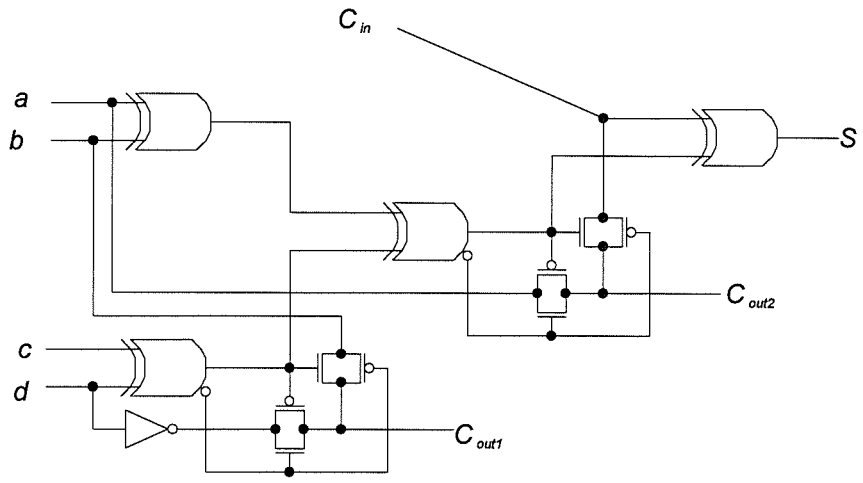


Fig. 5.11.2. Type II 5:3 counter.

This 5:3 counter in Fig. 5.11.2 is termed a type II 5:3 counter. At the same time, the previous version (in Fig. 5.7.3) is termed a type I 5:3 counter. Fig. 5.11.3 demonstrates the connections between the two types of cells.

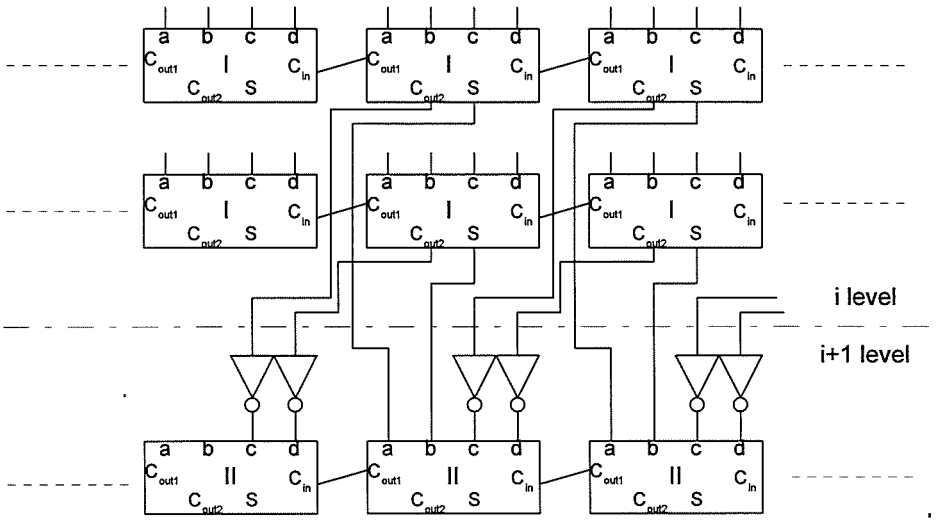


Fig. 5.11.3. Connection between type I and type II of 5:3 counter in CFA.

The idea behind the type II cell circuit is that, when inputs c and d of the type I cell are complemented, this doesn't change an XOR operation due to $\bar{c} \oplus \bar{d} = c \oplus d$. The only effect on the type I cell is that a signal for the carry generation is inverted. This is

why an INV is added between input *d* and a transmission gate, then it generates type II cell, see Fig. 5.11.2.

It can be seen that the two types of 5:3 counters are very similar, therefore, it can hardly affect the complexity of design. In addition, the INV as a buffer does not introduce an extra delay, because the delay of C_{out2} with an INV is still smaller than that of the sum output *S*.

5.12 Summary

In this chapter, parallel multipliers and their implementations are investigated. The study of the CFA is emphasized in this project since the CFA is usually implemented by employing *n:m* counters that are more concerned with the issue of logic minimization.

It can be found that for parallel multiplier design, in most situations, using RM logic or Boolean logic doesn't affect the architectural (or structural) design but only the design of basic building blocks, especially for *m:n* counters.

In this project, many different counters (including RBAs) for implementing the CFA are studied in both the RM domain and the Boolean domain, it is found that, in most cases, Boolean logic can yield simpler circuits than RM logic in terms of the number of transistors. The main reason for this may be that, as stated in Chapter 3, Boolean logic is more suitable for circuit (transistor) level design. But, it is also found that, sometimes, RM logic can solve some problems more easily than Boolean logic, the design of the 5:3 counter illustrates a good example for this.

All circuit modules discussed in this chapter are based on practical designs. The two most commonly used schemes (RBA and 5:3 counter) for the 4:2 compressor (both of them recently attracted many researchers' interests) are explored. It can be seen that these two schemes are very similar in structure. A unified structure for these two schemes is investigated. From this investigation, an important fact is obtained. That is, the CFA based on 5:3 counter and the CFA based on RBA have a similar structure, and the 5:3 counter is in fact a special form of RBA. As a result, a basic building block plays a critical role in the designs that are based on these two schemes. A survey and comparison show that the proposed circuit is better than all the existing known designs.

Two generalized algorithms for the converter from a BSD number to a binary number are presented. The difference between presented algorithms and the existing algorithms is that, the presented algorithms are format-independent and they are suitable for any formats, but nearly all the previous algorithms are format-dependent.

In this project, it is also found that a carry (borrow) propagation problem for the CPA and for the converter (from a BSD number to a binary number) is the same, therefore, nearly all speed-up schemes for the CPA can be employed by the converter.
Finally, our previous design[Guan 94B] is further improved for the case with large operands. At the same time, a variant of the Baugh and Wooley algorithm is also generalized for the case in which two operands have different lengths.

Chapter 6

Conclusion

This aims of project are to investigate arithmetic circuits, the effect of utilising RM techniques, and to develop highly efficient arithmetic circuits. In this chapter, the main findings and contributions of this project are summarized, and then, areas suitable for further research are suggested.

6.1 Summary of Results

In this project, logic circuit design based on RM logic is studied, and compared with that based on Boolean logic. The approach adopted in this project differs from previous research in two ways. Firstly, the number of transistors is employed to measure the minimum implementation for a given logic function. This leads to a more accurate evaluation for a logic function to be realized in CMOS circuitry. In most of the previous work, the number of products (or literals) is used to measure the minimized result. Secondly, all arithmetic functions are generated using the structured design approach, in this way, the circuits are easily generalized and made more suitable for practical applications. This made it possible for the actual effect of using RM logic on practical arithmetic circuits to be investigated. In nearly all previous work, benchmark arithmetic functions are used, these results can be hardly applied to a practical application, because it is difficult for them to be generalized and also it is difficult for them to be realized efficiently, especially for the circuits that have larger operands.

In this work, it is found that Boolean techniques are still more mature than RM techniques for a general logic design in MOS circuitry. The critical reason for this is that, AND and OR, the two basic operations in Boolean logic, can be simply explained by serial operation and parallel operation, the two basic principles in circuits, respectively. This makes a logic function in the Boolean domain easily mapped to a MOS circuit at the circuit (transistor) level, such circuit may be further simplified if *switching network theory* is employed in the design, and also, this circuit may be optimized at the layout level. In contrast, a logic function in the RM domain can only be mapped to a circuit at the gate level, and this result is hardly optimized further in lower levels, transistor level and layout level. This is because an XOR operation can not be explained by using a simple serial or parallel operation. This is why in nearly all circuit technologies, an XOR gate is always more complex than an AND gate or OR gate.

Although Boolean logic is more mature than RM logic, there still exist many logic functions whose circuits are not easily minimized by using Boolean logic, and it is found that these functions are easily minimized and implemented better in the RM domain. Especially, when the issue of testing a circuit is not considered at the gate level, most RM functions are actually realized more economically. 100 randomly generated four variable functions were tested, and the result shows that 33 of 100 functions realized in the RM domain are more economical than that in the Boolean domain. This demonstrates that Boolean logic is not dominant in logic design. That is, RM logic is also a useful tool for logic design, it can easily resolve many problems that Boolean can't. RM logic has another advantage of easily changing the electrical polarity in logic design.

Because in RM logic, the testability is considered to be one of the most important advantages by many researchers, it is briefly reviewed and discussed in this thesis. According to the existing methods about testing RM circuits[Reddy 72, Bhattacharya 85, Damarla 89, Sarabi 93], only two level RM functions with fixed polarity are realized and the resulting circuits are easy to test. This means that in fact, RM functions may be strictly defined as Exclusive sum of products in fixed polarity form when the testability is considered, and they are realized by using only three kinds of gates, INV, AND, and XOR. In the thesis, it is shown that if using more economical NAND gates to replace AND gates, the result does not change the property of easily testing the circuit.

In this project, the existing RM circuit testing methods are believed to be not very suitable for arithmetic circuit test. In arithmetic circuit test, the cell fault model[Parthasarathy 81, Wu 90] is accepted more widely than the gate fault model that is used in RM logic circuit test. The main reason for this is the difficulty for the

gate fault model to be used for testing a large circuit. In addition, if an arithmetic circuit is realized in Exclusive sum of products with fixed polarity form, its speed may be too slow to satisfy a general application, and also, this circuit can hardly be generalized. Therefore, a loose definition for RM logic (functions) is used in this thesis. That is, Exclusive sum of products with mixed polarity form and the mixed representation are still called RM functions in this work.

Various arithmetic circuits are studied in this work. In the thesis, only adders and multipliers are discussed, the reason for this is that, the adders and multipliers are basic circuits, and most of complex arithmetic operations can be ultimately reduced to addition and multiplication. Using Boolean logic or RM logic, in general, doesn't influence the architecture (structure) in arithmetic design.

The carry lookahead scheme, as a typical example, is explored in both the Boolean domain and the RM domain. The result shows that Boolean logic is better than RM logic for a recursive function to be implemented. This is because the circuit based on Boolean logic can share transistors and the circuit based on RM logic can only share gates.

A CMOS carry chain adder is presented. This adder is simple and fast, and was found to be better than many existing designs[Guan 93]. The structure of the proposed adder can be modified to be suitable for RNS adders. Because RNS adders with smaller operands are often used, in this situation, the carry chain circuit proposed is preferable to many other speed-up circuits. An algorithm for the on-line adder and its implementation are also proposed. Although all circuits proposed are not derived from RM logic (neither from Boolean logic), it is found that a FA in the proposed design can be explained well by using RM logic[Guan 95], which leads to generating a highly efficient 5:3 counter. It can be seen that the logic representations for the proposed 5:3 counter may not be very compact, but the resulting circuit is simpler and faster than all the existing circuits known. The reason for this is that the 5:3 counter is a multi-output function, therefore, the resulting circuit depends not only on a representation for each output, but also on the common factors shared by all outputs. This example demonstrates a potential advantage of RM logic.

The study of parallel multipliers is emphasized, because it is more closely related to logic level design than other arithmetic circuits. It is found that various parallel multipliers, in general, can be decomposed into three functional parts, PPG, CFA and CPA, and each part can be studied separately. Of these three parts, CFA is believed to have the most potential for improvement by using RM logic, particularly for various *n:m* counters that are used to realize CFA. In fact, the author's experience shows that, few circuits, which are simply designed by using a logic design means (Boolean logic or RM logic), are superior to the existing circuits[Guan 95].

4:2 compressor is concluded to be very suitable for the CFA implementation. There exist two of the most commonly used schemes to realize 4:2 compressor, one based on RBA and the other based on 5:3 counter. It is found that RBA and 5:3 counter have a unified structure, and also the structure of a parallel multiplier based on RBAs is similar to that based on 5:3 counter. Therefore, a basic building cell (RBA or 5:3 counter) plays a critical role in a parallel multiplier. A survey of parallel multipliers shows that the proposed 5:3 counter is better than all the previous circuits in terms of the number of transistors and the speed. Further, two algorithms for the conversion from a BSD number to a binary number are proposed. These two algorithms do not rely on any special format, which is different from the existing algorithms. A variant of the Baugh and Wooley algorithm, which has appeared in many other researchers' work, is generalized. One of our previous designs, a parallel multiplier[Guan 94B] is further improved to be suitable for larger operands.

Finally, it should be mentioned that even though Boolean techniques are more mature in lower level design, transistor level or layout level, than RM techniques, they still cannot describe the behavior of some structures in a satisfactory way. These structures include dynamic logic, pass transistor networks, etc.[Bryant 84], which are often used not only in arithmetic circuits, but also in other logic circuits. Logic minimization is just one way to improve logic circuits. According to the existing methods and the results of this project, it seems too early to say RM logic is better than Boolean in arithmetic circuit design, or vice versa. For optimum design, a hybrid approach utilising the advantages of both paradigms may be appropriate. For this to be practical, ECAD tools and techniques must be upgraded to make use of recent advances in logic synthesis and optimization techniques.

6.2 Future Work

•In practical applications, the XOR gate with multi-inputs is a critical component for RM logic to be realized, therefore, the improvement of this circuit directly affects applications of RM logic. This is also involved with designing a regular XOR plane and its efficient layout for XPLA implementation. In practice, it is a bottleneck problem for RM logic applications.

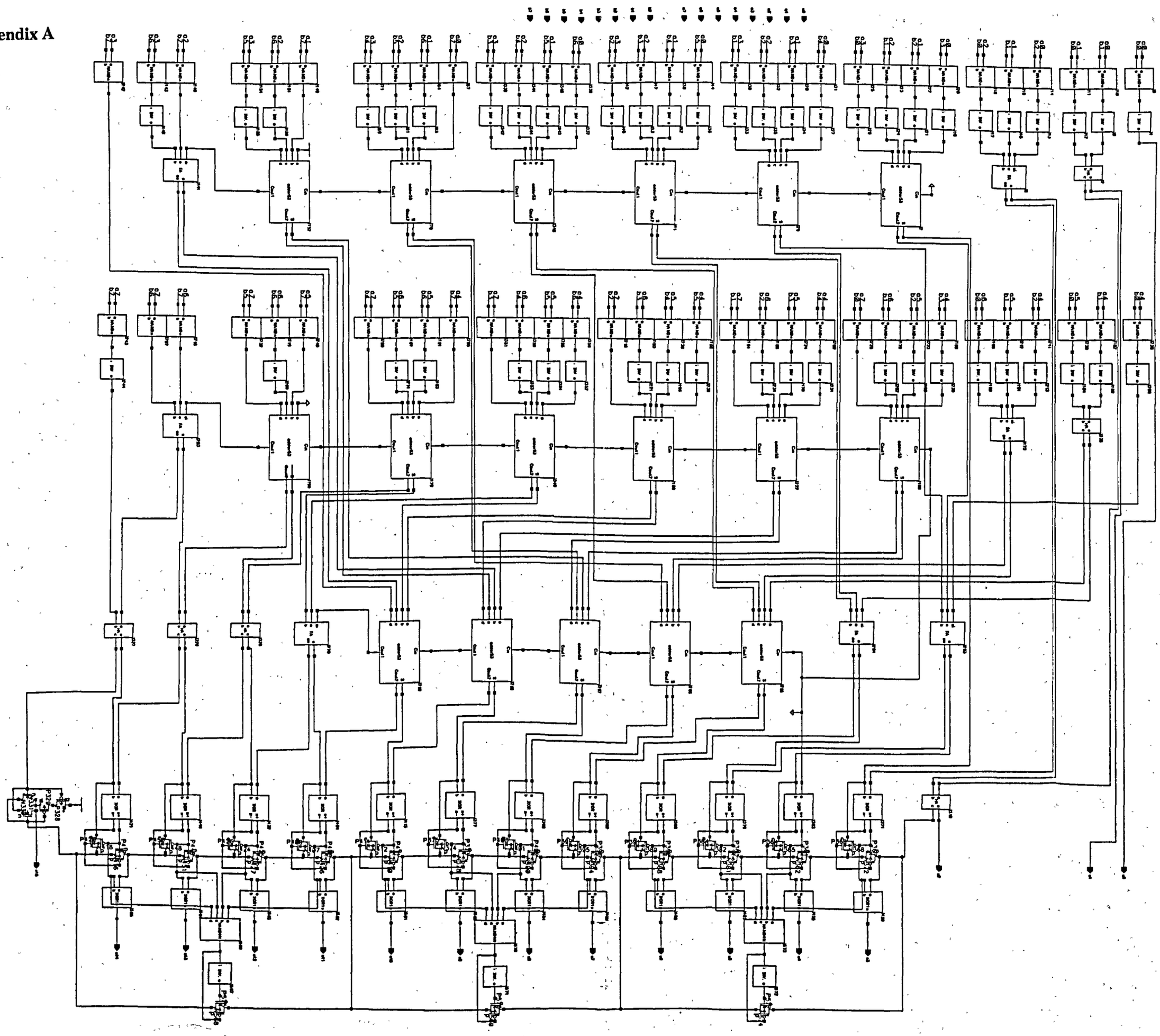
•Systematic and efficient algorithms for combining both Boolean logic and RM logic to minimize logic functions are desirable, because the results are believed to be better than that based only on Boolean logic or only on RM logic.

•From the design of the proposed 5:3 counter, it is conjectured that it is still possible for RM logic to yield highly efficient circuits of *n:m* counters with large input variables. The main advantage of the counters with large input variables is that, when these counters are used to construct a CFA, it requires less global connections in layout, therefore, the resulting circuit may be improved.

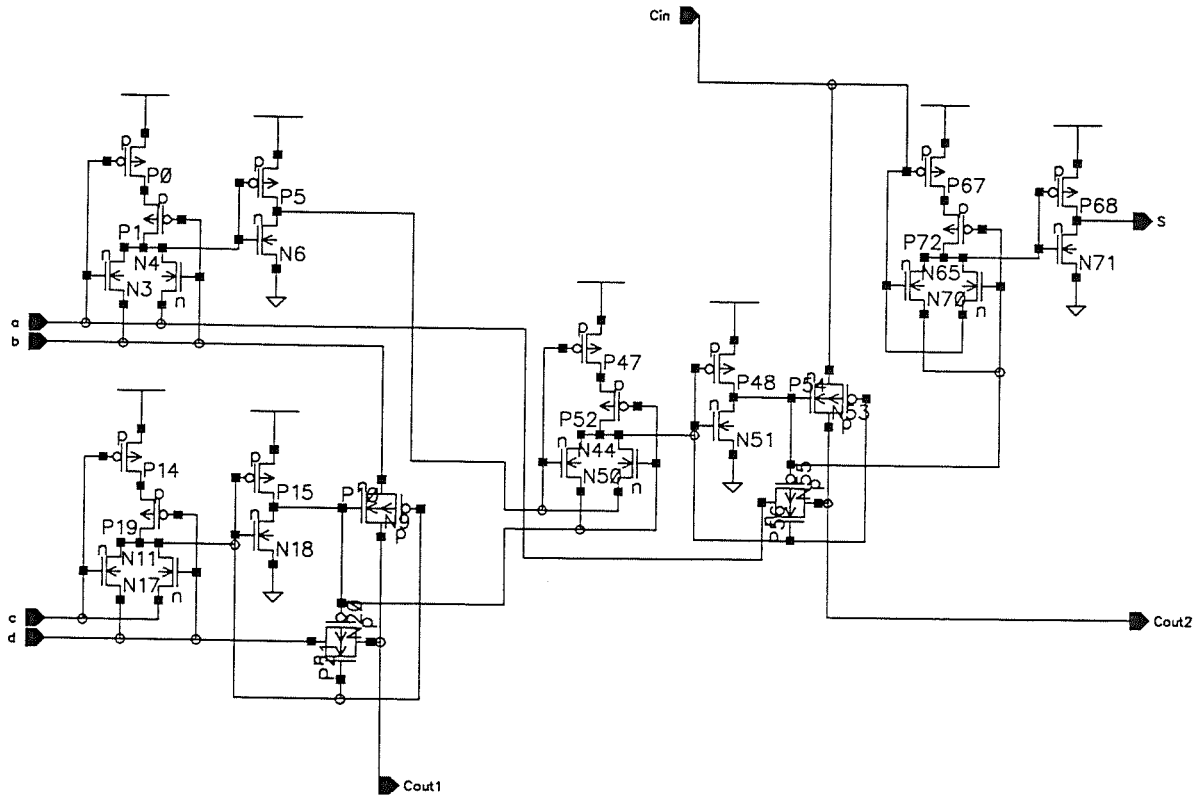
•The existing test methods for RM logic are only limited to the circuits based on two level representation and fixed polarity. In general, these circuits with good testability are still too costly and slow to be accepted for many practical applications. The circuits based on multi-level representation and mixed polarity are simpler and faster, test methods for these circuits are desirable.

•Although 4:2 compressor scheme for CFA has been investigated by many researchers, it is hard to find an efficient testing approach for it in the literature. Therefore, testing CFA based on 4:2 compressor should be explored.

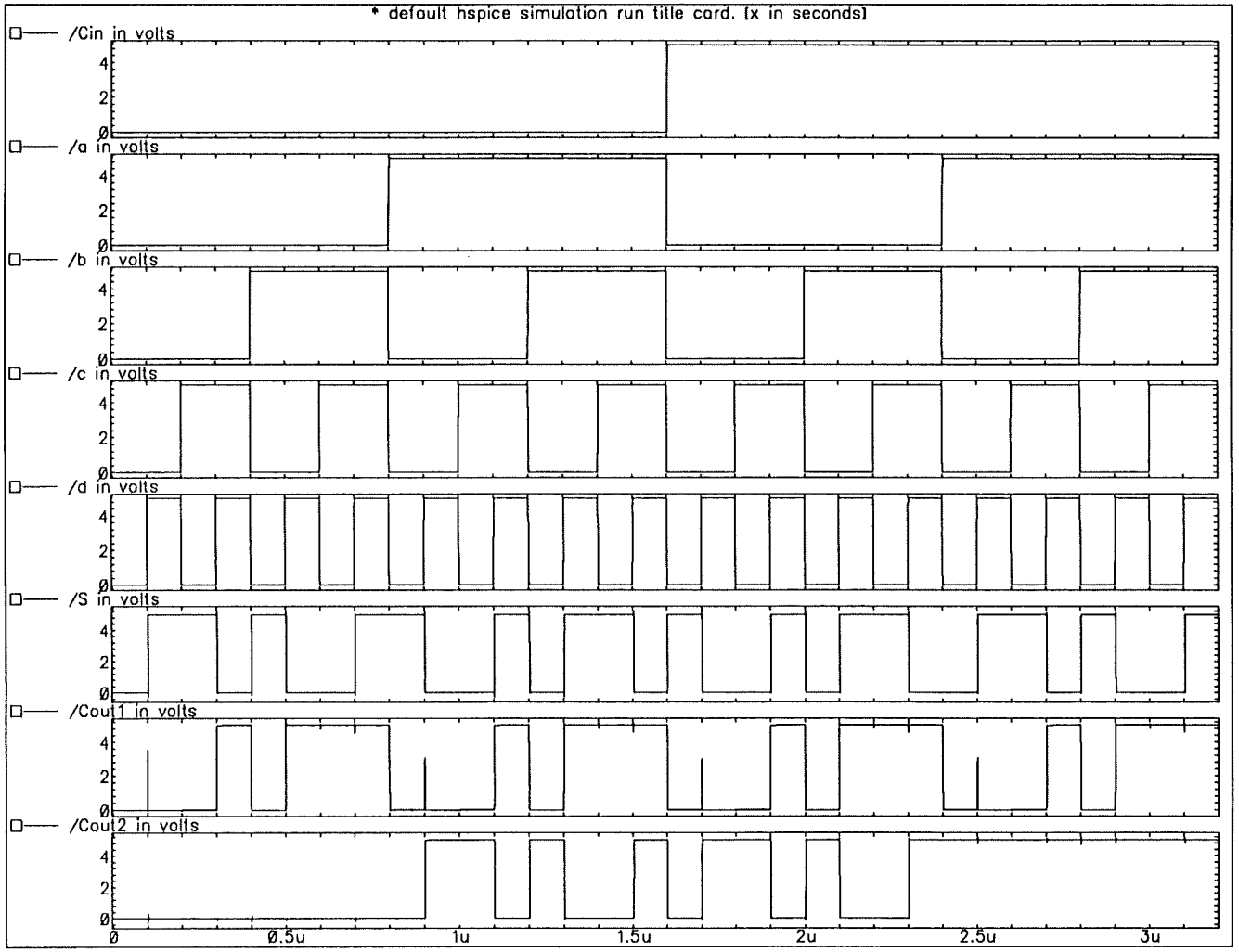
Appendix A



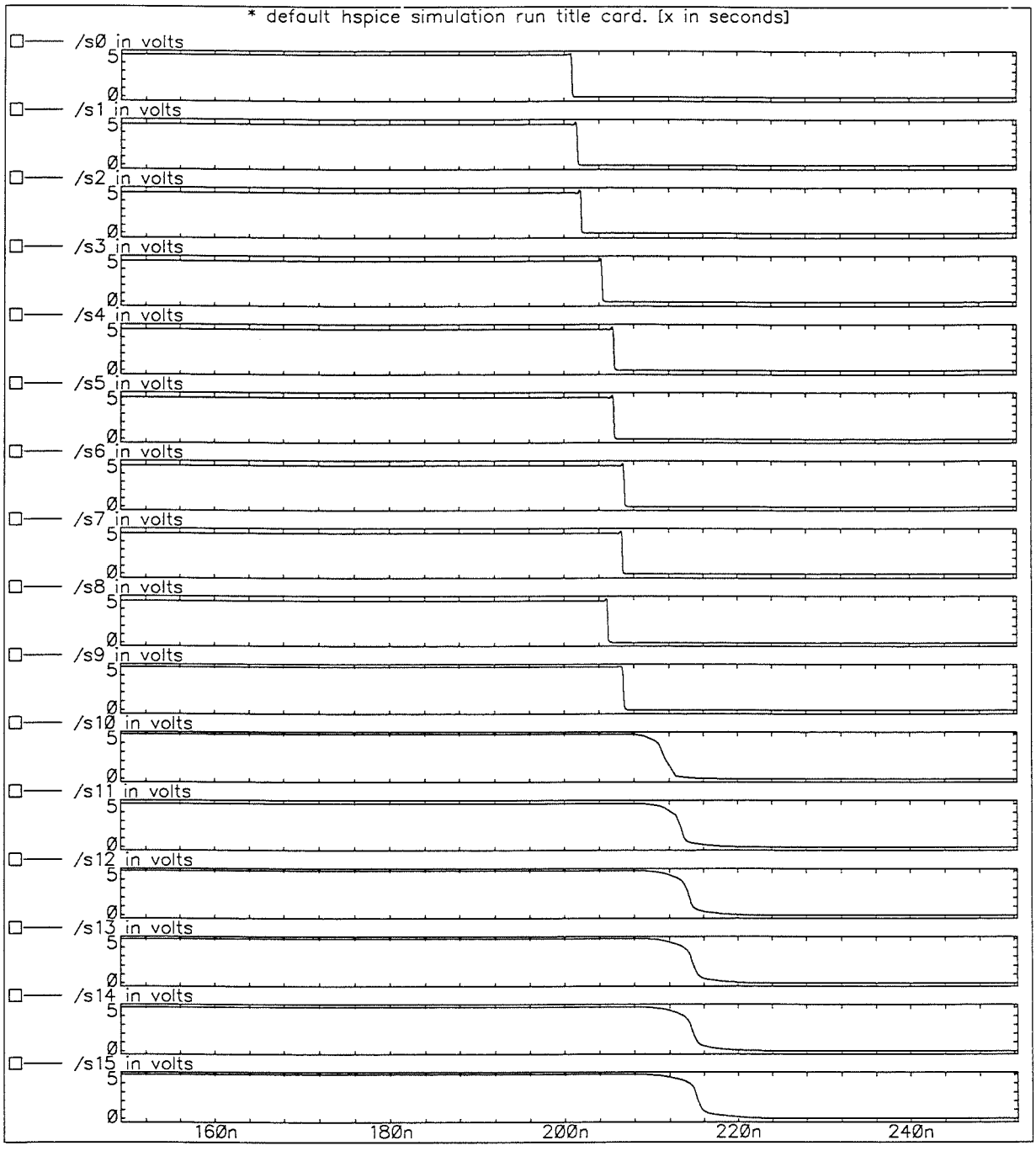
(1). The schematic for a 8×8 parallel multiplier based on 5:3 counter.



(2). The circuit of 5:3 counter.



(3). The simulation of the circuit of 5:3 counter
(the simulator is Hspice in Cadence suite).



(4). The waveform for the worst delay of the multiplier, it is less than 17ns (the test pattern is 00000001×11111111).

References

[Ait-Boudaoud 91] D. Ait-Boudaoud, M.K. Ibrahim and B.R. Hayes-Gill,"Novel cell architecture for bit level systolic arrays multiplication" *IEE Proc. E.*, Vol. 138, No. 1, Jan. 1991, pp. 21-26.

[Almaini 91] A.E.A. Almaini, P. Thomson and D. Hanson,"Tabular Techniques for Reed-Muller Logic", *Int. J. Electronics,* Vol. 70, No. 1, 1991, pp. 23-34.

[Almaini 94] A.E.A. Almaini, *Electronic Logic Systems*, 3rd Edition, Prentice Hall, 1994.

[Avizienis 61]A. Avizienis,"Signed-Digit Number Representations for Fast Parallel Arithmetic", *IRE Trans. Electronic Computers*, Vol. EC-10, 1961, pp. 389-400.

[Balakrishnan 92] W. Balakrishnan and N. Burgess,"Very-high-speed VLSI 2s-complement multiplier using signed binary digits", *IEE Proc. E*, Vol. 139, No. 1, Jan. 1992, pp. 29-34.

[Baugh 73] C.R. Baugh and B.A. Wooley,"A Two's Complement Parallel Array Multiplication Algorithm", *IEEE Trans. Computers*, Vol. C-22, No. 12, Dec. 1973, pp. 1045-1047.

[Besslich 83] Ph.W. Besslich,"Efficient Computer Method for EXOR Logic Design", *IEE Proc. Pt. E*, Vol. 130, No. 6, Nov. 1983, pp. 203-206.

[Bhatia 91] S. Bhatia and A. Albicki,"Testing of Iterative Logic Arrays", *IEEE Proc. 33rd Midwest Symp. on Circuits and Systems*, 1991, pp. 243-246.

[Bhattacharya 85] B.B. Bhattacharya, B. Gupta, S. Sarkar and A.K. Choudhury,"Testable design of RMC networks with universal tests for detecting stuck-at and bridging faults", *IEE Proc. Pt. E*, Vol. 132, No. 3, May 1985, pp. 155-162.

[Booth 51] A.D. Booth,"A Signed Multiplication Technique", Quarterly J. Mech. & Appl. Math., Vol. 4, Pt. 2, 1951, pp. 236-240.

[Brent 82] R.P. Brent and H.T. Kung,"A Regular Layout for Parallel Adders", *IEEE Trans. Computers*, Vol. C-31, No. 3, March, 1982, pp. 260-264.

[Bryant 84] R.E. Bryant,"A Switch-Level Model and Simulator for MOS Digital Systems", *IEEE Trans. Computers*, Vol. c-33, No. 2, Feb. 1984, pp. 160-177.

[Carlson 93] B.S. Carlson and C.Y.R. Chen,"Performance Enhancement of CMOS VLSI Circuits by Transistor Reordering", *30th ACM/IEEE Design Automation Conf.*, 1993, pp. 391-366.

[Cavanagh 84] J.J.F. Cavanagh, *Digital Computer Arithmetic Design and Implementation*, McGraw-Hill Book Company, 1984.

[Chan 90] P.K. Chan and M.D.F. Schlag,"Analysis and Design of CMOS Manchester Adders with Variable Carry-Skip", *IEEE Trans. Computers*, Vol. 39, No. 8, Aug. 1990, pp. 983-992.

[Chan 92] P.K. Chan, M.D.F. Schlag, C.D. Thomborson and V.G. Oklobdzija,"Delay Optimization of Carry-Skip Adders and Block Carry-Lookahead Adders Using Multidimensional Dynamic Programming", *IEEE Trans. Computers*, Vol. 41, No. 8, Aug. 1992, pp. 920-930.

[Chatterjee 87] A. Chatterjee and J.A. Abraham,"Test Generation for Arithmetic Units by Graph Labelling", *IEEE Int. Symp. Fault-Tolerant Comput.*, Pittsburgh, PA, July 1987, pp. 284-289.

[Chen 87] J.S.J. Chen and D.Y. Chen,"A Design Rule Independent Cell Compiler", *Proc. 24th ACM/IEEE Design Automation Conf.*, 1987, pp. 466-471.

[Cheng 87] W.T. Cheng and J.H. Patel,"A Minimum Test Set for Multiple Fault Detection in Ripple Carry Adders", *IEEE Trans. Computers*, Vol. C-36, No. 7, July 1987, pp. 891-895.

[Chow 78] C.Y. Chow and J.E. Robertson,"Logic Design of a Redundant Binary Adder", *IEEE Proc. 4th Symp. Comput. Arithmetic*, Oct. 1978, pp. 109-115.

[Csanky 93] L. Csanky, M.A. Perkowski and I. Schäfer,"Canonical Restricted Mixed-Polarity Exclusive-OR Sums of Products and the Efficient Algorithm for their Minimization", *IEE Proc.-E*, Vol. 140, No. 1, Jan. 1993, pp. 69-77.

[Dadda 65] L. Dadda,"Some Schemes for Parallel Multipliers", *ALTA Frequenza*, May 1965, pp. 349-356.

[Dagenais 91] M. Dagenais,"Efficient Algorithmic Decomposition of Transistor Groups into Series, Bridge, and Parallel Combinations", *IEEE Trans. Circuits and Systems*, Vol. 38, No. 6, June 1991, pp. 569-581.

[Damarla 89] T. Damarla and M. Karpovsky,"Detection of stuck-at and bridging faults in Reed-Muller canonnical(RMC) networks", *IEE Proc. Pt. E*, Vol. 136, No. 5, Sept. 1989, pp. 430-433.

[Davio 78] M. Davio, J.-P. Daschamps and A. Thayse, *Discrete and Switching Functions*, McGraw-Hill International, 1978.

[Detjens 87] E. Detjens, G. Gannot, R. Rudell, A.S. Vincentelli and A. Wang,"Technology Mapping in MIS", *IEEE Proc. Int. Conf. on Computer Aided Design*. 1987, pp. 116-119.

[Detjens 90] E. Detjens,"FPGA Devices Require FPGA-Specific Synthesis Tools", *Computer Design*, Nov. 1, 1990, pp. 124.

[Doran] R.W. Doran,"Variants of an Improved Carry Look-Ahead Adder", *IEEE Trans. Computers*, Vol. 37, No. 9, Sept. 1988, pp. 1110-1113.

[Dubrova 95] E.V. Dubrova, D.M. Miller and J.C. Muzio,"Upper bound on number of products in AND-OR-XOR expansion of logic functions", *Electronics Letters*, Vol. 31, No. 7, 30th March 1995, pp. 541-542.

[Dugdale 92] M. Dugdale,"VLSI Implementation of Residue Adders based on Binary Adders", *IEEE Trans. Circuits and Systems-II: Analog and Digital signal processing*, Vol. 39, No. 5, May 1992, pp. 325-329.

[Ercegovac 80] M.D. Ercegovac and A.L. Grnarov,"On the Performance of On-line Arithmetic", *IEEE Proc. Int. Conf. on Parallel Processing*, Aug. 1980, pp. 55-62.

[Ercegovac 87] M.D. Ercegovac and T. Land,"On-the-fly Conversion of Redundant into Conventional Representations", *IEEE Trans. Computers*, Vol. C-36, No. 7, July 1987, pp.895-897.

[Ercegovac 90] M.D. Ercegovac and T. Land,"Fast Multiplication without Carry-Propagate Additions", *IEEE Trans. Computers*, Vol. 39, No. 11, Nov. 1990, pp. 1385-1390.

[Geiger 90] R.L. Geiger, P.E. Allen and N.R. Strader, *VLSI Design Techniques for Analog and Digital Circuits*, McGraw-Hill Publishing Company, 1990.

[Gnanasekaran 85] R. Gnanasekaran 85,"A Fast Serial-Parallel Binary Multiplier", *IEEE Trans. Computers*, Vol. c-34, No. 8, Aug. 1985, pp. 741-744.

[Goto 92] G. Goto, T. Sato, M. Nakajima and T. Sukemura,"A 54×54-b Regularly Structure Tree Multiplier", *IEEE J. Solid-State Circuits*, Vol. 27, No. 9, Sept. 1992, pp. 1229-1236.

[Green 86] D.H. Green, *Modern Logic Design*, Addison-Wesley, 1986.

[Green 90] D.H. Green,"Reed-Muller canonical forms with mixed polarity and their manipulations", *IEE Proc. Pt. E*, Vol. 137, No. 1, Jan. 1990, pp. 103-113.

[Green 91A] D.H. Green,"Families of Reed-Muller Canonical Forms", *Int. J. Electronics*, Vol. 70, No. 2, 1991, pp. 259-280.

[Green 91B] D. H. Green,"Reed-Muller Algebraic Techniques", *Digital Systems Reference Book*, edited by B. Holdsworth and G.R. Martin, Butterworth-Heinemann Ltd, Oxford, 1991, **3.3**/2-**3.3**/15.

[Green 93] D.H. Green, and G.A. Khuwaja,"Tabular simplification method for switching functions expressed in Reed-Muller algebraic form", *Int. J. Electronics*, Vol. 75, No. 2, 1993, pp. 297-314.

[Green 94] D.H. Green,"Dual Forms of Reed-Muller Expansions", *IEE Proc. Comput. Digit. Tech.*, Vol. 141, No. 3, May 1994, pp. 184-192.

[Guan 93] Z. Guan, A.E.A. Almaini and P. Thomson,"A simple and high speed CMOS carry chain adder architecture", *Int. J. Electronics*, Vol. 75, No. 4, 1993, pp. 742-752.

[Guan 94A] Z. Guan,"Logic Realization Using Mixed Representations based on Reed-Muller forms", *IEE Colloquium on Synthesis and Optimisation of Logic Systems*, London, 14, March, 1994, pp. 2/1-2/4.

[Guan 94B] Z. Guan, P. Thomson and A.E.A. Almaini,"A Parallel CMOS 2's Complement Multiplier Based on 5:3 Counter", *Proc. IEEE Int. Conf. Comput. Design*, Boston, USA, 10-12, Oct. 1994, PP. 298-301.

[Guan 95] Z. Guan and A.E.A. Almaini,"One Bit Adder Design Based on Reed-Muller Expansions", *Int. J. Electronics*, to be published.

[Guyot 87] A. Guyot, B. Hochet and J.M. Muller,"A Way to Build Efficient Carry-Skip Adders", *IEEE Trans. Computers*, Vol. C-36, No. 10, Oct. 1987, pp. 1144-1152.

[Harata 87] Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa and N. Nakagi," A High-Speed Multiplier Using a Redundant Binary Adder Tree", *IEEE J. of Solid-States Circuits*, Vol. SC-22, No. 1, Feb. 1987, pp. 28-34.

[Harking 90] B. Harking,"Efficient algorithm for canonical Reed-Muller expansions of Boolean functions", *IEE Proc. E.*, No. 5, Sept. 1990, pp. 366-370.

[Helliwell 88] M. Helliwell and M. Perkowski,"A Fast Algorithm to Minimize Multi-Output Mixed-Polarity Generalized Reed-Muller Forms", *25th ACM/IEEE Design Automation Conference*, 1988, pp. 427-432.

[Huang 94] X. Huang, W. Liu and B.W.Y. Wei,"A High-Performance CMOS Redundant Binary Multiplication-and-Accumulation(MAC) Unit", *IEEE Trans. Circuit and Systems-I: Fundamental Theory and Applications*, Vol. 41, No. 1, Jan. 1994, pp. 33-39.

[Hurst 92] S.L. Hurst, *Custom VLSI Microelectronics*, Prentice Hall, UK, 1992

[Hwang 79] K. Hwang, *Computer Arithmetic Principles, Architecture and Design*, Wiley, New York, 1979.

[Joseph 84] J.F.C. Joseph, *Digital Computer Arithmetic Design and Implementation*, McGraw-Hill, USA, 1984.

[Kantabutra 93] V. Kantabutra,"Designing Optimum One-Level Carry-Skip Adders", *IEEE Trans. Computers*, Vol. 42, No. 6, June 1993, pp. 759-764.

[Kilburn 60] T. Kilburn, D.B.G. Edwards and D. Aspinall,"A Parallel Arithmetic Unit Using a Satuated-Transistor Fats-Carry Circuit", *Proc. IEE Pt. B*, Vol. 107, Nov. 1960, pp. 573-584.

[Kohavi 78] Z. Kohavi, *Switching and Finite Automata Theory*, Second Edition, McGraw-Hill Book Company, 1978.

[Koren 93] I. Koren, *Computer Arithmetic Algorithms*, Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[Kornerup 94] P. Kornerup,"Digit-Set Conversion Generalizations and Applications", *IEEE Trans. Computers*, Vol. 43, No. 5, May, 1994, pp. 622-629.

[Kuninobu 87] M. Kuninobu T. Nishiyama, H. Edamatsu, T. Taniguchi and N. Takagi,"Design of High Speed MOS Multiplier and Divider Using Redundant Binary Representation", *Proc. IEEE 8th Symp. Comput. Arithmetic*, pp.80-86.

[Lala 85] P.K. Lala, *Fault Tolerant & Fault Testable Hardware Design*, Prentice-Hall International, Inc., 1985.

[Lee 93] Y.-T. Lee, I.-C. Park and C.-M. Kyung,"Design of Compact Static CMOS Carry Look-Ahead Adder using Recursive Output Property", *Electronics Letters*, Vol. 29, No. 9, 29th Apr. 1993, pp. 794-796.

[Lester 93] N.L.K. Lester and J.M. Saul,"Technology Mapping of Mixed Polarity Reed-Muller Representations", *IEEE Proc. of the European Conf. on Design Automation*, Feb. 1993, pp.305-309.

[Lin 93] T. Lin and A. Tran,"Minimization of multiple-output exclusive-OR switching functions", *Int. J. Electronics*, Vol. 75, No. 4, 1993, pp. 665-674.

[Ling 81] H. Ling,"High-Speed Binary Adder", *IBM R.&D.*, Vol. 25, No. 3, May 1981, pp. 156-166.

[Lui 92] P.K. Lui and J.C. Muzio,"Boolean Matrix Transforms for the Minimization of Modulo-2 Canonical Expansions", *IEEE Trans. Computers*, Vol. 41, No. 3, March 1992, pp. 342-347.

[MacSorley 61] O.L. MacSorley,"High-Speed Arithmetic in Binary Computers", *Proc. IRE*, Vol. 99, Jan. 1961, pp. 67-91.

[Makino 93] H. Makino, Y. Nakase and H. Shinohara,"A 8.8-ns 54×54-bit Multiplier Using New Redundant Binary Architecture", *Proc. IEEE Int. Conf. Comput. Design*, pp. 202-205.

[Maziasz 87] R.L. Maziasz and J.P. Hayes,"Layout Optimization of CMOS Functional Cells", *Proc. 24th ACM/IEEE Design Automation Conf.*, 1987, pp. 544-551.

[Mead 80] C.A. Mead and L.A. Conway, *Introduction to VLSI Systems*, Additon Wesley, 1980.

[McAuley 92] A.J. McAuley,"Dynamic Asynchronous Logic for High-Speed CMOS Systems", IEEE J. of Solid-State Circuits, Vol. 27, No. 3, March 1992, pp. 382 388.

[McKenzie 93] L. McKenzie, A.E.A. Almaini, J.F. Miller and P. Thomson,"Optimization of Reed-Muller logic functions", *Int. J. Electronics*, Vol. 75, No. 3, 1993, pp. 451-466.

[Mehta 91] M. Mehta, V. Parmar and E. Swartzlander,"High-Speed Multiplier Design Using Multi-Input Counter and Compressor Circuits", *Proc. 10th Symp. Compt. Arithmetic*, pp. 43-50.

[Mekhallalati 92] M.C. Mekhallalati and M.K. Ibrahim,"New Parallel Multiplier Design", *Electronics Letters*, Vol. 28, No. 17, Aug. 1992, pp. 1650-1651.

[Moh 95] S.M. Moh and S.H. Yoon,"Serial-parallel multiplier for two's complement numbers", *Electronics Letters*, Vol. 31, No. 9, 27th April 1995, pp. 703-704.

[Muller 54]D.E. Muller, "Application of Boolean Algebra to Switching Circuit Design and to Error Detection", *IRE Trans. Electronic Computers*, Sept. 1954, pp. 6-12.

[Nagamatsu 90] M. Nagamatsu, S. Tanaka, J. Mori, K. Hirano, T. Noguchi and K. Hatanaka,"A 15-ns 32×32-b CMOS Multiplier with an Improved Parallel Structure", *IEEE J. of Solid-State Circuits*, Vol. 25, No. 2, April 1990, pp. 494-497.

[Parhi 90] K.K. Parhi and C.Y. Wang,"Digit-Serial DSP Architectures", *Proc. IEEE Conf. Application Specific Array Processors*, 1990, pp. 341-351.

[Parthasarathy 81] R. Parthasarathy and S.M. Reddy,"A Testable Design of Iterative Logic Arrays", *IEEE Trans. Computers*, Vol. c-30, No. 11, Nov. 1981, pp 833-841.

[Perkowski 89] M. Perkowski, M. Helliwell and P. Wu,"Minimization of Multiple-Valued Input Multi-Output Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions", *IEEE Proc. 19th Int. Symp. on Multiple-Valued Logic*, 1989, pp. 256-263.

[Perkowski 90] M. Perkowski and M.C. Jeske,"An Exact Algorithm to Minimize Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions", *Proc. IEEE Int. Conf. Comput. Design*, 1990, pp.1652-1655.

[Phatak 94] D.S. Phatak and I. Koren,"Hybrid Sighed-Digit Number Systems: A Unified Framework for Redundant Number Representations With Bounded Carry Propagation Chains", *IEEE Trans. Computers*, Vol. 43, No. 8, Aug. 1994, pp. 880-891.

[Pitty 88] E.B. Pitty,"A Critique of the GATEMAP Logic Synthesis System", *Proc. Int. Workshop on Logic and Architecture Synthesis for Silicon Compilers*, Grenoble, France, May, 1988, pp. 65-84.

[Pucknell 88] D.A. Pucknell, and K. Eshraghian, *Basic VLSI Design*, Englewood Cliffs, NJ: Prentice Hall, 1988.

[Rajashekhara]. T.N. Rajashekhara, and O. Kal,"Fast multiplier design using redundant signed-digit numbers", *Int. J. Electronics*, Vol. 69, No. 3, 1990, pp. 359-368.

[Razavi 92] H.M. Razavi and J. Battelini,"Design of a residue arithmetic multiplier", *IEE Proc. G*, Vol. 139, No. 5, Oct. 1992, 581-585.

[Reddy 72] S.M. Reddy,"Easily Testable Realization for Logic Functions", *IEEE Trans. Computers*, Vol. c-21, No. 11, Nov. 1972, pp. 1183-1188.

[Reed 54]I.S. Reed,"A Class of Multiple-Error-Correcting Codes and the Decoding Scheme", *IRE Trans. Information Theory*, Vol. PGIT-4, 1954, pp. 38-49.

[Salmon 89] J.V. Salmon, E.B. Pitty and M.S. Abrahams,"Syntactic Translation and Logic Synthesis in Gatemap", *IEE Proc. Pt. E*, Vol. 136, No. 4, July 1989, pp. 321-328.

[Sam 90] H. Sam and A. Gupta,"A Generalized Multibit Recoding of Two's Complement Binary Numbers and Its Proof with Application in Multiplier Implementations", *IEEE Trans. Computers*, Vol. 39, No. 8, Aug. 1990, pp. 1006-1015.

[Sarabi 92] A. Sarabi and M.A. Perkowski,"Fast Exact and Quasi-Minimal Minimization of Highly Testable Fixed-Polarity AND/XOR Canonical Networks", *29th ACM/IEEE Design Automation Conf*, 1992, pp. 30-35.

[Sarabi 93] A. Sarabi and M.K. Perkowski,"Design for Testabilities of AND/XOR Networks", *Proc. IFIP W.G. 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, Hamburg, Germany, Sept. 16-17, 1993, pp. 147-153.

[Sasao 90] T. Sasao and P. Besslich,"On the complexity of MOD-2 Sum PLA's", *IEEE Trans. Computers*, Vol. 32, No. 2, Feb. 1990, pp. 262-266.

[Sasao 93A] T. Sasao,"Logic Synthesis with EXOR Gates", *Logic Synthesis and Optimization"*, Editor: T. Sasao, Kluwer Academic, 1993, pp. 259-284.

[Sasao 93B] T. Sasao,"AND-EXOR Expressions and their Optimization", *Logic Synthesis and Optimization*, Editor: T. Sasao, Kluwer Academic, 1993, pp. 287-312.

[Saul 90] J.M. Saul,"An Improved Algorithm for the Minimization of Mixed Polarity Reed-Muller Representations", *Proc. IEEE Int. Conf. Comput. Design*, 1990, pp. 372-375.

[Saul 91] J. Saul,"An Algorithm for the Multi-level Minimization of Reed-Muller Representations", *Proc. IEEE Int. Conf. Comput. Design*, 1991, pp. 634-637.

[Saul 92] J. Saul,"Logic Synthesis for Arithmetic Circuits Using the Reed-Muller Representation", *EuroASIC'92*, 1992, pp. 109-113.

[Saul 93] J. Saul, B. Eschermann and J. Froessl,"Two-Level Logic Circuits using EXOR sums of Products", *IEE Proc.-E*, Vol. 140, No. 6, Nov. 1993, pp. 348-356.

[Scott 85] N.R. Scott, *Computer Number Systems and Arithmetic*, Prentice-Hall, Inc., 1985.

[Sklansky 60] J. Sklansky,"Conditional-Sum Addition Logic", *TRE Trans. Electronic Computers*, June 1960, pp.226-231.

[Srinivas 91] H.R. Srinivas and K.K. Parhi,"High-Speed VLSI Arithmetic Processor Architectures Using Hybrid Number Representation", *Proc. IEEE Int. Conf. Comput. Design*, 1991, pp. 564-571.

[Srinivas 92] H.R. Srinivas and K.K. Parhi,"A Fast VLSI Adder Architecture", *IEEE J. Solid-State Circuits*, Vol. 27, No. 5, May 1992, pp. 761-767.

[Steinbach 93] B. Steinbach and G. Kempe,"Minimization of AND-ExOR Expressions", *Proc. IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Logic Design*, Hamburg, Germany, Sept. 16-17, 1993, pp. 20-26.

[Stenzel 77] W.J. Stenzel, W.J. Kubitz and G.H. Garcia,"A Compact High-Speed Parallel Multiplication Scheme", *IEEE Trans. Computers*, Vol. c-26, No. 10, Oct. 1977, pp. 948-957.

[Swartzlander 73] E. Swartzlander,"Parallel Counters", *IEEE Trans. Computers*, Vol. C-22, No. 11, Nov. 1973, pp. 1021-1024.

[Szabo 67] N.S. Szabo and R.I. Tanaka, *Residue Arithmetic and its Applications to Computer Technology*, McGraw-Hill Book Company, 1967.

[Takagi 85] N. Takagi, H. Yasuura and S. Yajima,"High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree", *IEEE Trans. Computers*, Vol. c-34, No. 9, Sept. 1985, pp. 789-796.

[Takagi 92] N. Takagi,"A Radix-4 Modular Multiplication Hardware Algorithm for Modular Exponentiation", *IEEE Trans. Computers*, Vol. 41, No. 8, Aug. 1992, pp. 949-956.

[Tran 87] A. Tran,"Graphical Method for the Conversion of Minterms to Reed-Muller Coefficients and Minimisation of Exclusive-OR Switching Functions", *IEE Proc. Pt. E*, Vol. 134, No. 2, Mar. 1987, pp. 93-99.

[Tran 89] A Tran,"Tri-state Map for the Minimisation of Exclusive-OR Switching Functions", *IEE Proc. Pt. E*, Vol. 136, No. 1, Jan. 1989, pp. 16-21.

[Tran 93A] A. Tran and E. Lee,"Generalisation of Tri-State Map and a Composition Method for Minimisation of Reed-Muller Polynomials in Mixed Polarity", *IEE Proc.-E*, Vol. 140, No. 1, Jan. 1993, pp. 59-64.

[Tran 93B] A Tran and J. Wang,"Decomposition method for minimization of Reed-Muller Polynomials in Mixed Polarity", *IEE Proc. E.*, Vol. 140, No. 1, Jan. 1993, pp. 65-68.

[Uehara 81] T. Uehara and W.M. Vancleemput,"Optimal Layout of CMOS Functional Arrays", *IEEE Trans. Computers*, Vol. c-30, No. 5, May 1981, pp. 305-312.

[Urquhart 84] R.B. Urquhart and D. Wood,"Systolic matrix and vector multiplication methods for signal processing", *IEE Proc. Pt. F*, Vol. 131, No. 6, Oct. 1984, pp. 623-631.

[Vassiliadis 89]. S. Vassiliadis, E.M. Schwarz and D.J. Hanrahan,"A General Proof for Overlapped Multiple-Bit Scanning Multiplications", *IEEE Trans. Computers*, Vol. 38, No. 2, Feb. 1989, pp. 172-183.

[Wallace 64] C.S. Wallace,"A Suggestion for a Fast Multiplier", *IEEE Trans. Electronic Computers*, Feb. 1964, pp. 14-17.

[Weste 93] N.H.E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design, A Systems Perspective*, Second Edition, Addison-Wesley Publishing Company, 1993.

[Wu 82] X. Wu, X. Chen and S.L. Hurst,"Mapping of Reed-Muller Coefficients and the Minimization of Exclusive OR Switching Functions", *IEE Proc. Pt. E*, Vol. 129, No. 1, Jan. 1982, pp. 15-20.

[Wu 85] M. Wu, W. Shu and S. Chan,"A Unified Theory for MOS Circuit Design-Switching Network Logic", *Int. J. Electronics*, Vol. 58, No. 1, 1985, pp. 1-33.

[Wu 87] C.E. Wu, A.S. Wojcik and L.M. Ni,"A Rule-Based Circuit Representation for Automated CMOS Design and Verification", *Proc. 24th ACM/IEEE Design Automation Conf.*, 1987 pp.786-792.

[Wu 90] C.W. Wu and P.R. Cappello,"Easily Testable Iterative Logic Arrays", *IEEE Trans. Computers*, Vol. 39, No. 5, May 1990, pp. 640-652.

[Yano 90] K. Yano, T. Yamanaka, T. Nishida, M. Saito, K. Shimohigashi and A. Shimizu,"A 3.8-ns CMOS 16×16-b Multiplier Using Complementary Pass-Transistor Logic", *IEEE J. Solid-State Circuits*, Vol. 25, No. 2, Apr. 1990, pp. 388-395.

[Yen 92] S.M. Yen , C.S. Laih, C.H. Chen and J.Y. Lee,"An Efficient Redundant-Binary Number to Binary Number Converter", *IEEE J. Solid-State Circuits*, Vol. 27, No. 1, Jan. 1992, pp. 109-112.

[Zhu 93] J. Zhu and M. Abd-El-Barr,"On the optimization of MOS circuits", *IEEE Trans on Circuits and Systems-I:Fundamental Theory and Applications*, Vol. 40, No. 6, June 1993, pp. 412-422.