An Application Specific Integrated Circuit for Optimization of Fixed Polarity Reed-Muller Expressions

Tahseen Kamal
B. Sc. (CSE)

A Thesis submitted to

Department of Computer Science and Engineering
Faculty of Sciences and Engineering
East West University
Dhaka-1212, Bangladesh

as partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering

April 30, 2006

This work is done with the financial support of "Imdad-Sitara Khan Foundation, Saratoga, California, USA" under AABEA Imdad-Sitara Khan Foundation Fellowship program

DECLARATION

This is certified that this thesis is an original work and was done by me and it has not been submitted elsewhere for the requirement of any degree or diploma or for any other purposes except for publication.

Signature of the candidate

Tahseen Kamal

Tahseen Kamal)

ACCEPTANCE

The **Thesis** entitled **An Application Specific Integrated Circuit for Optimization of Fixed Polarity Reed-Muller Expressions** submitted by Tahseen Kamal, I.D. No. 2005-2-96-003, to the Department of Computer Science and Engineering, East West University, Dhaka-1212, Bangladesh is accepted as satisfactory for partial fulfillment of the requirements for the degree of Master of Science (MS) in Computer Science and Engineering on April 30, 2006.

BOARD OF EXAMINERS

1. _____ 30.04.06 Chairman and Thesis Supervisor

Prof. Dr. Md. Mozammel Huq Azad Khan
Dean, Faculty of Sciences and Engineering
East West University
Dhaka-1212, Bangladesh

2. _____ 30/4/06 Member (Exofficio)

Mr. Syed Akhter Hossain
Associate Professor and Chairperson
Department of Computer Science and Engineering
East West University
Dhaka-1212, Bangladesh

3. _____ 30/4/06 External Member

Prof. Dr. A. B. M. Harun-ur Rashid
Department of Electrical and Electronic Engineering
Bangladesh University of Engineering and Technology
Dhaka-1000, Bangladesh

Acknowledgement

I would like to start by thanking the **Imdad-Sitara Khan Foundation** for giving me th opportunity to conduct the Master of Science in Computer Science and Engineering by providing fellowship.

I must thank Dr. Md. Mozammel Huq Azad Khan, for being the best guide possible to me. Without his support this thesis and my MS would never be completed.

The person who has always guided me on this path is the department Chairperson, Mr. Syed Akhter Hossain. The thesis completion is a result of his support and care which I must admit.

Mr. Shahrier Kabir, Lab In-charge, East West University, should also be thanked for his technical support.

I must also thank my family members who are and will always be with me when I am in bad time.

But above everything I would like to thank His almighty, Allah, for giving me the ability, chance, patience and stamina to complete this task successfully.

Abstract

Classically logic functions are realized using AND-OR two-level circuits. Now a days, EXOR-based logic functions have become popular, because they have some specific advantages over AND-OR realizations [Sasao 1993a]. Two-level AND-EXOR logic is one of the EXOR-based logics, which is also known as Reed-Muller logic. There are seven classes of AND-EXOR logic expressions [Sasao 1993a]. A Fixed Polarity Reed-Muller (FPRM) expression is one of them which is canonical and uses a fixed polarity for each variable. An n-variable function has 2^n different polarity vectors; consequently, there are 2^n different FPRM expressions. The expression with minimum number of products is the minimum FPRM expression. Therefore, the minimization problem of FPRM expressions is to find a polarity vector that produces an FPRM expression with minimum number of products along with corresponding coefficients. There are many software methods for FPRM minimization which are sequential in nature and require exponential execution time. In this thesis an ASIC has been developed to minimize 3-variable FPRM expressions which is parallel in nature and requires constant time. This ASIC takes the minterm coefficients of a Boolean function as input. It generates all the polarity vectors for a three variable function and determines the optimum polarity and corresponding FPRM coefficients.

Table of Contents

Chapter 1
Introduction to AND-EXOR Expressions

1.1 Introduction

There are many ways a Boolean function can be represented. The most popular one is a *truth table* representation. The size of the *truth table* increases exponentially with the increase of *n* (*number of variables in the function)*. Another commonly used approach is the *AND-OR* representation, also known as the *Sum-of-Products* representation which is more compact than the *truth table* representation. During the last two decades, researchers focused their eyes extensively on realizing logic functions using *EXOR-based* circuits which is more compact than the *AND-OR* representation. For example, for representing a parity function an *AND-OR* representation takes 2^{n-1} product terms, whereas *AND-EXOR* representation takes *n* product terms [Sasao 1993a]. In this chapter, classification of AND-EXOR logic is presented along with detailed description of each type, their uses and advantages are also discussed [Sasao 1993a].

1.2 AND-EXOR Expansions of Logic Functions

The following three expansions are the basis of the AND-EXOR representation of logic functions [Sasao 1991, 1993a, 1995]:

i. Shannon expansion
ii. Positive Davio expansion
iii. Negative Davio expansion

1.2.1 Shannon Expansion

The Shannon expansion of a logic function is defined as follows.

Theorem 1.1 (Shannon expansion) An arbitrary n-variable function, $f(x_1, x_2, ..., x_n)$ can be expanded using the following expansion.

$$f(x_1, x_2, ..., x_n) = \bar{x}_i f_0 + x_i f_1 \tag{1.1}$$

Here, we obtain f_0 by putting 0 (zero) for x_i in $f(x_1, x_2, ..., x_n)$ and f_1 by putting 1 (one) for x_i in $f(x_1, x_2, ..., x_n)$.

Thus, $f_0 = f(x_1,\ldots x_{i-1},0,x_{i+1},\ldots,x_n)$ and $f_1 = f(x_1,\ldots x_{i-1},1,x_{i+1},\ldots,x_n)$.

Proof. The theorem is proved using proof by induction. If we put $x_i=0$ in (1.1), we have $f(x_1,\ldots x_{i-1},0,x_{i+1},\ldots,x_n)=\bar{0}\cdot f_0+0\cdot f_1=f_0$. Again, if we put $x_i=1$ in (1.1), we get, $f(x_1,\ldots x_{i-1},1,x_{i+1},\ldots,x_n)=\bar{1}\cdot f_0+1\cdot f_1=f_1$. Thus we have the theorem. (Q.E.D)

The Shannon expansion can also be represented in the following way.

Lemma 1.1 (Shannon expansion) An arbitrary n-variable function $f(x_1,x_2,\ldots,x_n)$ can be expanded using the following expansion.

$$f(x_1,x_2,\ldots,x_n)=\bar{x}_i f_0 \oplus x_i f_1 \tag{1.2}$$

where, $f_0 = f(x_1,\ldots x_{i-1},0,x_{i+1},\ldots,x_n)$ and $f_1 = f(x_1,\ldots x_{i-1},1,x_{i+1},\ldots,x_n)$.

Proof. The sub-functions $\bar{x}_i f_0$ and $x_i f_1$ of (1.1) are mutually disjoint. So, + of (1.1) can be replaced by $\oplus$. Thus we have the lemma. (Q.E.D)

The circuit for the Shannon expansion is shown in Figure 1.1(a). In Shannon expansion, the variable x_i appears both as x_i and $\bar{x}_i$.

1.2.2 Positive Davio Expansion

The positive Davio expansion of a logic function is defined as follows.

Theorem 1.2 (Positive Davio expansion) An arbitrary n-variable function $f(x_1,x_2,\ldots,x_n)$ can be expanded using the following expansion.

$$f(x_1,x_2,\ldots,x_n)=f_0 \oplus x_i f_2 \tag{1.3}$$

where $f_0 = f(x_1,\ldots x_{i-1},0,x_{i+1},\ldots,x_n)$, $f_1 = f(x_1,\ldots x_{i-1},1,x_{i+1},\ldots,x_n)$ and $f_2 = f_0 \oplus f_1$.

Proof. Since $1 \oplus x_i = \bar{x}_i$, from (1.2) we can write $f(x_1,x_2,\ldots,x_n)=(1\oplus x_i)f_0 \oplus x_i f_1 = f_0 \oplus x_i(f_0 \oplus f_1) = f_0 \oplus x_i f_2$. Thus we have the theorem. (Q.E.D)

The circuit for the positive Davio expansion is shown in Figure 1.1(b). In positive Davio expansion, the variable x_i appears as only x_i.

1.2.3 Negative Davio Expansion

The negative Davio expansion of a logic function is defined as follows.

Theorem 1.3 (Negative Davio expansion) An arbitrary n-variable function $f(x_1, x_2, ..., x_n)$ can be expanded using the following expansion.

$$f(x_1, x_2, ..., x_n) = f_1 \oplus \overline{x_i} f_2 \tag{1.4}$$

where $f_0 = f(x_1, ..., x_{i-1}, 0, x_{i+1}, ..., x_n)$, $f_1 = f(x_1, ..., x_{i-1}, 1, x_{i+1}, ..., x_n)$ and $f_2 = f_0 \oplus f_1$.

Proof. Since $1 \oplus \overline{x_i} = x_i$, from (1.2) we can write $f(x_1, x_2, ..., x_n) = \overline{x_i} f_0 \oplus (1 \oplus \overline{x_i}) f_1 = f_0 \oplus \overline{x_i}(f_0 \oplus f_1) = f_1 \oplus x_i f_2$. Thus we have the theorem. (Q.E.D)

The circuit for the negative Davio expansion is shown in Figure 1.1(c). In negative Davio expansion, the variable x_i appears as only $\overline{x_i}$.

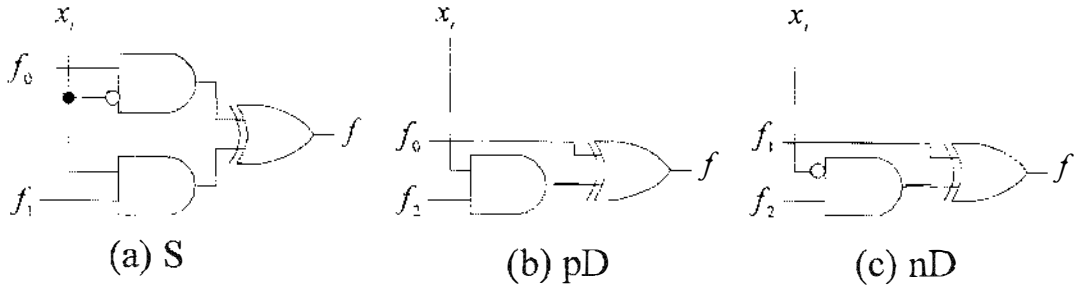


Figure 1.1 Circuits corresponding to three types of expansions.

1.3 AND-EXOR Expansion Trees of Logic Functions

By applying the three expansions of (1.2), (1.3), and (1.4) for each variable of a logic function, we can represent a logic function using the following expansion trees [Sasao 1995].

1.3.1 Shannon Tree

By applying the Shannon expansion recursively to a logic function, we can represent a logic function by a Shannon tree. Figure 1.2 shows an example of a Shannon tree for a 3-variable function f, where the symbol S denotes the Shannon expansion. The terminal nodes represent binary constants. Each edge has a literal *(uncomplemented or complemented form of a variable)* $\in \{x_i, \overline{x_i}\}$ of a variable as a label. A product of the literals from the root node to a terminal node represents a product term. For example, the right most path represents the product term $\overline{x_1} x_2 x_3$. The expression corresponding to this tree is,

$$\begin{aligned} f = {} & f_{000}\, \overline{x_1}\, \overline{x_2}\, \overline{x_3} \oplus f_{001}\, \overline{x_1}\, \overline{x_2}\, x_3 \oplus f_{010}\, \overline{x_1}\, x_2\, \overline{x_3} \oplus f_{011}\, \overline{x_1}\, x_2\, x_3 \oplus \\ & f_{100}\, x_1\, \overline{x_2}\, \overline{x_3} \oplus f_{101}\, x_1\, \overline{x_2}\, x_3 \oplus f_{110}\, x_1\, x_2\, \overline{x_3} \oplus f_{111}\, x_1\, x_2\, x_3 \end{aligned} \tag{1.5}$$

This expression is a canonical expression (where each product term contains all variables in the function). The products having zero coefficients disappear. Thus, the number of non-zero coefficients equals to the number of products in the expression.

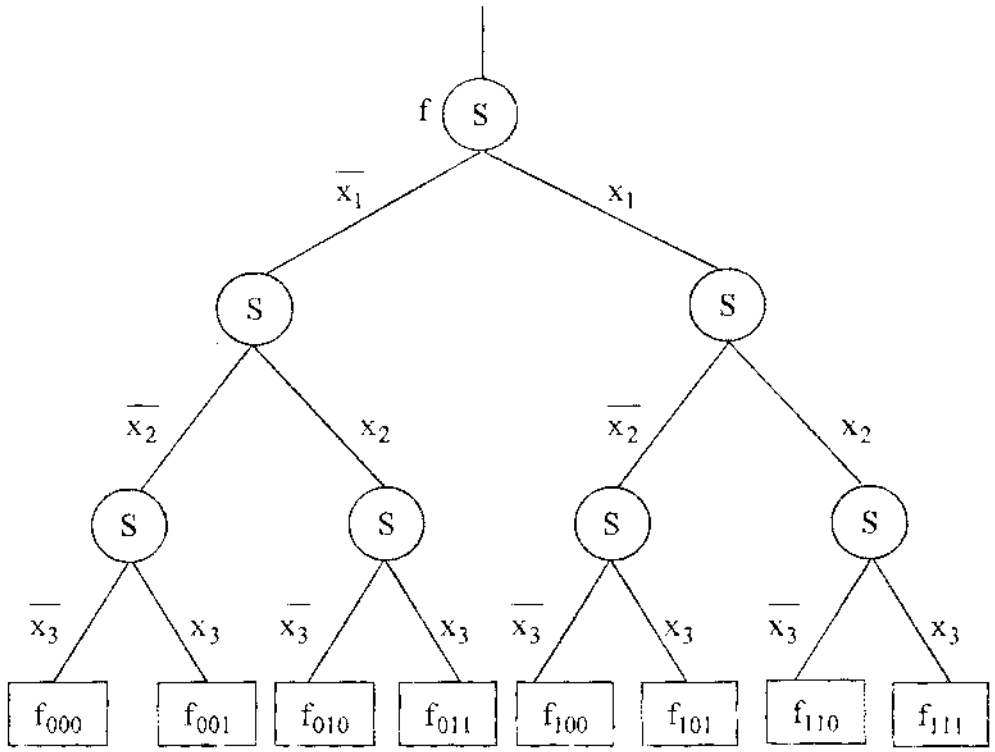


Figure 1.2 A Shannon tree for 3-variable function.

1.3.2 Positive Davio Tree

By applying the positive Davio expansion recursively to a logic function, we can represent a logic function by a positive Davio tree. Figure 1.3 shows an example of a positive Davio tree for a three-variable function f, where the symbol pD denotes the positive Davio expansion. Each edge has a literal $x_i^* \in \{1, x_i\}$ of a variable as a label. The expression corresponding to this tree is

$$f = f_{000}\, 1\cdot 1\cdot 1 \oplus f_{002}\, 1\cdot 1\cdot x_3 \oplus f_{020}\, 1\cdot x_2\cdot 1 \oplus f_{022}\, 1\cdot x_2\, x_3 \oplus$$
$$f_{200}\, x_1\cdot 1\cdot 1 \oplus f_{202}\, x_1\cdot 1\cdot x_3 \oplus f_{220}\, x_1\, x_2\cdot 1 \oplus f_{222}\, x_1\, x_2\, x_3 \qquad \textbf{(1.6)}$$

This expression is a canonical expression and uses only positive (uncomplemented) literals.

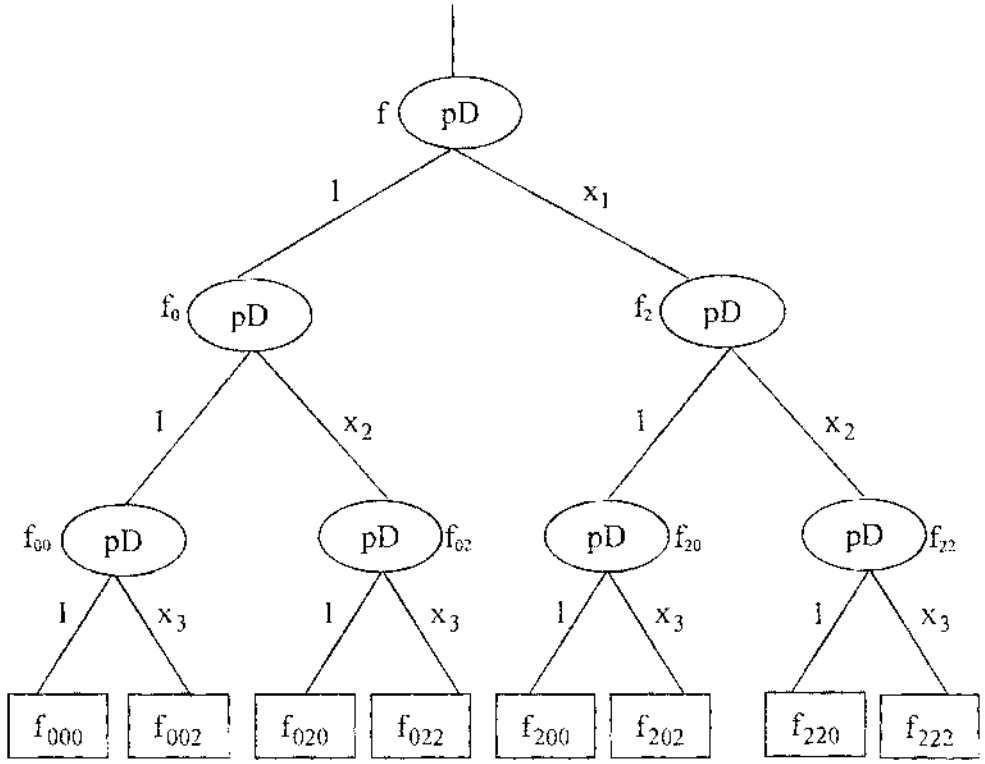


Figure 1.3 A positive Davio tree for 3-variable function.

1.3.3 Reed-Muller Tree

If we use either the positive or the negative Davio expansion for each variable, we can represent a logic function by a Reed-Muller tree. Figure 1.4 shows an example of a Reed-Muller tree for a 3-variable function f, where the symbols pD and nD denote the Positive and the Negative Davio expansions, respectively. In this tree, variable x_1 and x_3 use the Positive Davio expansion and variable x_2 uses the Negative Davio expansion. The expression corresponding to this tree is,

$$\begin{aligned} f = {} & f_{010}\, 1\cdot 1\cdot 1 \oplus f_{012}\, 1\cdot 1\cdot x_3 \oplus f_{020}\, 1\cdot \overline{x_2}\cdot 1 \oplus f_{022}\, 1\cdot \overline{x_2}\, x_3 \oplus \\ & f_{210}\, x_1\cdot 1\cdot 1 \oplus f_{212}\, x_1\cdot 1\cdot x_3 \oplus f_{220}\, x_1\, \overline{x_2}\cdot 1 \oplus f_{222}\, x_1\, \overline{x_2}\, x_3 \end{aligned} \tag{1.7}$$

This expression is canonical for a given way of expansion. There are 2^n different expansions for an n-variable function. Different expansions will produce expressions with different number of products.

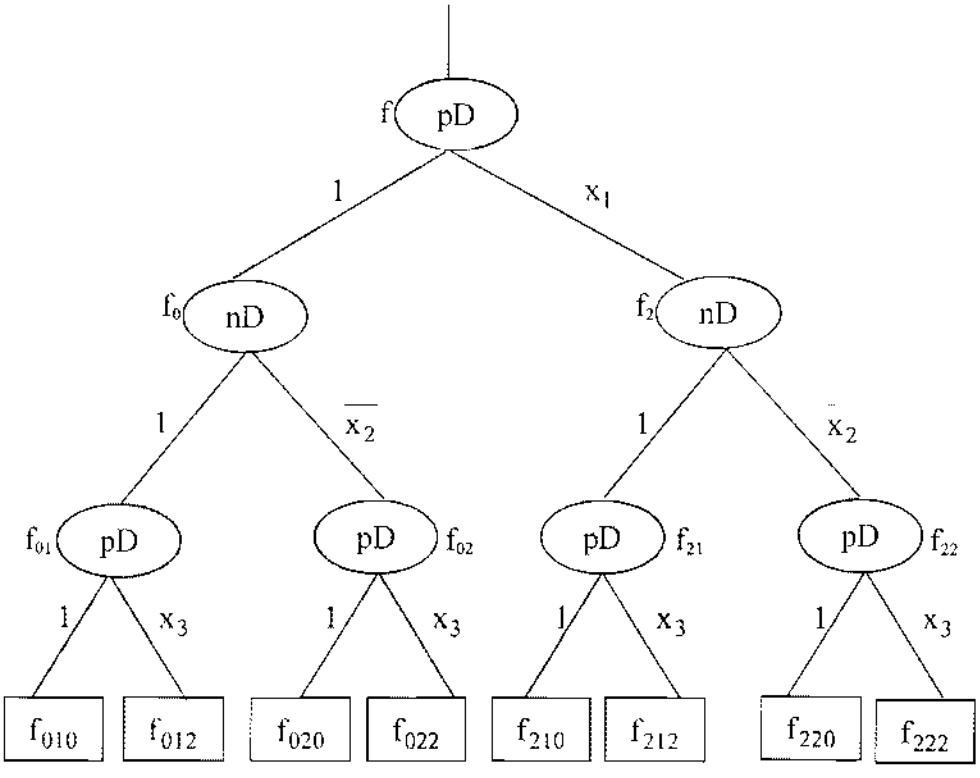


Figure 1.4 A Reed-Muller tree for 3-variable function.

1.3.4 Kronecker Tree

If we use any of the Shannon, the positive Davio and the negative Davio expansions for each variable, we can represent a logic function by a Kronecker tree. Figure 1.5 shows an example of a Kronecker tree for a three-variable function f, where the symbols S, pD and nD denote the Shannon, the positive Davio and the negative Davio expansions, respectively. In this tree, variable x_1 uses the Shannon expansion, variable x_2 uses the positive Davio expansion, and variable x_3 uses the negative Davio expansion. The expression corresponding to this tree is,

$$\begin{aligned} f = {} & f_{001}\,\overline{x_1}\cdot 1\cdot 1 \oplus f_{002}\,\overline{x_1}\cdot 1\cdot \overline{x_3} \oplus f_{021}\,\overline{x_1}\cdot x_2\cdot 1 \oplus f_{022}\,\overline{x_1}\cdot x_2\,\overline{x_3} \oplus \\ & f_{101}\,x_1\cdot 1\cdot 1 \oplus f_{102}\,x_1\cdot 1\cdot \overline{x_3} \oplus f_{121}\,x_1\,x_2\cdot 1 \oplus f_{122}\,x_1\,x_2\,\overline{x_3} \end{aligned} \tag{1.8}$$

This expression is canonical for a given way of expansion. There are 3^n different expansions for an n-variable function. Different expansions will produce expressions with different number of products.

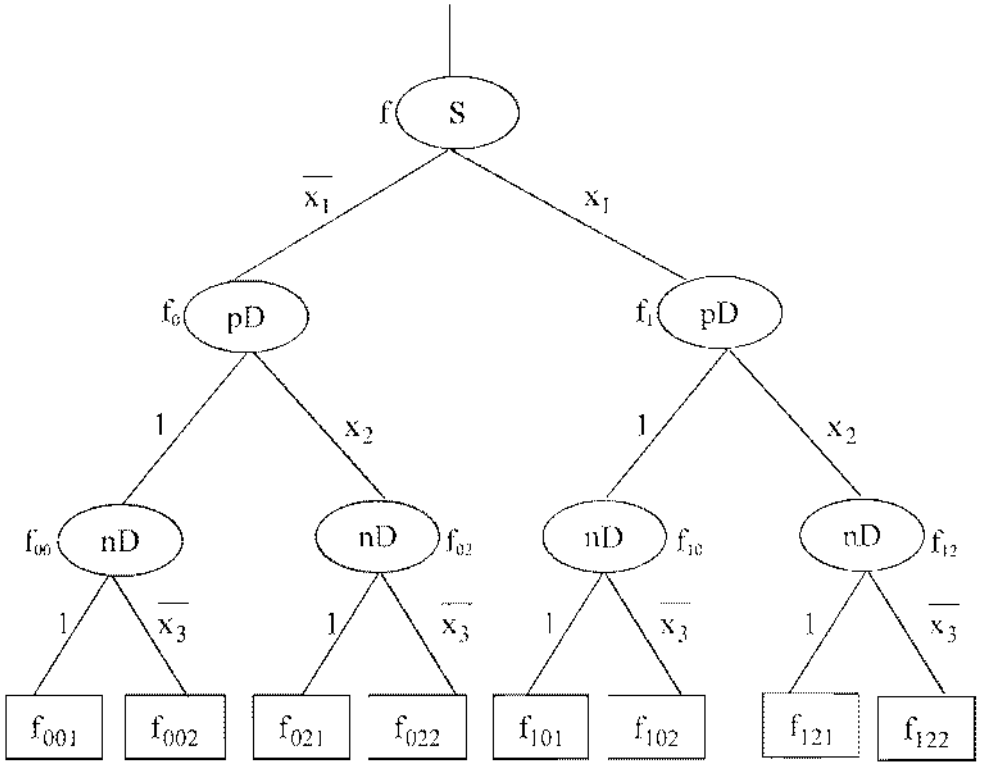


Figure 1.5 A Kronecker tree for 3-variable function.

1.3.5 Pseudo Reed-Muller Tree

If we use either the positive or the negative Davio expansion for each node, we can represent a logic function by a pseudo Reed-Muller tree. Figure 1.6 shows an example of a pseudo Reed-Muller tree for a three-variable function f. In this tree, variable x_1 uses the positive Davio expansion, variables x_2 and x_3 use both the positive and the negative Davio expansions. The expression corresponding to this tree is

$$\begin{aligned} f = {} & f_{000}\, 1\cdot 1\cdot 1 \oplus f_{002}\, 1\cdot 1\cdot \overline{x_3} \oplus f_{020}\, 1\cdot x_2\cdot 1 \oplus f_{022}\, 1\cdot x_2\, x_3 \oplus \\ & f_{120}\, x_1\cdot 1\cdot 1 \oplus f_{212}\, x_1\cdot 1\cdot x_3 \oplus f_{221}\, x_1\, \overline{x_2}\cdot 1 \oplus f_{222}\, x_1\, \overline{x_2}\, \overline{x_3} \end{aligned} \tag{1.9}$$

In a Kronecker tree, there are $2^n - 1$ nodes for n-variable functions. So for a given order of the input variables, there are 2^{2^n-1} different expansions for n-variable functions. There are $n!$ ordering of the input variables. Therefore, there are $n!\cdot 2^{2^n-1}$ different expansions for an n-variable functions. Different expansions will produce expressions with different number of products.

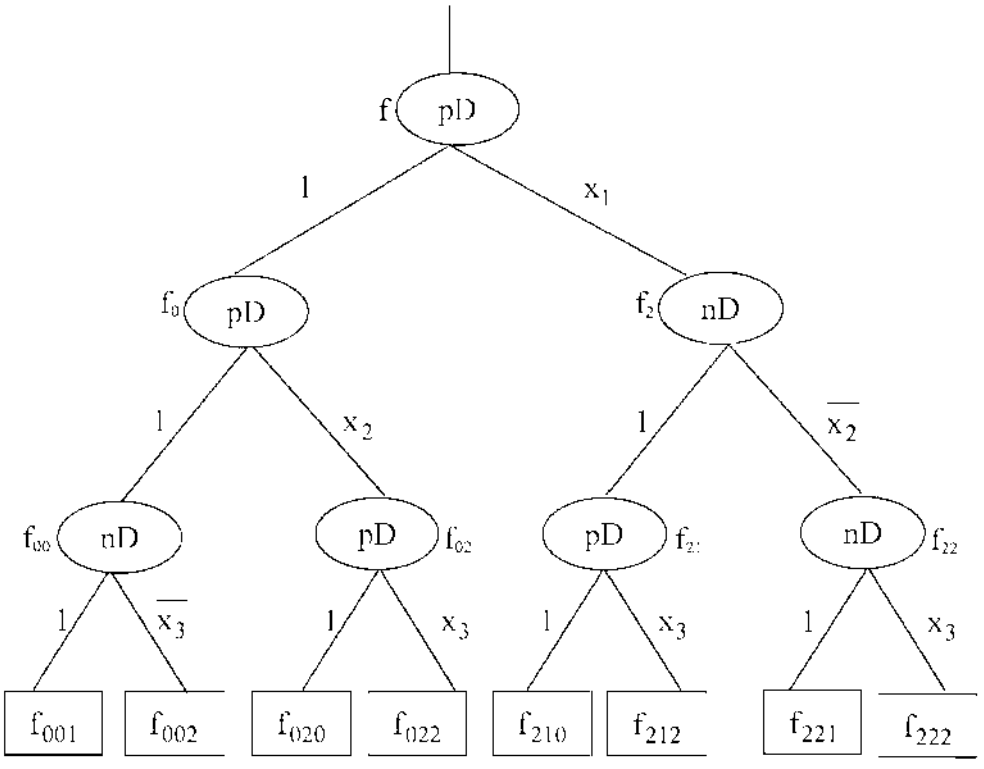


Figure 1.6 A pseudo Reed-Muller tree for 3-variable function.

1.3.6 Pseudo Kronecker Tree

If we use any of the Shannon, the positive Davio and the negative Davio expansions for each node, we can represent a logic function by a pseudo Kronecker tree. Figure 1.7 shows an example of a pseudo Kronecker tree for a three-variable function f. In this tree, variable x_1 uses the Shannon expansion, variable x_2 uses both the positive and the negative Davio expansions, and variable x_3 uses all of the Shannon, the positive Davio and the negative Davio expansions. The expression corresponding to this tree is

$$
\begin{aligned}
f = {} & f_{000}\,\overline{x_1}\cdot 1\cdot\overline{x_3} \oplus f_{001}\,\overline{x_1}\cdot 1\cdot x_3 \oplus f_{020}\,\overline{x_1}\,x_2\cdot 1 \oplus f_{022}\,\overline{x_1}\,x_2\,x_3 \oplus \\
& f_{111}\,x_1\cdot 1\cdot 1 \oplus f_{112}\,x_1\cdot 1\cdot\overline{x_3} \oplus f_{120}\,x_1\,\overline{x_2}\,x_3 \oplus f_{121}\,x_1\,\overline{x_2}\,x_3
\end{aligned}
\tag{1.10}
$$

In this tree, there are $2^n - 1$ nodes for n-variable functions. So, for a given order of the input variables, there are 3^{2^n-1} different expansions for n-variable functions. There are $n!$ ordering of the input variables. Therefore, there are $n!\cdot 3^{2^n-1}$ different expansions for an n-variable function. Different expansions will produce expressions with different number of products.

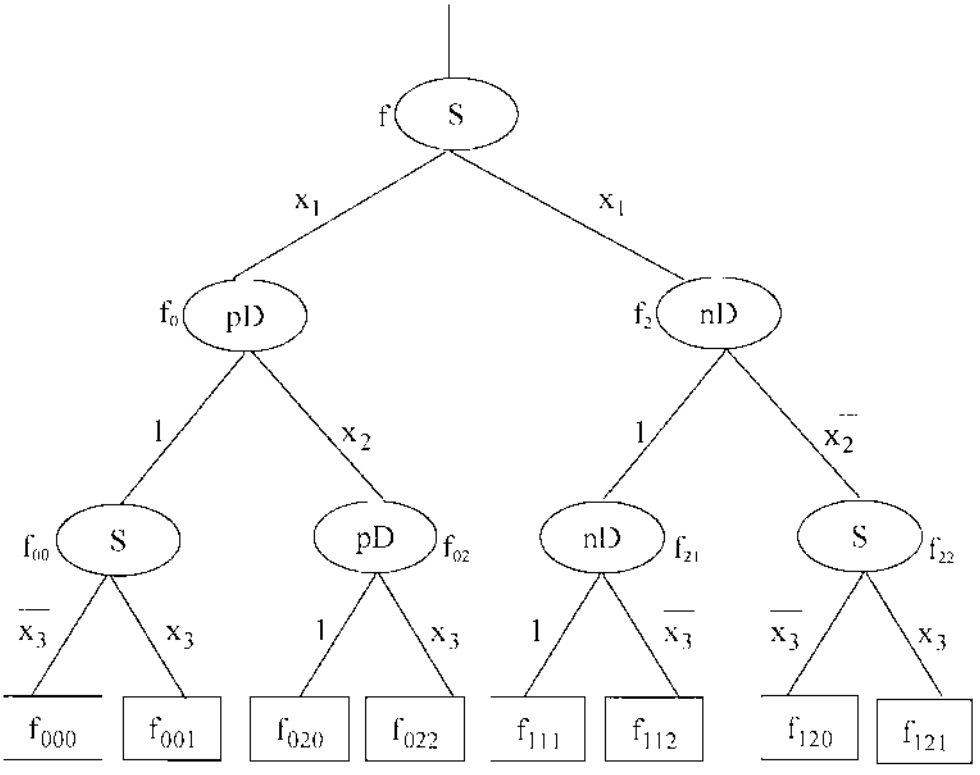


Figure 1.7 A pseudo Kronecker tree for 3-variable function.

1.4 AND-EXOR Representation of Canonical Sum of Products (CSOP) Expressions

A Shannon tree generates an expression of the form shown in (1.5). If we replace coefficient f with a in (1.5), we have the following expression:

$$
\begin{aligned}
f = {} & a_{000}\, \overline{x_1}\,\overline{x_2}\,\overline{x_3} \oplus a_{001}\, x_1\, \overline{x_2}\, x_3 \oplus a_{010}\, \overline{x_1}\, x_2\, \overline{x_3} \oplus a_{011}\, \overline{x_1}\, x_2\, x_3 \oplus \\
& a_{100}\, x_1\, \overline{x_2}\,\overline{x_3} \oplus a_{101}\, x_1\, x_2\, x_3 \oplus a_{110}\, x_1\, x_2\, \overline{x_3} \oplus a_{111}\, x_1\, x_2\, x_3
\end{aligned}
\tag{1.11}
$$

The expression of (1.11) is a canonical expression having all minterms. This is nothing but a sum of minterm expression or CSOP expression with OR replaced by EXOR, and is known as canonical EXOR-sum of products (ESOP) expression.

As a specific example, $f(x_1, x_2, x_3) = \overline{x_1}\,\overline{x_2}\, x_3 \oplus \overline{x_1}\, x_2\, x_3 \oplus x_1\, x_2\, \overline{x_3} \oplus x_1\, x_2\, x_3$ is an ESOP representation of the function $f(x_1, x_2, x_3) = \sum_m (1, 3, 6, 7)$.

1.5 EXOR-based Representation of Disjoint Sum of Products (DSOP) Expressions

A disjoint sum of products (DSOP) expression is defined as follows.

Definition 1.1 The disjoint sum of products (DSOP) expression for an n-variable function $f(x_1, x_2, \ldots, x_n)$ can be represented as

$$f(x_1, x_2, \ldots, x_n) = \sum + \; x_1^* x_2^* \ldots x_n^* \tag{1.12}$$

where $\sum +$ represents OR-sum, every instance of x_i^* in the expression can be 1, x_i or $\overline{x_i}$, and all the product terms are mutually disjoint.

The following lemma holds for the CSOP expressions.

Lemma 1.2 The canonical sum of products (CSOP) expression is a disjoint sum of products (DSOP) expression.

Proof. The lemma holds, since all the minterms of a function are mutually disjoint. (Q.E.D)

Lemma 1.3 The disjoint sum of products (DSOP) expression for an n-variable function $f(x_1, x_2, \ldots, x_n)$ can be represented as

$$f(x_1, x_2, \ldots, x_n) = \sum \oplus \; x_1^* x_2^* \ldots x_n^* \tag{1.13}$$

where $\sum \oplus$ represents EXOR-sum, every instance of x_i^* in the expression can be 1, x_i or $\overline{x_i}$, and all the product terms are mutually disjoint.

Proof. As all the product terms of (1.12) are mutually disjoint, the + operator can be replaced with $\oplus$. Thus, we have the lemma. (Q.E.D)

1.2 Types of AND-EXOR Logic Expressions

There are different ways of classification of AND-EXOR logic expressions in the literature. According to the classification of [Sasao 1991, 1993a, 1995], there are seven types of AND-EXOR logic expressions. They are,

- Positive Polarity Reed-Muller (PPRM) expressions
- Fixed Polarity Reed-Muller (FPRM) expressions
- Pseudo Reed-Muller (PSDRM) expressions
- Generalized Reed-Muller (GRM) expressions
- Kronecker (KRO) expressions
- Pseudo Kronecker (PSDKRO) expressions
- Exclusive-OR Sum of Products (ESOP) expressions.

These seven classes of AND-EXOR logic expressions are discussed elaborately in the following sections.

1.6.1 Positive Polarity Reed-Muller (PPRM) Expressions

A positive Davio tree generates an expression of the form as shown in (1.6). If we replace coefficient f with b and subscript 2 with 1 in (1.6), we have the following expression:

$$f = b_{000} \oplus b_{001}\, x_3 \oplus b_{010}\, x_2 \oplus b_{011}\, x_2 x_3 \oplus b_{100}\, x_1 \oplus b_{101}\, x_1 x_3 \oplus b_{110}\, x_1 x_2 \oplus b_{111}\, x_1 x_2 x_3 \tag{1.14}$$

This expression uses only positive literals, and is called a positive polarity Reed-Muller (PPRM) expression. According to some authors the polarities of all the variables are 1 and according to some others the polarities are 0. This expression is a canonical expression and no minimization problem exists.

For example, $F(x_1,x_2,x_3)=x_1x_2 \oplus x_2x_3 \oplus x_1x_3$ is a PPRM expression for the function $F(x_1,x_2,x_3)=\sum_m(3,5,6,7)$.

1.6.2 Fixed Polarity Reed-Muller (FPRM) Expressions

A Reed-Muller tree generates an expression of the form as shown in (1.7). In (1.7) we have the following observations:

- For a subscript $i \in \{0,1\}$ of a coefficient f, the corresponding literal of the associated product term appears as $x_i^* = 1$.
- For a subscript $i = 2$ of a coefficient f, the corresponding literal of the associated product term appears as $x_i^* \in \{ x_i, \overline{x_i} \}$ depending on the expansion used.

In this work, the polarity of an uncomplemented variable is represented by 0 and that of a complemented variable by 1. The reverse polarity convention is also used in the literature, but throughout this work we will follow this convention.

If we replace coefficient f with b, subscript 1 with 0, subscript 2 with 1, and literals x_i and $\overline{x_i}$ with x_i^* in (1.7), we have the following expression:

$$f = b_{000} \oplus b_{001} x_3^* \oplus b_{010} x_2^* \oplus b_{011} x_2^* x_3 \oplus b_{100} x_1^* \oplus b_{101} x_1^* x_3^* \oplus b_{110} x_1^* x_2^* \oplus b_{111} x_1^* x_2^* x_3^* \tag{1.15}$$

where, $x_i^* = \begin{cases} x_i & if\ polarity\ of\ x_i\ is\ 0 \\ \overline{x_i} & if\ polarity\ of\ x_i\ is\ 1 \end{cases}$

expression uses fixed polarity for a given variable and is called a fixed polarity Reed-Muller (FPRM) expression. This expression is canonical for a given polarity vector of the variables. In general, the FPRM expression for a logic function is represented as follows:

For example, the FPRM expression for a 3-variable function $f(x_1, x_2, x_3)$ with polarity vector p = (101) can be represented as,

$$f(x_1, x_2, x_3) = b_{000} \oplus b_{001} x_3 \oplus b_{010} \bar{x}_2 \oplus b_{011} \bar{x}_2 x_3 \oplus b_{100} x_1 \oplus b_{101} x_1 x_3 \oplus b_{110} x_1 \bar{x}_2 \oplus b_{111} x_1 \bar{x}_2 x_3$$

In specific example, $F(x_1, x_2, x_3) = \bar{x}_1 x_2 \oplus x_2 \bar{x}_3 \oplus \bar{x}_1 \bar{x}_3$ is an FPRM expression for the function $f(x_1, x_2, x_3) = \sum_m (1, 4, 5, 7)$ with the polarity vector p = (101).

1.3 Pseudo Reed-Muller (PSDRM) Expressions

A pseudo Reed-Muller tree generates an expression of the form as shown in (1.9). This type of expression is called a pseudo Reed-Muller (PSDRM) expression. For a given order of the input variables, 2^{2^n-1} different pseudo Reed-Muller trees exist. Different orderings of the input variables produce different expressions. There are $n!$ different orderings of n input variables. Therefore, $n! \cdot 2^{2^n-1}$ different PSDRM expressions exist for an n-variable function. Different expressions will produce different number of products. An expression with minimum number of products is the minimum PSDRM expression for a given function. Therefore, the minimization problem of PSDRM expressions is to find an expression having minimum number of products. For example, $F(x_1, x_2, x_3) = x_1 x_2 \oplus \bar{x}_2 x_3 \oplus x_1 \bar{x}_3$ is a PSDRM expression for the function $f(x_1, x_2, x_3) = \sum_m (1, 4, 6, 7)$.

1.4 Generalized Reed-Muller (GRM) Expressions

A generalized Reed-Muller (GRM) expression is derived from a PPRM expression. The PPRM expression for a 3-variable function is represented as

$$\begin{aligned} f(x_1, x_2, x_3) = {} & b_{000} \oplus b_{001} x_3 \oplus b_{010} x_2 \oplus b_{011} x_2 x_3 \oplus \\ & b_{100} x_1 \oplus b_{101} x_1 x_3 \oplus b_{110} x_1 x_2 \oplus b_{111} x_1 x_2 x_3 \end{aligned} \tag{1.16}$$

If we freely choose the polarities of the literals in (1.16), we have the following expression.

$$\begin{aligned} f(x_1, x_2, x_3) = {} & b_{000} \oplus b_{001} x_3^* \oplus b_{010} x_2^* \oplus b_{011} x_2^* x_3 \oplus b_{100} x_1^* \oplus \\ & b_{101} x_1^* x_3^* \oplus b_{110} x_1^* x_2^* \oplus b_{110} x_1^* x_2^* x_3^* \end{aligned} \tag{1.17}$$

where every existence of x_i^* denotes either x_i or $\bar{x}_i$.

of expression is called generalized Reed Muller (GRM) expression. In a GRM both the positive and the negative literals may appear at the same time for a given In a GRM expression, no two products have the same set of variables. For an n-variable the total number of literals is $n \cdot 2^{n-1}$. Thus $2^{n2^{n-1}}$ different GRM expressions exist for able function. Different expressions will produce different number of products. An with minimum number of products is the minimum GRM expression for a given Therefore, the minimization problem of GRM expressions is to find an expression the minimum number of products. For example, $G(x_1,x_2,x_3)=\overline{x_1}\,\overline{x_2} \oplus x_2\overline{x_3} \oplus x_1x_3$ is a pression for the function $f(x_1,x_2,x_3)= \sum_m(0,1,2,5,6,7)$.

Kronecker (K O) Expressions

ker tree generates an expression of the form as shown in (1.8). This type of expression a Kronecker (KRO) expression. There are 3^n different KRO expressions for an function. Different expressions will produce different number of products. An with minimum number of products is the minimum KRO expression for a given Therefore, the minimization problem of KRO expressions is to find an expression minimum number of products. For example, $F(x_1,x_2,x_3)=x_1x_2x_3 \oplus \overline{x_1}\,\overline{x_2}\,\overline{x_3}$ is a KRO on for the function $f(x_1,x_2,x_3)= \sum_m(0,7)$.

Pseudo Kronecker (PSDK O) Expressions

o Kronecker tree generates an expression of the form as shown in (1.10). This type of ion is called a pseudo Kronecker (PSDKRO) expression. For a given order of the input es, 3^{2^n-1} different pseudo Kronecker trees exist. Different orderings of the input variables different expressions. There are $n!$ different orderings of n input variables. Therefore, 3^{2^n-1} different PSDKRO expressions exist for an n-variable function. Different expressions produce different number of products. An expression with minimum number of products is mum PSDKRO expression for a given function. Therefore, the minimization problem of KRO expressions is to find an expression having minimum number of products. For e, $F(x_1,x_2,x_3)=\overline{x_1} \oplus x_1x_2 \oplus x_1\overline{x_2}$ is a PSDKRO expression for the function $f(x_1,x_2,x_3)= \sum_m(0,1,2,3)$.

Exclusive-OR sum of products (ESOP) Expressions

sive-OR sum of products (ESOP) expression is the most general class of AND-EXOR ssions and can be represented as follows:

Exclusive-OR sum of products (ESOP) expression for an arbitrary n-variable function $(x_1, x_2, \ldots, x_n)$ can be represented as

$$F(x_1, x_2, \ldots, x_n) = \sum \oplus x_1^* x_2^* \ldots x_n^* \tag{1.18}$$

where $\sum \oplus$ represents EXOR-sum and each existence of x_i^* can be chosen as 1, x_i or $\overline{x_i}$ independently of the other choice.

x_i^* may assume any of the three values, the number of ESOP product terms in (1.18) will [illegible] Each of the 3^n ESOP product terms may be present or absent. So, there are 2^{3^n} different [illegible]ations of ESOP product terms, some of which satisfy the given Boolean function. The [illegible]ation with minimum number of ESOP product terms is the minimum ESOP expression [illegible] given Boolean function. For example, $F(x_1, x_2) = x_1 \oplus x_2 \oplus x_1 x_2 \oplus \overline{x_1}\,\overline{x_2}$ is a ESOP [illegible]ssion for the function $f(x_1, x_2) = \sum_m (0,1,2,3)$.

[illegible] Double Fixed Polarity Reed-Muller Expressions

[illegible] class of AND-EXOR expression has been proposed in [Hirayama 2001]. It is stated that [illegible] Fixed Polarity Reed-Muller (DFPRM) expressions are generalized FPRM expressions [illegible] require less product terms than FPRM expressions. The definition is elaborated below.

[illegible]ition 1.2 The polarity vector of a given FPRM F is denoted by $v(F)$. $\bar{v}(F)$ is defined as the [illegible] complement of $v(F)$.

[illegible] each polarity vector v_i denotes the polarity of the variable x_i; $v_i = 0$ means the positive [illegible] expansion is used for the variable x_i and $v_i = 1$ means the negative Davio expansion is [illegible] instead.

[illegible]ition 1.3 Let F_a and F_b be FPRMs such that $v(F_a) = \bar{v}(F_b)$, where we assume that the n-[illegible] of $v(F_a)$ is 0 without loss of generality. The EXOR combination of the two FPRMs, $F_a \oplus F_b$, is called a *Double Fixed-Polarity Reed-Muller expression* (DFPRM) *with the polarity* [illegible] $v(F_a)$.

[illegible] example, we have two FPRM expressions $F_a(x_1, x_2, x_3, x_4) = x_1 \oplus \overline{x_2}\,\overline{x_3} x_4$ and [illegible] $x_2, x_3, x_4) = \overline{x_1} x_2 x_3 \overline{x_4} \oplus \overline{x_1}\,\overline{x_4}$. Here, $v(F_a) =$ [0, 1, 1, 0] and $v(F_b) =$ [1, 0, 0, 1]. [illegible] $= F_a \oplus F_b = x_1 \oplus \overline{x_2}\,\overline{x_3} x_4 \oplus \overline{x_1} x_2 x_3 \overline{x_4} \oplus \overline{x_1}\,\overline{x_4}$ is a DFPRM. The polarity of F is [0, 1, 1, 0].

Relations among Various Classes of AND-EXOR Logic Expressions

The relations among various classes of AND-EXOR logic expressions are stated in the following theorem [Sasao 1991].

Theorem 1.4 Suppose that PPRM, FPRM, PSDRM, GRM, KRO, PSDKRO, and ESOP denote classes of AND-EXOR logic expressions. Then the following relations hold:

i. PPRM $\subset$ FPRM
ii. FPRM $\subset$ PSDRM
iii. FPRM $\subset$ KRO
iv. KRO $\subset$ PSDKRO
v. PSDRM $\subset$ PSDKRO
vi. PSDRM $\subset$ GRM

Proof: Relations (i) to (v) are trivial and follow from the definitions. From definition, a PSDRM is a GRM and relation (vi) holds. Thus we have the theorem. (Q.E.D)

The relations among the various classes of AND-EXOR logic expressions stated in theorem (1.4) are shown in Figure 1.8 [Sasao 1991].

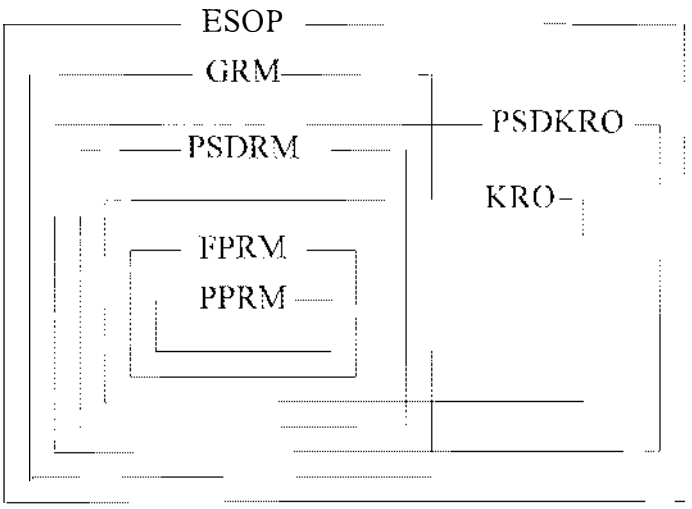


Figure 1.8 Relation among various types of AND-EXOR expressions

The relations among various classes of AND-EXOR logic expressions are discussed using the following examples [Sasao 1991]:

- $x_1x_2 \oplus x_2x_3 \oplus x_1x_3$ is a PPRM expression, since all the literals are positive.
- $x_1\bar{x}_2 \oplus \bar{x}_2x_3 \oplus x_1x_3$ is a FPRM expression, but not a PPRM expression, since x_1 and x_3 have positive literals, but x_2 has negative literals.

- $x_1x_2 \oplus \overline{x_2}x_3 \oplus x_1\overline{x_3}$ is a PSDRM expression, but not a FPRM expression, since x_2 and x_3 have literals of both polarities.
- $x_1x_2x_3 \oplus \overline{x_1}\,\overline{x_2}\,\overline{x_3}$ is a KRO expression since x_1, x_2 and x_3 have literals of both polarities.
- $\overline{x_1} \oplus x_1x_2 \oplus x_1\overline{x_2}$ is a PSDKRO expression, but not a KRO expression.
- $\overline{x_1} \oplus x_1x_2 \oplus x_1\overline{x_2}$ is a PSDKRO expression, but not a PSDRM expression, since it contains two products of the highest degree.
- $x_1 \oplus x_2 \oplus \overline{x_1}\,\overline{x_2}$ is a GRM expression, but not a PSDRM expression.
- $x_1 \oplus x_2 \oplus \overline{x_1}\,\overline{x_2}$ is a GRM expression, but not a PSDKRO expression.
- $x_1x_2x_3 \oplus \overline{x_1}\,\overline{x_2}\,\overline{x_3}$ is a KRO expression, but not a GRM expression, since it contains two products of the highest degree.
- $x_1 \oplus x_2 \oplus x_1x_2 \oplus \overline{x_1}\,\overline{x_2}$ is an ESOP expression, but neither GRM nor PSDKRO expressions.

[illegible] Advantages of AND-EXOR Logic

AND-EXOR logic exhibits specific advantages in the following areas [Sasao 1993a].

Testability

AND-EXOR logic shows the following advantages in the field of testability:

1. PLA implementation of PPRM, FPRM and GRM expressions are very easy to test [Reddy 1972, Saluja 1975, Fujiwara 1985, Sasao 1994, 1997].

2. In AND-EXOR two-level networks, tests that detect all detectable struck-at faults can be generated in polynomial time of the number of the products. On the other hand, in AND-OR two-level networks, the test generation problem is not polynomial time solvable [Toida 1992].

3. AND-EXOR circuits are more amenable to efficient testing strategies than their Boolean counterpart [Mukhopadhyay 1970, Bosslich 1985, Helliwell 1988, Harking 1990].

Products Requirements

Experiments with the existing minimization methods show the following advantages of AND-EXOR logic in the field of products requirements:

1. For symmetric functions, ESOP expressions never require more products than SOP expressions [Rollwage 1993].

2. For arithmetic functions, the number of products tends to decrease in the following order: PPRM, FPRM, SOP, KRO, PSDRM, PSDKRO, GRM and ESOP [Sasao

1991, 1993b, Debnath 1995]. Therefore, for arithmetic functions, KRO, PSD PSDKRO, GRM, and ESOP expressions require fewer products than expressions.

c. For pseudo-randomly generated function with 2^{n-1} true minterms, the numb products tends to decrease in the following order: PPRM, FPRM, KRO, PSD PSDKRO, SOP, GRM, and ESOP [Sasao 1991, 1993a, Debnath 1995]. There for pseudo-randomly generated functions with 2^{n-1} true minterms, GRM and E expressions require fewer products than SOP expressions.

d. For 4-variable functions, the average number of products decreases in the follo order: FPRM, KRO, PSDRM, SOP, PSDKRO, and ESOP [Sasao 1991, 1 Therefore, for 4-variable functions, PSDKRO and ESOP require on average products than SOPs.

e. For 5-variable functions, the average number of products decreases in the follo order: KRO, PSDRM, PSDKRO, and ESOP [Sasao 1991, 1993a].

f. For 6-variavble functions, an ESOP requires at most 16 products whereas a requires 32 products for realizing an arbitrary function [Sasao 1991, 1993a].

From the above discussion, it can be summarized that in general PSDKRO, GRM, and E requires fewer products than SOP.

Synthesis and Minimization Techniques

AND-EXOR logic supports design methods, which involve algebraic techniques simil those encountered with the algebra of real numbers [Mukhopadhyay 1970, Harking 1 Davio 1978].

VLSI Design

AND-EXOR logic circuits exhibit a modular structure which may make them suitab VLSI design [Fleisher 1983, Helliwell 1988, Green 1991].

Multiple-valued Logic Synthesis

The techniques for synthesis and minimization of AND-EXOR logic extend readil incorporate multiple-valued logic circuits [Green 1976, 1987, Sasao 1993b].

Application as Tool

The AND-EXOR logic expressions have the following applications as tool:

a. Fault of any logic circuit can be detected by verification of its Reed-Muller coefficients [Damarla 1989].

b. Boolean matching can be detected using FPRM representation as a tool [Tsai 1994a].

c. Symmetry of Boolean functions can be detected using FPRM expressions as a tool [Tsai 1996].

d. Boolean functions can be classified using FPRM expressions as a tool [Tsai 1997].

Chapter 2
Minimization of Fixed Polarity Reed-Muller Expressions

2.1 Introduction

Logic circuits may be minimized as AND-OR expression using established techniques such as the K-map, Quine-McCluskey method, Espresso, etc. The starting point, generally, is the sum of products (SOP) forms, and the aim is to reduce the number of terms/literals. The minimization of boolean functions can also be done as AND-EXOR expressions. If we consider the Fixed Polarity Reed-Muller expressions and have an n variable function, then we have 2^n different polarity vectors. So there are 2^n distinct FPRMs for an n variable function. Different expressions will have different number of products. An expression with minimum number of products is the minimum FPRM expression for a given function. Therefore, the minimization problem of FPRM expressions is to find a polarity vector that produces an FPRM expression with minimum number of products.

2.2 Literature Review on Minimization of Fixed polarity Reed-Muller (FPRM) Expressions

For completely and incompletely specified functions, numerous exact and heuristic minimization algorithms for FPRMs exist. We discuss some of the representative methods.

2.2.1 Fast exact and quasi-minimal minimization of highly testable fixed polarity AND EXOR canonical networks [Sarabi 1992]

From chapter 1 we know about the different classes of AND-EXOR expressions and their advantages. Here, we will discuss about a fast exact and quasi-minimal algorithm which minimizes FPRM canonical networks.

In order to differentiate between the Boolean product terms and the terms in Reed-Muller forms, the term "monoterm" is used for the latter.

Definition 2.1 A monoterm is a product term in Reed- Muller canonical (RMC) forms.

Here the acronym RMC is the same expression as the PPRM expressions discussed in chapter 1. Here the FPRM is replaced by CGRM (Consistent Generalized Reed-Muller) expressions.

[Fisher 1974], the problem of CGRM minimization of a switching function can be into two steps. The first step is to identify the minimal polarity and the second is to the CGRM expansion of the function with this polarity.

ient method for realizing a CGRM expansion of switching functions is by operating on disjoint cubes which represent the function [Fisher 1974, Schafer 1991, Varma 1991]. method, the function is represented by disjoint cubes rather than minterms to reduce the requirements. Monoterms representing each cube are expanded and those occurring in number of cubes are retained as the ones representing the function. The fast method here uses the new operations of cube commonality, difference, and symmetric together with a fast Gray-code approach to realize a CGRM expansion. Before the method, the monoterms representing each cube, originally reported by [Fisher the case of CGRM, are given by Theorem 2.1.

em 2.1 The monoterms originating from a cube for the RMC expansion are all the cubes their 1s in the same literal positions as the 1s of the original cube and either "0" or "-" literal positions of the original cube. These monoterms is referred to as monoterms the cube.

Table 2.1 Various operators for a single bit

(a) Equivalence

$\equiv$	0	1	—
0	1	0	—
1	0	1	—
—	—	—	—

(b) Cube commonality

Γ	0	1	—
0	0	1	—
1	1	1	Φ
—	—	Φ	—

(c) Cube difference

—	0	1	—
0	ϵ	—	1
1	ϵ	ϵ	Φ
—	ϵ	Φ	ϵ

ocess of realization of CGRM expansion for a given polarity is outlined in the following First, the function is represented as a set of disjoint cubes. The cubes are then operated by alence operation with the polarity of the CGRM. The symmetric difference of all these found and monoterms representing each of the resulting cubes are generated in a Gray- der. Finally, the Equivalence operation with the polarity cube is performed on each of monoterms to give the CGRM expansion. The number of the resulting monoterms can be om the *inclusion-exclusion* principle. This number is given by the following theorem:

rem 2.2 Let $C_1, C_2, ..., C_n$ be a set of n disjoint cubes. Let S_k denote the sum of the number monoterms common in all possible k cubes. The number of monoterms representing the set is:

$$S_1 - 2S_2 + 4S_3 - 8S_4 + ... + (-2)^{k-1} S_k + ... + (-2)^{n-1} S_n .$$

e above equation, S_1 denotes the number of all monoterms that are only in one cube, S_2 the number of all monoterms that are common in any two cubes, etc.

gh identification of the minimal polarity is an NP-hard problem, certain features of the disjoint cubes can be used to reduce the required search space. Some of these features, if

[illegible] in an array, can be used to find the minimal polarity without any search. Others can just [illegible] the amount of search needed to find the exact solution. In the case that none of the [illegible] exist, the whole exhaustive search needs to be performed.

[illegible] number of expansion monoterms for a set of disjoint cubes is the difference between the [illegible] sum of the number of monoterms representing each cube and the number of monoterms that [illegible] subtracted because they occur in an even number of cubes. Both of these numbers change [illegible] different polarities. The minimal polarity is the one which results in the optimum balance [illegible] these two numbers resulting in the least number of expansion monoterms.

[illegible] are certain features of the function that can be used to reduce the search space for [illegible] the minimal polarity. For the case of functions that are comprised of only one cube, [illegible] minimal polarity can be found directly without any search.

Theorem 2.3 The minimal polarities for a single cube are the polarities which match all the [illegible] in the cube. The number of such polarities is equal to $2^{N_{DC}}$ where N_{DC} is the number of DC-literals in the cube.

When a function is comprised of more than one cube, there are other features that if they exist, [illegible] to the search space reduction. Theorem 4 provides one criteria for identifying a minimal [illegible] literal based on the columns of an array of disjoint cubes, using Theorem 2.2.

Theorem 2.4 Let S^1_i denote the sum of S_is of the cubes which have a value of 1 in a given [illegible]. Let S^0_i denote the sum of S_is of the cubes which have a value of 0 in that column. Let S^{1-DC}_i denote the sum of S_is of the cubes that have both 1s and DCs in the column, assuming [illegible] has been changed to a 0. Let S^{0-DC}_i denote the sum of S_is of the cubes that have both 0s and DCs in the column. The corresponding minimal polarity literal for a column in the array of disjoint cubes should be changed when

$$\sum_{k=1}^{n}(-2)^{k-1}S_k^1+\sum_{k=2}^{n}(-2)^{k-1}S_k^{1-DC}<\tfrac{1}{2}\sum_{k=1}^{n}(-2)^{k-1}S_k^0+\sum_{k=2}^{n}(-2)^{k-1}S_k^{0-DC}$$

From Theorem 2.4, it is possible to infer the following theorem:

Theorem 2.5 For a column comprised of all 0s or all 1s, the corresponding minimal polarity literal is the same as the value in the column. (If the opposite is chosen, the number of monoterms representing the cubes would be doubled.) For a column comprised of all DC values, either 0 or 1 will be the minimal literal value.

In the Exact method of minimization, first it is checked if the function is only comprised of one cube or two. Direct solution for these cases is found using Theorems 2.3, 2.4 and 2.5. If there are more cubes involved, first Theorem 2.5 is used to identify any columns in the array of disjoint cubes for which minimal polarity literal can be found readily. All the other columns are set to the

zero polarity and a search for minimal polarity is performed in Gray-code order, changing one column at a time.

Then a fast heuristic approach to the minimization problem is introduced. The corresponding heuristics combine the characteristics of the overall number of monoterms and the ones subtracting, in order to identify the minimal CGRM polarity for a given array of disjoint cubes. Based on these heuristics, a priority of search for different polarities is devised and a minimization algorithm is introduced. This fast program can be used to bring more EXOR realization into the realm of logic synthesis.

2.2.2 Minimization of fixed-polarity AND/XOR canonical networks [Tsai 1994b]

In the paper [Tsai 1994b] the term GRM is same as the FPRM expressions discussed in chapter 1.

Let $f(x_1, x_2, ..., x_n)$ be a completely specified Boolean function. Each of x_i, where, $x_i \in \{0,1\}$ is a *vertex in the domain of the function. The set of vertices that the function evaluates to 1 is called the on-set* of *f*. The set of vertices that the function evaluates to *0* is called the *off-set* of *f*. A *cofactor of a function f, denoted* f_{x_i}, *is the function derived from f when* x_i *is set to 1.* Similarly, $f_{\bar{x}_i}$ is a cofactor of *f* when x_i is set to *0* in *f*. $|f|$ denotes the number of the on-set vertices of *f*. Here t_i is used to represent the literal of variable x_i; t_i can be either x_i or $\bar{x}_i$. A cube is a product of literals. A vertex is *covered* by a cube if the vertex is contained in the cube.

A *Boolean difference* of *f* with respect to a variable x_i, denoted $f^B_{x_i}$ is defined as $f(x_1, ..., x_i, ..., x_n) \oplus f(x_1, ..., \bar{x}_i, ..., x_n)$. It can be computed from the formula $f^B_{x_i} = f_{x_i} \oplus f_{\bar{x}_i}$.

Using the Shannon expansion and the identity $\bar{x}_i = 1 \oplus x_i$, *we can derive,*

$$f = x_i f^B_{x_i} \oplus f_{\bar{x}_i} \tag{2.1}$$

Decision Diagram (FDD) [Kebschull
heuristic algorithm that simultaneously generates both a

[illegible]imal polarity vector and the GRM form. To find the optimal polarity of a variable x_i, it is [illegible] decided which of the expression between (2.1) or (2.2) is to be chosen. The one which [illegible] fewer cubes in the final GRM form must be chosen. This process will continue recursively [illegible] each variable and an FDD can be formed.

[illegible]mental results show that the algorithm produces less cubes than compared to [Sarabi 1992, [illegible] 1992] and takes ½ second on an average for computation.

[illegible] A Genetic Algorithm for minimization of Fixed Polarity Reed-Muller [illegible]ssions [Drechsler 1995]

[illegible] Algorithms (GAs) are often used in optimization and machine learning [Davis 1991, [illegible]berg 1989]. One approach to minimize Fixed Polarity Reed-Muller expressions using GA [illegible] been outlined in [Drechsler 1995].

[illegible] is comprised of several steps. These include *representation of the problem in GA domain-* [illegible]*ation of the population, finding an object function or fitness function, setting selection* [illegible]. *working with various GA operators.* For representing FPRM expressions polarity vector [illegible] chosen for each variable. Thus, each element of the population corresponds to an *n-*[illegible]ional binary vector. A population is a set of vectors. Using this binary encoding each [illegible] represents a valid solution.

[illegible] *objective function* that measures the *fitness* of each element used here is the number of [illegible] in the FPRM corresponding to the chosen polarity. This function has to be minimized to [illegible] small two-level representation of the function.

[illegible] *selection* is performed by *roulette wheel selection.* Additionally *elitarism* is also used [Davis [illegible]]. This guarantees that the best element is never gets lost and thus faster convergency is [illegible].

[illegible] is helpful to combine GAs with problem specific heuristics [Davis 1991]. The resulting [illegible] are called *Hybrid GAs* (HGAs).

[illegible] GA operators used here are *reproduction, crossover, 2-time crossover, mutation, 2-time* [illegible] *mutation with neighbor.*

[illegible]posed GA works in the following steps:

[illegible] Initially a random population of binary finite strings is generated and i elements are [illegible]timized by the greedy heuristic as discussed above.

[illegible] The better half of the population is copied in each iteration without modification. Then [illegible] genetic operators, *reproduction* and *crossover* are applied to another $\frac{pop}{2}$ elements.

The elements are chosen according to their fitness as described above. The newly created elements are then mutated by one of the three mutation operators with a given probability.

iii. The algorithm stops if no improvement is obtained for $50 \cdot \log(best\_fitness)$ iterations, where *best_fitness* denotes the fitness of the best element in the population. Finally if $i > 0$ the greedy algorithm is applied to the best element.

The genetic operators are iteratively applied corresponding to their probabilities.

i. *Reproduction* is performed with a probability of 20%.

ii. *Crossover* and *2-time crossover* are performed with a probability of 80%.

iii. *Mutation*, *2-time mutation* and *mutation with neighbor* are carried out on the newly generated elements with a probability of 15%.

Experimental results show that for up to 15 variables the proposed HGA gives as many as product terms as the exact algorithms give but require much less CPU seconds. For functions with larger variables, where no optimal solution is known, the results were compared with [Drechsler 1994]. The number of product terms are much less than the heuristic approach.

The pure GA *i.e.* the GA without application of the greedy heuristic, performs not very good, since the starting points are too bad. This avoids a fast convergency.

2.2.4 Fast OFDD based minimization of Fixed Polarity Reed-Muller Expressions [Drechsler 1996]

In this paper a Fast OFDD (Ordered Functional Decision Diagrams) based minimization approach has been proposed.

Definition 2.2 A DD over $X_n := \{x_1, x_2, ..., x_n\}$ is a rooted directed acyclic graph $G = (V, E)$ with vertex set V containing two types of vertices, non-terminal and terminal vertices. A non-terminal vertex v is labeled with a variable from X_n, called the decision variable for v, and has exactly two successors denoted by $low(v), high(v) \in V$. A terminal vertex v is labeled with a 0 or 1 and has no successors.

Definition 2.3 A DD is free if each variable is encountered at most once on each path in the DD from the root to a terminal vertex. A DD is ordered if it is free and the variables are encountered in the same order on each path in the DD from the root to a terminal vertex.

It is possible to define certain reductions on the decision diagrams in order to reduce their size. Two reduction types are used in this paper [Drechsler 1996]:

Type I: Delete a node v' whose successors are identical to the successors of another node v and redirect the edges pointing to v' to point to v.

Type D: Delete all nodes v whose successor *high(v)* points to the terminal 0 and connect the incoming edges of the deleted node to the corresponding successor.

Definition 2.4 A DD is reduced if no reductions can be applied to the DD. Two DDs, G1 and G2, are called *equivalent* iff G2 results from G1 by repeated applications of reductions and inverse reductions. A DD, G2, is called the *reduction of* a DD, G1, if G1 and G2 are equivalent and G2 itself is reduced.

Definition 2.5 An OFDD over X_n is given by an ordered DD over X_n together with a uniquely determined decomposition type (here, the positive Davio (1.3) and the negative Davio (1.4) expansions are termed as decomposition types), $d_i \in \{pD, nD\}$ assigned to each variable x_i $(i \in \{1,\ldots,n\})$. The function $f_a : B^n \rightarrow B$ represented by an OFDD G over X_n is defined as:

i. If G consists of a single node labeled with 0 (1), then G is an OFDD for f = 0 (f = 1).

ii. If G has a root v with label x_i, then G is an OFDD for

$$\begin{cases} f_{low(v)} \oplus x_i f_{high(v)} : d_i \ is\ pD \\ f_{low(v)} \oplus \bar{x}_i f_{high(v)} : d_i \ is\ nD \end{cases}$$

where $f_{low(v)}(f_{high(v)})$ is the function represented by the OFDD rooted at $low(v)(high(v))$.

Definition 2.6 A node in an OFDD is called a positive Davio-node if it is expanded by Davio decomposition (2.1) and it is called a negative Davio-node if it is expanded by Davio decomposition (2.2).

Then the relation between OFDDs and FPRMs are outlined. This relation directly outlines methods for the construction of small or minimum FPRMs. In an OFDD a positive Davio or a negative Davio decomposition is carried out in each node. The reduction type D guarantees that a node is deleted, if the function represented at this node is independent from the corresponding variable. The paths from the root of the OFDD to the terminal one (1) are closely observed. They are called 1-paths. Each 1-path defines a subset of the variables X_n that uniquely corresponds to a 1-term in the FPRM. Thus, the following theorem is derived.

Theorem 2.6 The number of 1-paths in an OFDD for the Boolean function $f : B^n \rightarrow B$ is equal to the number of terms in the FPRM. The choice of decompositions in the OFDD determines the polarity of the FPRM.

If positive Davio decomposition is used, the variable in the FPRM is uncomplemented and it is complemented, if negative Davio decomposition is used. Obviously, the construction of the FPRM from a given OFDD for a single output function has running time $O(n.|terms|)$, where

$|terms|$ denotes the number of terms. The number of terms for the FPRM can easily be determined from the OFDD for a single output function by a simple depth-first-search algorithm that counts the number of 1-paths. The algorithm has running time $O(G)$. Thus, OFDDs with a minimum number of 1-paths to get FPRMs with a minimum number of terms is determined.

If an OFDD with fixed decompositions is given and the decomposition corresponding to one variable x_i is to be changed from positive Davio to negative Davio or vice versa this can be done efficiently in polynomial time as follows: An EXOR-operation is carried out at each node labeled with x_i.

To determine the minimum FPRM for a given function f an OFDD with only positive Davio-nodes is built up. Then it is transformed step by step to an OFDD with only negative Davio-nodes. OFDDs for each possible choice of decomposition types are constructed, *i.e.* 2^n OFDDs are constructed. For each OFDD the number of 1-paths is determined and the best result is stored.

To avoid the construction of the same OFDD two times, gray code is used to enumerate all decompositions. Although the algorithm is conceptually very simple, the experimental results show that it performs very fast. One main reason for this, are the efficient operations on the OFDDs.

For all functions for which the OFDD can be constructed the minimum FPRM can be obtained. This can be done for (some) functions with several hundred variables. Thus, this approach is mainly limited by the running time, but not by the space requirement.

2.2.5 A semicustom IC for generating optimum generalized Reed-Muller expansions [Almaini 1997]

The paper [Almaini 1997] explains the theory and design of a semicustom integrated circuit (IC) for the generation of the optimum polarity of a given Boolean function. Given the minterm coefficients of a Boolean function, the chip computes coefficients of all the fixed polarities of the generalized Reed-Muller (GRM) expansions, and identifies the polarity with the least number of terms.

The method for conversion between function coefficients and GRM coefficients is based on the Pascal triangle [Almaini 1996]. The X coefficients, where, $X \in \{0,1\}$, of the minterms are placed in a row and adjacent digits are EXORed to produce the below and so on. The resulting elements of each row are the coefficients for f_0. It was observed that the right-most elements of each row are the coefficients for f_{15}. The transformation may be reversed if the coefficients of f_0 and f_{15} are reversed.

Design of the ASIC for a 4-variable function, with sixteen input lines, includes CONV, the Pascal triangle converter, a four bit counter COUNT, ADD-which computes the weights of the vectors. Here, the weight refers to the number of logic ones, which is equal to the

number of terms. The weight is stored in REG while the coefficients of the polarity are stored in REGOUT. REG and REGOUT are 4 and 16 bit registers. These hold the weight of f_0 and its coefficients and update their content only if a better polarity is found. COMP compares the weights of the present polarity and last best polarity. Polarity vectors are computed on the positive edge of clock pulse. By the end of the sixteenth clock pulse REGOUT hold the coefficients of the best polarity.

2.2.6 Mapping of fixed polarity Reed-Muller coefficients from minterms and the minimization of fixed polarity Reed-Muller expressions [Khan 1997]

In the paper [Khan 1997], an efficient and simple algorithm for mapping FPRM coefficients from the on-set minterms of the function for a given polarity vector is presented. Another heuristic algorithm for finding an optimal polarity vector from the on-set minterms that produces the near minimum FPRM expression is also presented. Both these algorithms are developed for single-output fully specified functions.

For the purpose of mapping FPRM coefficients and the heuristic determination of an optimal polarity vector from the on-set minterms, the following definitions and lemmas are required.

Definition 2.7 Let x be a variable and $e \in \{0, 1, 2\}$. x^e is a literal of x such that

$$x^e \begin{cases} \bar{x} & if \quad e = 0 \\ x & if \quad e = 1 \\ 1 & if \quad e = 2 \end{cases}$$

The *canonical sum of products (CSOP)* for an arbitrary n-variable function $f(X)$ is represented as

$$f(X) = \sum_{k \in \{0,1\}^n} + a_k X^k = \sum_{k \in \{0,1\}^n} \oplus a_k X^k \tag{2.3}$$

where $\sum +$ and $\sum \oplus$ represent OR-Sum and EXOR-Sum respectively, $X = (x_1, x_2, ..., x_n)$ is the variable array, $k = (k_1\ k_2 ... k_n) \in \{0, 1\}^n$ is the polarity n-tuple for the minterm $X^k = (x_1^{k_1}\ x_2^{k_2} ... x_n^{k_n}) \in \{0, 1\}^n$ and $a_k \in \{0, 1\}$ are the CSOP coefficients, and $X^k = (x_1^{k_1}\ x_2^{k_2} ... x_n^{k_n})$ is the minterm corresponding to the coefficient a_k and exists iff $a_k = 1$.

Definition 2.8 Let $y = (y_1\ y_2 ... y_n) \in \{0, 1, 2\}^n$. The number of 1s in y is denoted by $\tau(y)$.

Definition 2.9 The value of Boolean difference of an arbitrary function of n-variables $f(X)$ with respect to X^h at $X = (0,0,...,0)$ is defined as

$$\left.\frac{d^{\tau(h)} f(X)}{dX^h}\right|_{(0,0,\ldots,0)} = \sum_{m \in M} \oplus f(Z)$$

$$h = (h_1\, h_2 \ldots h_n) \in \{1,2\}^n,\ dX^h = dx_1^{h_1} dx_2^{h_2} \ldots dx_n^{h_n},\ d^0 = d^1 = 1,$$

where $M = \left\{ m = (m_1 m_2 \ldots m_n) \middle| (\forall i) m_i = \begin{cases} 0, 1 & if\ h_i = 1 \\ 2 & if\ h_i = 2 \end{cases} \right\}$, $Z = (z_1\, z_2 \ldots z_n)$ *and*

$$(\forall i) z_i = \begin{cases} 0 & if\ m_i = 0 \\ 1 & if\ m_i = 1 \\ 2 & if\ m_i = 2 \end{cases}$$

Definition 2.10 $K_{on} = \{k = (k_1 k_2 \ldots k_n) \in \{0,1\}^n | a_k = 1\}$ is the set of polarity n-tuples for the ON-minterms of *f(X)*. $K_{off} = \{k = (k_1 k_2 \ldots k_n) \in \{0,1\}^n | a_k = 0\}$ is the set of polarity n-tuples for the OFF-minterms of *f(X)*.

Theorem 2.7 $b_{11\ldots1} = 1$ *if and only if* $|K_{on}|$ is odd.

Theorem 2.8 $(\forall t \in \{0, 1\}^n) b_t = \sum_{k \in K_{on}} \oplus d$ where $d^* = (d_1^* d_2^* \ldots d_n^*) = (t \vee k \vee p) \wedge (t \vee \bar{k} \vee \bar{p})$, $p = (p_1\, p_2 \ldots p_n) \in \{0, 1\}^n$ is the polarity vector for the variable array, and

$$d = \begin{cases} 1\ iff\ (\forall i) d_i^* = 1 \\ 0\ iff\ (\exists i) d_i^* = 0 \end{cases}$$

$\vee$, $\wedge$ *and* $\bar{\ }$ are bitwise operators.

Theorem 2.9 $b_{00\ldots0} = 1$ *iff* $\bar{p} \in K_{on}$.

Depending on Theorems 2.7-2.9, an algorithm is developed for mapping the FPRM coefficients from the on-set minterms.

In this algorithm, $|K_{on}|$ numbers of on-set minterms are to be stored, where average and maximum values of $|K_{on}|$ are,

$$\sum_{i=0}^{2^n} i\binom{2^n}{i} \Big/ \sum_{i=0}^{2^n}\binom{2^n}{i} = 2^{n-1}$$

2^n, respectively. Therefore, both the average and maximum space complexities of this algorithm is $O(2^n)$, where the coefficient of the average complexity is 0.5 times that of the maximum complexity. The average and maximum computational time complexities of the algorithm are $O(4^n)$ where the coefficient of the average complexity is 0.5 times that of the maximum complexity.

Definition 2.11 Let $K_{on,i} = \{k_i \mid k_i \text{ is the ith bit of } k \in \{0,1\}^n \text{ and } a_k = 1\}$ be the set of ith bits of on-set minterms. $ONE(i) = \sum_{k \in K_{on}} k_i$ is the number of 1s in $K_{on,i}$. $ZERO(i) = |K_{on}| - ONE(i)$ is the number of 0s in $K_{on,i}$.

The following observations were found during the experimentation of the algorithm.

Observation 2.1 If all product terms of the function are canonical product terms, i.e. minterms, then the optimal value of p_i is likely to be 1 if $|K_{on}|$ is even or both $|K_{on}|$ and $ONE(i)$ are odd; and the optimal value of p_i is likely to be 0 if $|K_{on}|$ is odd and $ONE(i)$ is even.

Example 2.1 Let $f(x_1, x_2, x_3) = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2x_3 + x_1\bar{x}_2x_3$. Here, the function contains four minterms, therefore the optimal values of p_1, p_2 and p_3 are likely to be 1.

Observation 2.2 If some product terms of the function are non-canonical product terms, then the optimal value of p_i is likely to be 1 if $ONE(i) \geq ZERO(i)$; and the optimal value of p_i is likely to be 0 if $ONE(i) < ZERO(i)$.

Example 2.2 Let $f(x_1, x_2, x_3) = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_3 + x_1x_3$. Here, $ONE(1) = 1$, $ZERO(1) = 2$, $ONE(3) = 2$, and $ZERO(3) = 1$. Therefore, the optimal value of p_1 is likely to be 0 and the optimal value of p_3 is likely to be 1.

Lemma 2.1 If a function $f(x_1, x_2, ..., x_n)$ contains only minterms and $ONE(i) = |K_{on}|$, then $f_1 \oplus x_i f_2$ (expansion using $p_i = 1$) contains the same set of minterms as the original function. Similarly, if a function $f(x_1, x_2, ..., x_n)$ contains only minterms and $ONE(i) = 0$, then $f_0 \oplus \bar{x}_i f_2$ (expansion using $p_i = 0$) contains the same set of minterms as the original function.

Depending on Observations 2.1 and 2.2 and Lemma 2.1, an algorithm is developed for heuristically finding an optimal polarity vector from on-set minterms that produces the near optimal FPRM expressions. The maximum and the average computational time of this algorithm is $O(n2^n)$.

2.2.7 Sympathy: Fast Exact Minimization of Fixed Polarity Reed-Muller Expressions for Symmetric Functions [Drechsler 1997]

Here an exact algorithm, *Sympathy* has been implemented, to minimize FPRMs for symmetric functions and the required computation time is polynomial. In [Drechsler 1996] close relations between OFDDs and FPRMs have been investigated.

The definition of OFDD and related terms are described in the previous section.

Theorem 2.10 Let G_1, G_2 be OFDDs with the same Decomposition Type List (DTL) d and with the same ordering. Then the EXOR-synthesis of G_1 and G_2 can be performed by an algorithm of complexity $O(|G_1| \cdot |G_2|)$ resulting in an OFDD bounded by the same size.

Symmetric functions are used here. Let $f : B^n \to B$ be a totally defined Boolean function and $X_n := \{x_1, x_2, ..., x_n\}$ be the corresponding set of variables. The function f is said to be symmetric with respect to a set $S \subseteq X_n$ if f remains invariant under all permutations of the variables in S. For completely specified functions the symmetry is an equivalence relation which partitions the set X_n into disjoint classes $S_1, ..., S_k$ that will be named the symmetry sets. A function f is called partially symmetric if it has at least one symmetry set S_i with $S_i > 1$. If a function f has only one symmetry set $S = X_n$, then f is called totally symmetric. For example, $f = \bar{x}_1 \bar{x}_2 ... \bar{x}_n$ is a totally symmetric function. If $x_i, x_j \in S_l \subseteq X_n$ $x_i \neq x_j$ and $1 \leq l \leq k$ f is called *pairwise symmetric* in (x_i, x_j). A simple consequence of pairwise symmetry is the following lemma.

Lemma 2.2 A function is pairwise symmetric in (x_i, x_j) iff $f_{x_i \bar{x}_j} = f_{\bar{x}_i x_j}$

In the following paragraphs the problem of finding minimal FPRMs for totally symmetric functions is considered. The results will also be applied to partially symmetric functions.

It is showed below that the number of different FPRMs that have to be considered during minimization of FPRMs for symmetric functions can be tremendously reduced.

Theorem 2.11 Let f be pair-wise symmetric in (x_i, x_j). For Decomposition Type Lists (DTLs) $d = (d_1 ... d_i ... \bar{d}_j .. d_n)$ and $d' = (d_1 ... \bar{d}_i ... d_j .. d_n)$ it holds $|f_d| = |f_{d'}|$.

Lemma 2.2 a straightforward computation shows that it does not influence the number of whether f is first decomposed by pD for x_i and then by nD for x_j or first by nD for x_i then by pD for x_j. From the theorem it is obtained:

Corollary 2.1 There exist at most n+1 FPRMs for a totally symmetric function that differ in size.

The Corollary found that it is sufficient in the following to only consider FPRMs for the set of $n+1$ DTLs: $D = \{d^i \mid d^i = (nd)^i (pd)^{n-i} \wedge 0 \le i \le n\}$.

OFDDs are used for the construction of the FPRMs and a polynomial algorithm is aimed minimization, it is to be proved that an OFDD for a totally symmetric function has at polynomial size in the number of n variables.

the following, given is an upper bound for the size of the OFDDs for totally symmetric ns with DTLs as they will occur in the exact algorithm.

Theorem 2.12 Each OFDD with DTL $d^i \in D(i \in \{0,...,n\})$ that represents a totally symmetric function f has size $O(n^3)$.

the size of OFDDs with only positive Davio-nodes is $O(n^2)$. (The same argumentation for OFDDs with only negative Davio-nodes.) Then the case where the upper variables are posed by negative Davio-nodes are considered, while the lower variables are decomposed positive Davio-nodes. The assertion of the theorem then follows from the fact that the lowest decomposed by negative Davio-nodes has at most n nodes and that the functions f_{x_k} and $f_{\bar{x}_k} = f_{\bar{x}_k}$ of a totally symmetric function f are totally symmetric.

Theorem 2.13 The transformation of an OFDD with DTL $d^i \in D(i \in \{0,...,n-1\})$ of a totally symmetric function f of n variables to an OFDD with DTL $d^{i+1} \in D$ has time and space complexity $O(n^6)$.

Theorem 2.14 The exact algorithm for the FPRM minimization of a totally symmetric function f of n variables has running time $O(n^7)$ and space requirement $O(n^6)$.

Proof: The running time of the algorithm is dominated by the transformation (from Theorem 2.13). Thus, the overall performance of the algorithm is directly obtained, since the transformation has to be carried out n times. The space requirement is $O(n^6)$, since this is the worst case during the transformation. (The resulting OFDDs have at most size $O(n^3)$.)

ough the algorithm can only be estimated by a polynomial of high degree (due to the worst behavior of the EXOR-operation) all experiments have shown that the algorithm is very fast shows linear behavior with respect to the running time.

2.8 Exact minimization of Fixed Polarity Reed-Muller expressions for completely Specified Functions [Debnath 2000]

e paper [Debnath 2000], the operators '+' and '⊕' indicate arithmetic and mod-2 addition, ctively.

nition 2.12 An n-variable switching function f is a mapping $f:\{0,1\}^n \rightarrow \{0,1\}$ and an n-ble integer valued function g is a mapping $g:\{0,1\}^n \rightarrow \{0,1,...,p-1\}$ where $p<2$. uld be noted that switching functions are a subset of integer-valued functions.

nition 2.13 An n-variable integer-valued function $f(x_1,x_2,...,x_n)$ can be written as $\sum_{j=0}^{2^n-1} m_j x_1^{b_1} x_2^{b_2} ... x_n^{b_n}$ where $m_j \in \{0,1,...,p-1\}(p \geq 2), b_1,b_2,...,b_n \in \{0,1\}$ such that $b_1 b_2 ... b_n$ is n-bit binary number representing j, $x_i^{b_i} = \bar{x}_i$ when $b_i = 0$, $x_i^{b_i} = x_i$ when $=1$, *and* $i = 1,2,...,n$. Then $\lfloor m_0, m_1, ..., m_{2^n-1} \rfloor$ is the truth vector of f.

mple 2.3 The truth vector of the three-variable switching function $\bar{x}_1\bar{x}_2\bar{x}_3 \vee x_1$ is ,0,1,1,1,1], and that of the three-variable integer-valued function $3x_1 + 4x_2x_3 + 2\bar{x}_3$ is 2,4,5,3,5,7].

an n-variable completely specified switching function there are 2^n distinct FPRMs, and the mization problem is to find a polarity vector that produces an FPRM with minimum number ducts. On the other hand, for an n-variable incompletely specified switching function with specified minterms there are $2^{n+\alpha}$ distinct FPRMs, and the minimization problem is to find arity vector and an assignment of the unspecified minterms to 0's and 1's that produce an RM with minimum number of products. Once the polarity vector and the assignment of the cified minterms are determined, generation of an FPRM is relatively easy [Davio 1978, 1996].

thod for the exact minimization of FPRMs for three-variable switching function has been sed in this paper. The method is based on the computation of *extended truth vector* and *vector* [Davio 1978, Sasao 1996]. In general, for an n-variable completely specified hing function, extended truth vector is a binary vector $\lfloor t_0, t_1, ..., t_{3^n-1} \rfloor$ with 3^n elements, and t vector is an integer vector $\lfloor w_0, w_1, ..., w_{2^n-1} \rfloor$ with $\lfloor t_0, t_1, ..., t_{3^n-1} \rfloor$ elements. Each element e weight vector is associated with a *polarity vector*.

For an n-variable switching function f, polarity vector for w_j is a binary vector $(b_1, b_2, ..., b_n)$ such that $b_1, b_2, ..., b_n$ the n-bit binary number representing j, $(j = 0,1,...,2^n - 1)$, and w_j represents the number of products in the FPRM for f with polarity vector $(b_1, b_2, ..., b_n)$.

For an n-variable switching function with α unspecified minterms $d_1, d_2, ..., d_\alpha$, extended truth vector is a vector of switching functions $t_i(d_1, d_2, ..., d_\alpha)(i = 0,1,...,3^n - 1)$, and weight vector is a vector of integer-valued functions $w_j(d_1, d_2, ..., d_\alpha)(j = 0,1,...,2^n - 1)$.

Definition 2.14 Let the *minimum value* of the α-variable integer-valued function $w(d_1, d_2, ..., d_\alpha)$, denoted by $w^{\min}$, be $\min_{0 \le i \le 2^n - 1} m_i$, where $\lfloor m_0, m_1, ..., m_{2^\alpha - 1} \rfloor$ represents the truth vector for w.

Let $\lfloor w_0, w_1, ..., w_{2^n - 1} \rfloor$ be the weight vector for an n-variable incompletely specified switching function $f(x_1, x_2, ..., x_n)$, and $w_j^{\min}$ be the minimum value for $w_j(d_1, d_2, ..., d_\alpha)$ where $d_1, d_2, ..., d_\alpha$ represent unspecified minterms of f. Let $0 \le k \le 2^n - 1$ and $a_1, a_2, ..., a_\alpha \in \{0,1\}$ such that $w_k(a_1, a_2, ..., a_\alpha) = \min_{0 \le i \le 2^n - 1} w_j^{\min}$. Let $c_1, c_2, ..., c_n$ be the n-bit binary number representing k. Then, $(a_1, a_2, ..., a_\alpha)$ represents an assignment of $(d_1, d_2, ..., d_\alpha)$ and $(c_1, c_2, ..., c_n)$ represents a polarity vector that produces a minimum FPRM for f.

To manipulate integer-valued function a multi-terminal binary decision diagram (MTBDD) [Clarke 1993] is used. An MTBDD, which is a natural extension of binary decision diagram (BDD) [Bryant 1986], is a directed acyclic graph with multiple terminal nodes each of which has an integer value.

A straightforward method to build MTBDDs for weight vector requires excessive computation time and memory resources, because they represent all possible FPRMs for the given incompletely specified function. However, the concentration is in an FPRM with the fewest products. Suppose there is an FPRM for the given function with $t_{threshold+1}$ products, then it is sufficient to search for an FPRM with $t_{threshold}$ or fewer products. If such an FPRM does not exist then the FPRM with $t_{threshold+1}$ products is the minimum FPRM. Thus, to restrict the search space without sacrificing the minimality of the solution, we use *threshold value*, $t_{threshold}$, during construction of MTBDDs. The threshold value can be obtained by using any simplification program for FPRMs.

Based on the above discussions, an algorithm for exact minimization of FPRM for incompletely specified n-variable switching function f is developed.

After implementation of the developed algorithm it is revealed that the factors on which the computation time mainly depends are the threshold value, the number of variables in the function, and the number of unspecified minterms. The implementation results shown in this

er proves that the algorithm works favorably for many functions with eight or fewer variables with any number of unspecified minterms. However, for functions with nine or more ables it often requires excessive CPU time and memory resources when the number of specified minterms is more than 30.

Chapter 3
Introduction to Application Specific Integrated Circuits (ASICs)

Introduction

ASIC is an *application-specific integrated circuit*. Before knowing what an ASIC is let us look at the evolution of the silicon chip or integrated circuit (IC). [Smith 1997]

Figure 3.1(a) shows an IC package (this is a pin-grid array, or PGA, shown upside down; the pins will go through holes in a printed-circuit board). People often call the package a chip, but Figure 3.1(b) shows that the silicon chip itself (more properly called a die) is mounted in the cavity under the sealed lid. A PGA package is usually made from a ceramic material, but plastic packages are also common.

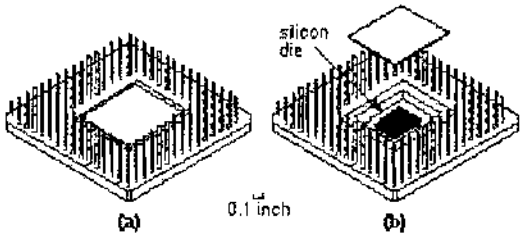


Figure 3.1: An integrated circuit (IC). (a) A pin-grid array (PGA) package. (b) The silicon die or chip is under the package lid.

Examples of ICs that are *not* ASICs include standard parts such as: memory chips sold as a commodity item—*read only memory (ROMs)*, *dynamic random-access memory (DRAM)*, and *static RAM (SRAM)*; microprocessors; transistor-transistor logic (TTL) or TTL-equivalent ICs at small-scale integration (SSI), medium-scale integration (MSI), and large-scale integration (LSI) levels.

Examples of ICs that *are* ASICs include: a chip for a toy bear that talks; a chip for a satellite; a chip designed to handle the interface between memory and a microprocessor for a workstation CPU; and a chip containing a microprocessor as a cell together with other logic. Two ICs that might or might not be considered ASICs are a controller chip for a PC and a chip for a modem. Both of these examples are specific to an application (shades of an ASIC) but are sold to many different system vendors (shades of a standard part). ASICs such as these are sometimes called *application-specific standard products (ASSPs)*.

3.2 Types of ASICs

ICs are made on a thin (a few hundred microns (a micron is 10^{-6} m) thick), circular silicon wafer, with each wafer holding hundreds of die (sometimes people use dies or dice for the plural of die). The transistors and wiring are made from many layers (usually between 10 and 15 distinct layers) built on top of one another. Each successive *mask layer* has a pattern that is defined using a *mask* similar to a glass photographic slide. The first half-dozen or so layers define the transistors. The last half-dozen or so layers define the metal wires between the transistors (the *interconnect*).

The different types of ASICs discussed below are,

- Full Custom ASICs
- Semicustom ASICs
- Programmable ASICs

3.2.1 Full Custom ASICs

In a *full-custom ASIC* an engineer designs some or all of the logic cells, circuits, or layout specifically for one ASIC. This means the designer abandons the approach of using pretested and precharacterized cells for all or part of that design. It makes sense to take this approach only if there are no suitable existing cell libraries available that can be used for the entire design. This might be because existing cell libraries are not fast enough, or the logic cells are not small enough or consume too much power. A full-custom design may be needed if the ASIC technology is new or so specialized that there are no existing cell libraries or because the ASIC is so specialized that some circuits must be custom designed.

3.2.2 Semicustom ASICs

Semicustom ASICs are,

- Standard Cell based ASICs
- Gate Array based ASICs

3.2.2.1 Standard-Cell–Based ASICs

A *cell-based ASIC* (*cell-based IC, or CBIC*) uses predesigned logic cells (AND gates, OR gates, multiplexers, and flip-flops, for example) known as *standard cells*. It is generally accepted that a cell-based ASIC or CBIC means a standard-cell–based ASIC.

The standard-cell areas (also called flexible blocks) in a CBIC are built of rows of standard cells—like a wall built of bricks. The standard-cell areas may be used in combination with larger predesigned cells, perhaps microcontrollers or even microprocessors, known as *megacells*.

Megacells are also called megafunctions, full-custom blocks, system-level macros (SLMs), fixed blocks, cores, or Functional Standard Blocks (FSBs).

The ASIC designer defines only the placement of the standard cells and the interconnect in a CBIC. However, the standard cells can be placed anywhere on the silicon; this means that all the mask layers of a CBIC are customized and are unique to a particular customer. The advantage of CBICs is that designers save time, money, and reduce risk by using a predesigned, pretested, and precharacterized *standard-cell library*. In addition each standard cell can be optimized individually. During the design of the cell library each and every transistor in every standard cell can be chosen to maximize speed or minimize area, for example. The disadvantages are the time or expense of designing or buying the standard-cell library and the time needed to fabricate all layers of the ASIC for each new design. Figure 3.2 shows a CBIC.

The important features of this type of ASIC are as follows:

i. All mask layers are customized—transistors and interconnect.

ii. Custom blocks can be embedded.

iii. Manufacturing lead time is about eight weeks.

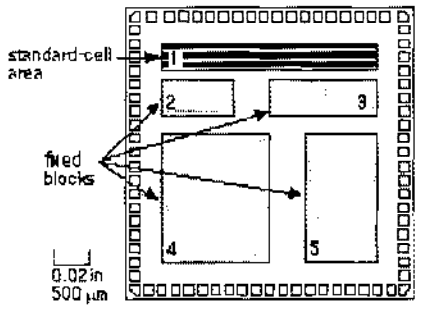


Figure 3.2: A cell-based ASIC (CBIC) die with a single standard-cell area (a flexible block) together with four fixed blocks. The flexible block contains rows of standard cells. This is the low-powered microscopic look of the die of Figure 3.1(b). The small squares around the edge of the die are bonding pads that are connected to the pins of the ASIC package.

Each standard cell in the library is constructed using full-custom design methods, but these predesigned and precharacterized circuits can be used without having to do any full-custom design. This design style gives the same performance and flexibility advantages of a full-custom ASIC but reduces design time and reduces risk.

Standard cells are designed to fit together like bricks in a wall. Figure 3.3 shows an example of a simple standard cell. Power and ground buses (VDD and GND or VSS) run horizontally on metal lines inside the cells.

Standard-cell design allows the automation of the process of assembling an ASIC. Groups of standard cells fit horizontally together to form rows. The rows stack vertically to form flexible

rectangular blocks. Then a *flexible block* built from several rows of standard cells is connected to other standard-cell blocks or other full-custom logic blocks.

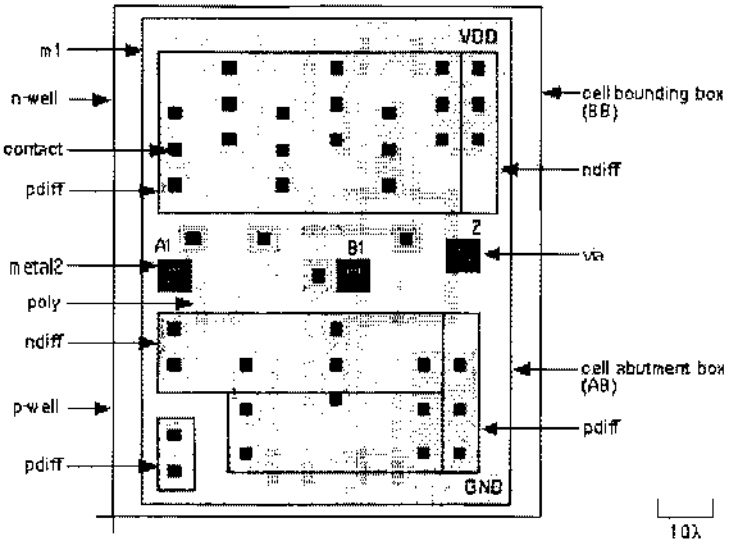


Figure 3.3: Looking down on the layout of a standard cell. This cell would be approximately 25 microns wide on an ASIC with l (lambda) = 0.25 microns. Standard cells are stacked like bricks in a wall; the abutment box (AB) defines the "edges" of the brick. The difference between the bounding box (BB) and the AB is the area of overlap between the bricks. Power supplies (labeled VDD and GND) run horizontally inside a standard cell on a metal layer that lies above the transistor layers. Each different shaded and labeled pattern represents a different layer. This standard cell has center connectors (the three squares, labeled A1, B1, and Z) that allow the cell to connect to others.

Both cell-based and gate-array ASICs use predefined cells, but there is a difference—the transistor sizes in a standard cell can be changed to optimize speed and performance, but the device sizes in a gate array are fixed. This results in a trade-off in performance and area in a gate array at the silicon level. The trade-off between area and performance is made at the library level for a standard-cell ASIC.

3.2.2.2 Gate Array based ASICs

In a *gate array* (sometimes abbreviated to GA) or gate-array–based ASIC the transistors are predefined on the silicon wafer. The predefined pattern of transistors on a gate array is the *base array*, and the smallest element that is replicated to make the base array is the *base cell* (sometimes called a *primitive cell*). Only the top few layers of metal, which define the interconnect between transistors, are defined by the designer using custom masks. To distinguish this type of gate array from other types of gate array, it is often called a *masked gate array* (MGA). The designer chooses from a gate-array library of predesigned and precharacterized logic cells. The logic cells in a gate-array library are often called *macros*. The reason for this is that the base-cell layout is the same for each logic cell, and only the interconnect (inside cells

and between cells) is customized, so that there is a similarity between gate-array macros and a software macro. There are the following different types of MGA or gate-array–based ASICs:

i. Channeled gate arrays.

ii. Channelless gate arrays.

iii. Structured gate arrays.

There are two common ways of arranging (or arraying) the transistors on a MGA: in a channeled gate array space is left between the rows of transistors for wiring; the routing on a channelless gate array uses rows of unused transistors. The channeled gate array was the first to be developed, but the channelless gate-array architecture is now more widely used. A structured (or embedded) gate array can be either channeled or channelless but it includes (or embeds) a custom block.

3.2.2.2.1 Channeled Gate Array

Figure 3.4 shows a *channeled gate array*. The important features of this type of MGA are:

i. Only the interconnect is customized.

ii. The interconnect uses predefined spaces between rows of base cells.

iii. Manufacturing lead time is between two days and two weeks.

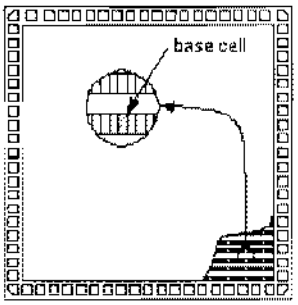


Figure 3.4: A channeled gate-array die. The spaces between rows of the base cells are set aside for interconnect.

A channeled gate array is similar to a CBIC—both use rows of cells separated by channels used for interconnect. One difference is that the space for interconnect between rows of cells are fixed in height in a channeled gate array, whereas the space between rows of cells may be adjusted in a CBIC.

3.2.2.2.2 Channelless Gate Array

Figure 3.5 shows a *channelless gate array* (also known as a *channel-free gate array*, *sea-of-gates array*, or *SOG* array). The important features of this type of MGA are as follows:

i. Only some (the top few) mask layers are customized– the interconnect.

ii. Manufacturing lead time is between two days and two weeks.

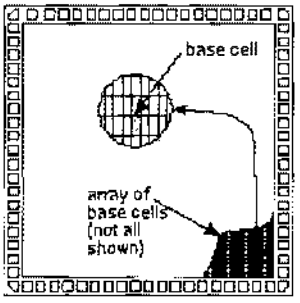


Figure 3.5: A channelless gate-array or sea-of-gates (SOG) array die. The core area of the die is completely filled with an array of base cells (the base array).

The key difference between a channelless gate array and channeled gate array is that there are no predefined areas set aside for routing between cells on a channelless gate array. Instead routing is done over the top of the gate-array devices. When an area of transistors is used for routing in a channelless array, no contact is made to the devices lying underneath; the transistors are left unused.

The logic density– the amount of logic that can be implemented in a given silicon area– is higher for channelless gate arrays than for channeled gate arrays. This is usually attributed to the difference in structure between the two types of array. In fact, the difference occurs because the contact mask is customized in a channelless gate array, but is not usually customized in a channeled gate array. This leads to denser cells in the channelless architectures. Customizing the contact layer in a channelless gate array allows increasing the density of gate-array cells because we can route over the top of unused contact sites.

3.2.2.2.3 Structured Gate Array

An *embedded gate array* or *structured gate array* (also known as *masterslice* or *masterimage*) combines some of the features of CBICs and MGAs. One of the disadvantages of the MGA is the fixed gate-array base cell. This makes the implementation of memory, for example, difficult and inefficient. In an embedded gate array some of the IC area is set aside and dedicated to a specific function. This embedded area either can contain a different base cell that is more suitable for building memory cells, or it can contain a complete circuit block, such as a microcontroller.

Figure 3.6 shows an embedded gate array. The important features of this type of MGA are the following:

i. Only the interconnect is customized.

ii. Custom blocks (the same for each design) can be embedded.

iii. Manufacturing lead time is between two days and two weeks.

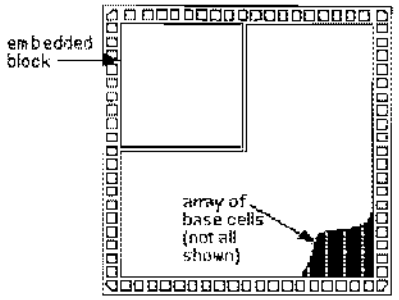


Figure 3.6: A structured or embedded gate-array die showing an embedded block in the upper left corner (a static random-access memory, for example). The rest of the die is filled with an array of base cells.

An embedded gate array gives the improved area efficiency and increased performance of a CBIC but with the lower cost and faster turnaround of an MGA. One disadvantage of an embedded gate array is that the embedded function is fixed. For example, if an embedded gate array contains an area set aside for a 32 k-bit memory, but only a 16 k-bit memory is needed, then half of the embedded memory function is wasted. However, this may still be more efficient and cheaper than implementing a 32 k-bit memory using macros on a SOG array.

ASIC vendors may offer several embedded gate array structures containing different memory types and sizes as well as a variety of embedded functions. ASIC companies wishing to offer a wide range of embedded functions must ensure that enough customers use each different embedded gate array to give the cost advantages over a custom gate array or CBIC.

3.2.3 Programmable ASICs

The programmable ASICs are,

i. Programmable Logic Devices (PLDs)

ii. Field Programmable Gate Arrays (FPGAs)

3.2.3.1 Programmable Logic Devices

Programmable logic devices (PLDs) are standard ICs that are available in standard configurations from a catalog of parts and are sold in very high volume to many different customers. However, PLDs may be configured or programmed to create a part customized to a specific application, and so they also belong to the family of ASICs. PLDs use different technologies to allow programming of the device. Figure 3.7 shows a PLD and the following important features that all PLDs have in common.

i. No customized mask layers or logic cells

ii. Fast design turnaround

iii. A single large block of programmable interconnect

iv. A matrix of logic macrocells that usually consist of programmable array logic followed by a flip-flop or latch

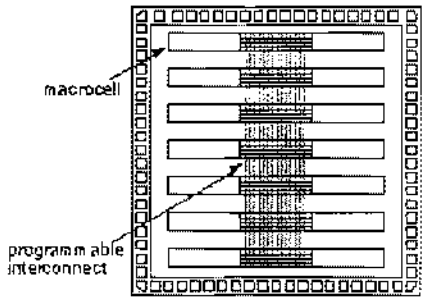


Figure 3.7 A programmable logic device (PLD) die. The macrocells typically consist of programmable array logic followed by a flip-flop or latch. The macrocells are connected using a large programmable interconnect block.

The simplest type of programmable IC is a *read-only memory (ROM)*. The most common types of ROM use a metal fuse that can be blown permanently (a *programmable ROM* or *PROM*). An *electrically programmable ROM*, or *EPROM*, uses programmable MOS transistors whose characteristics are altered by applying a high voltage. An EPROM can be erased either by using another high voltage (an *electrically erasable PROM*, or *EEPROM*) or by exposing the device to ultraviolet light (*UV-erasable PROM*, or *UVPROM*).

There is another type of ROM that can be placed on any ASIC—a *mask-programmable ROM* (*mask-programmed ROM* or *masked ROM*). A masked ROM is a regular array of transistors permanently programmed using custom mask patterns. An embedded masked ROM is thus a large, specialized, logic cell.

The same programmable technologies used to make ROMs can be applied to more flexible logic structures. By using the programmable devices in a large array of AND gates and an array of OR gates, a family of flexible and programmable logic devices called *logic arrays* are created. The *Programmable Array Logic* device produced first can be used, for example, as transition decoders for state machines. A PAL can also include registers (flip-flops) to store the current state information so a PAL can be used to make a complete state machine.

Just as a mask-programmable ROM, a logic array can be placed as a cell on a custom ASIC. This type of logic array is called a *programmable logic array* (PLA). There is a difference between a PAL and a PLA: a PLA has a programmable AND logic array, or *AND plane*, followed by a programmable OR logic array, or *OR plane*; a PAL has a programmable AND plane and, in contrast to a PLA, a fixed OR plane.

Depending on how the PLD is programmed, there is an *erasable PLD* (EPLD), or *mask-programmed PLD* (sometimes called a masked PLD but usually just PLD).

3.2.3.1 Field-Programmable Gate Arrays

A step above the PLD in complexity is the *field-programmable gate array (FPGA)*. There is very little difference between an FPGA and a PLD—an FPGA is usually just larger and more complex than a PLD. In fact, some companies that manufacture programmable ASICs call their products FPGAs and some call them *complex PLDs*. FPGAs are the newest member of the ASIC family and are rapidly growing in importance, replacing TTL in microelectronic systems. Even though an FPGA is a type of gate array, we do not consider the term gate-array–based ASICs to include FPGAs.

Figure 3.8 illustrates the essential characteristics of an FPGA:

i. None of the mask layers are customized.

ii. A method for programming the basic logic cells and the interconnect.

iii. The core is a regular array of programmable basic logic cells that can implement combinational as well as sequential logic (flip-flops).

iv. A matrix of programmable interconnect surrounds the basic logic cells.

v. Programmable I/O cells surround the core.

vi. Design turnaround is a few hours.

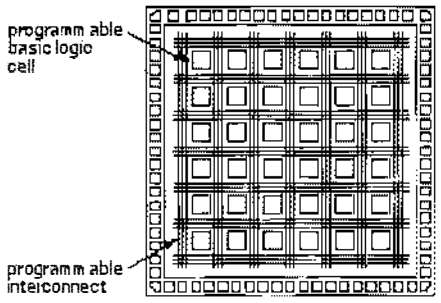


Figure 3.8: A field-programmable gate array (FPGA) die. All FPGAs contain a regular structure of programmable basic logic cells surrounded by programmable interconnect. The exact type, size, and number of the programmable basic logic cells varies tremendously.

3.3 Design Flow

Figure 3.9 shows the sequence of steps to design an ASIC, a *design flow*. The steps are listed below (numbered to correspond to the labels in Figure 3.9) with a brief description of the function of each step.

i. *Design entry:* The design is entered into an ASIC design system, either using a *hardware description language (HDL)* or *schematic entry*.

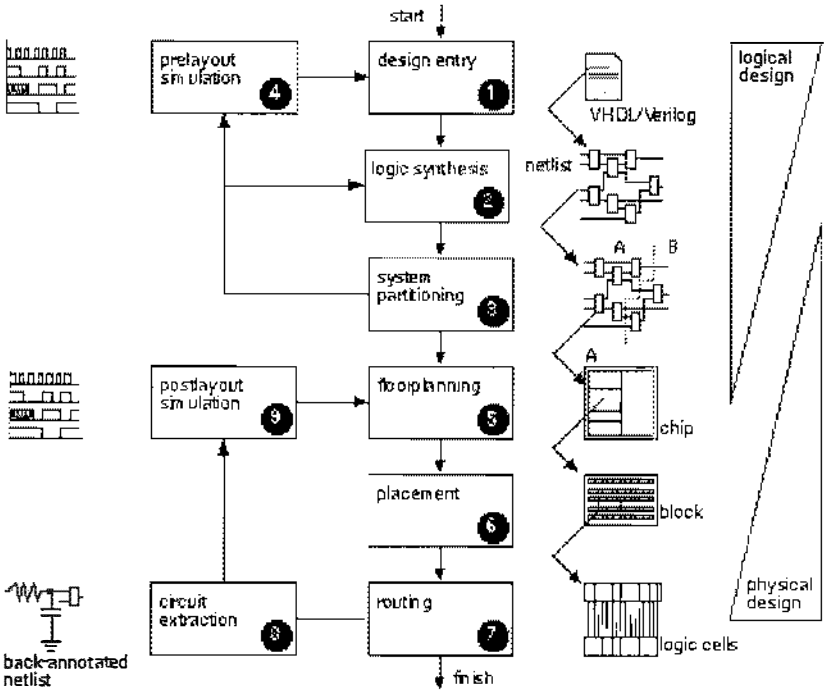


Figure 3.9: ASIC design flow.

ii. *Logic synthesis:* An HDL (VHDL or Verilog) and a logic synthesis tool is used to produce a *netlist* —a description of the logic cells and their connections.

iii. *System partitioning:* A large system is divided into ASIC-sized pieces.

iv. *Prelayout simulation:* The design functions correctness is checked.

v. *Floorplanning:* The blocks of the netlist are arranged on the chip.

vi. *Placement:* The locations of cells in a block are decided.

vii. *Routing:* The connections between cells and blocks are made.

viii. *Extraction:* The resistance and capacitance of the interconnect are determined.

ix. *Postlayout simulation:* Check to see the design still works with the added loads of the interconnect.

Steps 1–4 are part of *logical design*, and steps 5–9 are part of *physical design*. There is some overlap. For example, system partitioning might be considered as either logical or physical design. To put it another way, when system partitioning is performed both logical and physical factors has to be considered.

Chapter 4
RTL level Architecture of the developed ASIC for FPRM Minimization

4.1 Basics of Register Transfer Logic

A digital system is a sequential logic circuit constructed with flip-flops and gates. Normally, sequential circuits are specified with state tables. Specifying a large digital system with a state table is very difficult, because the number of states would be very large. To overcome this difficulty digital systems are designed using a modular approach. The system is partitioned into modular subsystems, each of which performs some functional tasks. The modules are constructed from digital devices like registers, decoders, multiplexers, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital system. These modules are described by a set of registers and the operations performed on the information stored in them. The information flow and processing performed on the data stored in the registers are referred to as register transfer operations. A digital system designed around registers is referred to as the *register transfer logic* (RTL).

A register transfer logic (RTL) level digital system is specified by the following three components:

i. A set of registers in the system.

ii. The operations performed on the data stored in the registers.

iii. The control for supervising the sequence of operations in the system.

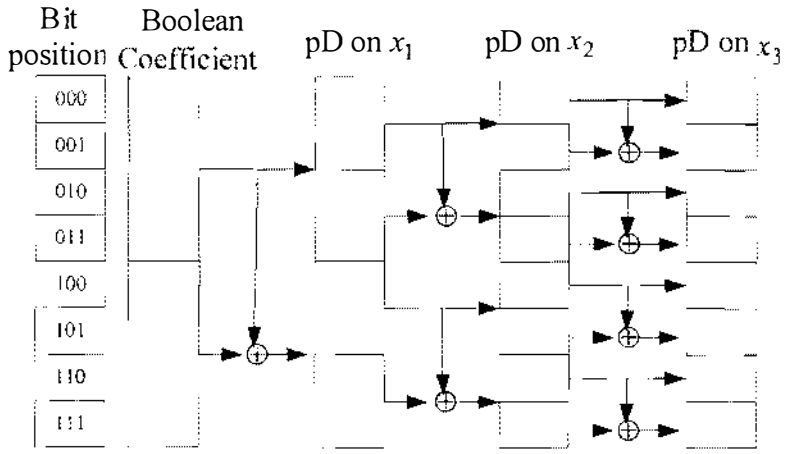
4.2 Computation of the positive and negative Davio expansions

The positive and negative Davio expansions are as below,

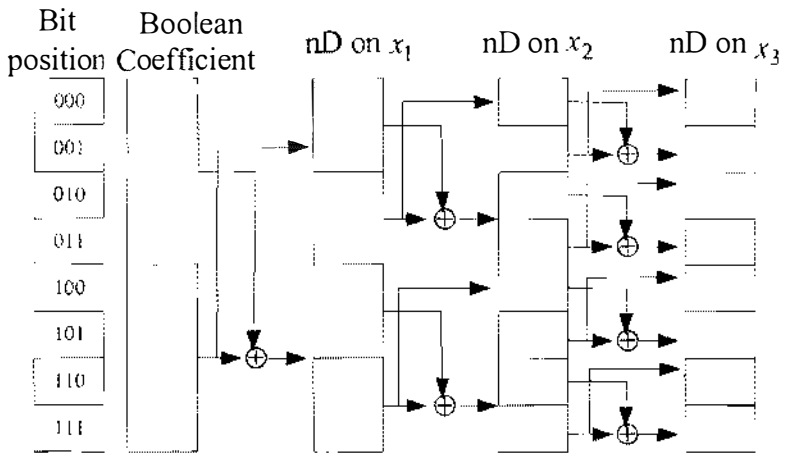
$$f(x_1, x_2, \ldots, x_n) = f_0 \oplus x_i f_2 \text{ (pD)}$$
$$f(x_1, x_2, \ldots, x_n) = f_1 \oplus \overline{x_i} f_2 \text{ (nD)}$$

where, $f_0 = f(x_1, \ldots x_{i-1}, 0, x_{i+1}, \ldots, x_n)$, $f_1 = f(x_1, \ldots x_{i-1}, 1, x_{i+1}, \ldots, x_n)$ and $f_2 = f_0 \oplus f_1$.

These expansions can be computed as shown in Figure 4.1 for a 3-variable function.



(a) Application of pD expansion



(b) Application of nD expansion

Figure 4.1: Computation of pD and nD expansions for a 3-variable function

Figure 4.1 the computation of the expansions on each variable for both positive and negative io expansions are shown. Figure 4.1 (a) shows the computation of the cofactors when the pD ansion is used for each variable x_1, x_2 and x_3. Figure 4.1 (b) shows the computation of ctors when the nD expansion is used for each variable x_1, x_2 and x_3. In this work the nsions are used on the variables in the similar approach shown in Figure 4.1 depending on polarity of the variables. If the polarity is 0 then pD expansion is used and if the polarity is 1 the nD expansion is used.

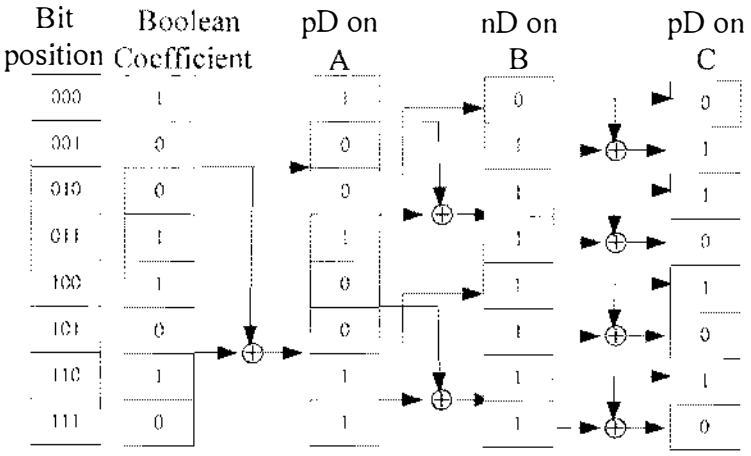


Figure 4.2: The transformation on variables A, B and C

example, for a function $f(A, B, C)$ the input vector $b = [1, 0, 0, 1, 1, 0, 1, 0]^T$ and polarity r. $cp = [0, 1, 0]$ then the transformations on each variable are shown in Figure 4.2. Here, the ity of variable A is 0, B is 1 and C is 0, respectively. So the transformation on A and C is using pD expansion and the transformation on B is done using nD expansion. The last n gives the FPRM coefficients of the given function for polarity vector [0, 1, 0].

RTL design of the developed ASIC for FPRM minimization

RTL design of the developed ASIC for minimizing FPRM expressions is shown in Figure

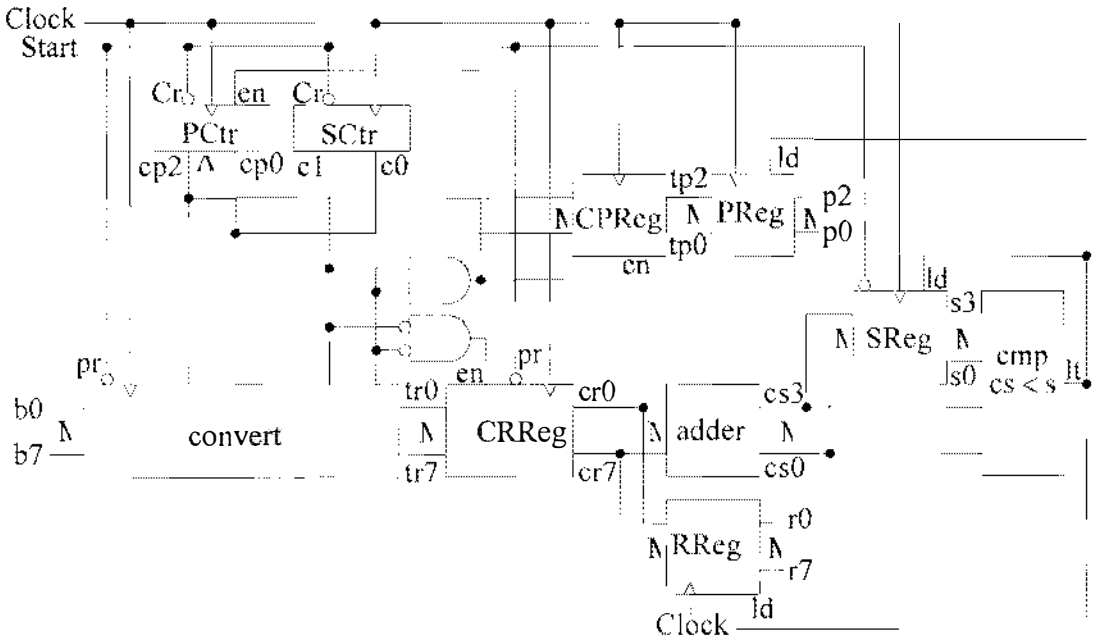


Figure 4.3: The block diagram of the developed ASIC to minimize FPRM expressions

Figure 4.3 is a block diagram minimizes FPRM expressions for 3-variable finactions. The eight input coefficients $b_0 .. b_7$ are applied to the inputs of the FPRM converter *convert*. Figure 4.4 shows the block diagram of the converter. The converter consists of $n.2^{n-1}$ 2-to-1 multiplexers and the same number of EXOR gates where n is the number of variables in the function. The converter also has 2^n number of (n+1)-to-1 multiplexers. Here, the polarity vector of an FPRM expression works as address lines of the 2-to-1 multiplexers. And thus provide either pD or nD expansion on each variable. Here, if the value of polarity cp_i $where\, 0 \le i \le n-1$ is 0 then pD expansion is used and if the value of polarity cp_i $where\, 0 \le i \le n-1$ is 1 then nD expansion is used.

As the address lines to the 2-to-1 multiplexers are the polarity vector cp_i, the value of cp_i multiplexes the input lines to the outputs of the converter. For the first variable, if $cp_2 = 0$ then the 2^{n-1} inputs go directly to the converter output register *tr* $(0 \ldots 2^{n-1}-1)$ and the EXOR of the, rest 2^{n-1} inputs and the first 2^{n-1} inputs, go to *tr*$(2^{n-1} \ldots 2^n-1)$. If $cp_2=1$, then the EXOR of the first 2^{n-1} and last 2^{n-1} inputs go to the register *tr* $(2^{n-1} \ldots 2^n-1)$ and the last 2^{n-1} inputs go to *tr*$(0 \ldots 2^{n-1}-1)$ directly. In this way, for the second variable the inputs to the 2-to-1 multiplexers are the values of the vector *tr*. Now, if $cp_1 = 0$ then $(0 \ldots 2^{n-2}-1)$ of inputs go directly to the register *tr*$(0 \ldots 2^{n-2}-1)$ and the EXOR of $(0 \ldots 2^{n-2}-1)$ and $(2^{n-2} \ldots (2^{n-2}+2^{n-2}-1))$ inputs go to *tr*$(2^{n-2} \ldots (2^{n-2}+2^{n-2}-1))$. Accordingly, the transformations on the other variables are done.

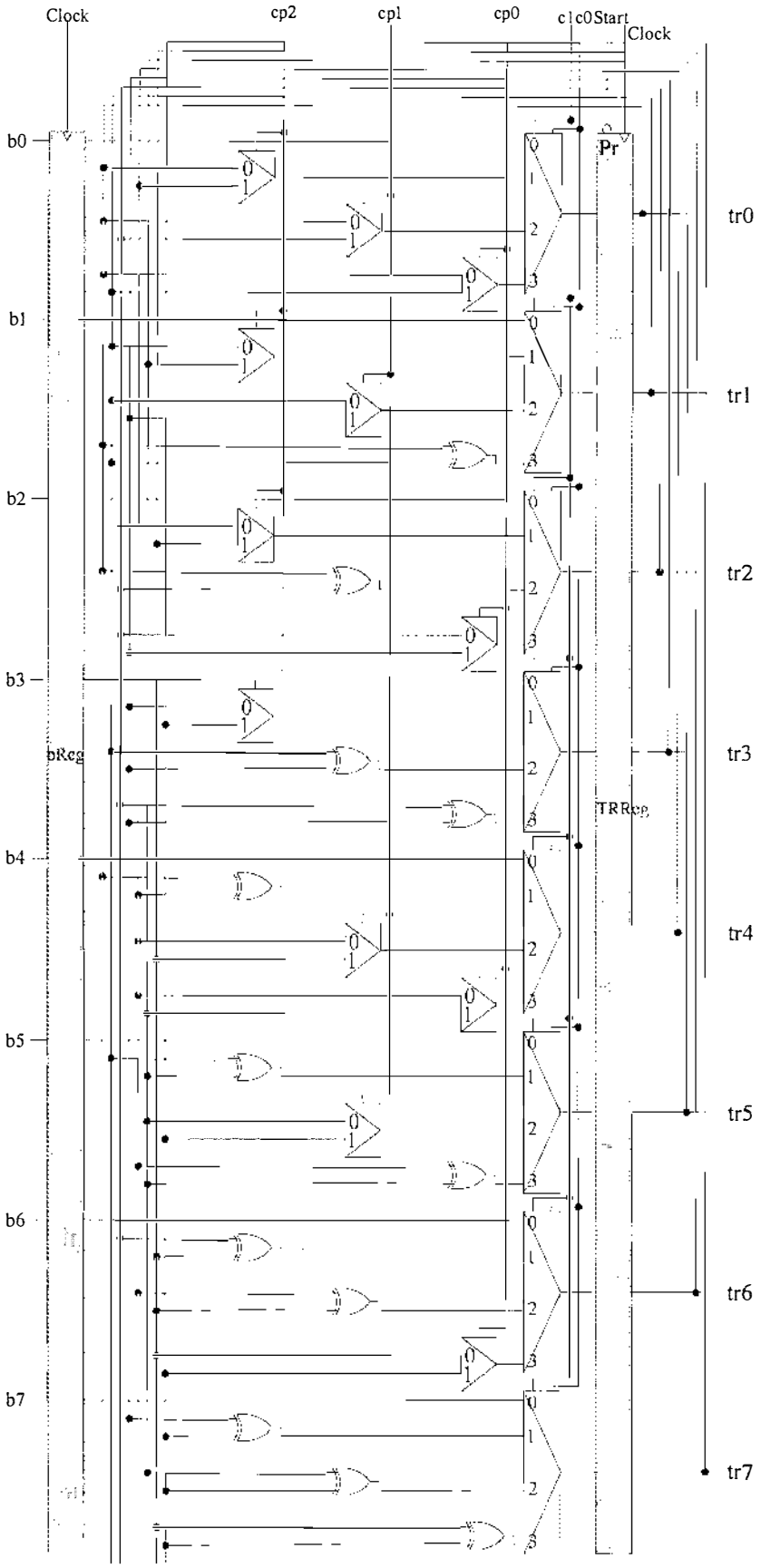


Figure 4.4: The converter

The *convert* produces 2^n bit FPRM coefficients for each polarity vector. For an n variable function we have 2^n polarity vectors and thus 2^n sets of FPRM coefficients.

PCtr and *SCtr* are three bit and two bit counters, respectively. PCtr provides 2^n numbers of n bit polarity vectors to *convert*. For a three variable function the converter takes four states to generate FPRM coefficients, where SCtr counts states for each polarity vector. At the first state the inputs are loaded into the converter, and then at each three of the following states the transformation on each variable is done. That is, for an n variable function it will need n+1 states for the converter to generate FPRM coefficients for a particular polarity vector.

The *CRReg* is a 2^n bit register which initially holds 2^n 1's. Then after the generation of each set of FPRM coefficients CRReg is loaded with the FPRM coefficients. That is, the converter passes the FPRM coefficients to CRReg.

The *adder* in the diagram is a module that adds the bits of the FPRM coefficients to determine the number of 1's in the FPRM coefficients. As we are intended to find the FPRM expressions with least number of product terms thus least number of 1's, we keep record of number of 1's in each set of FPRM coefficients. The adder computes the number of 1's in the FPRM coefficients for each polarity vector.

The register *RReg* in the design holds a set of FPRM coefficients which has the least number of 1's. If the i-th polarity vector produces FPRM coefficients which has lesser number of 1's than the FPRM coefficients produced by the $i+1$-th polarity then RReg holds the FPRM coefficients produced by the i-th polarity.

Another register *CPReg* holds the polarity vectors. When the converter computes the FPRM coefficients for a polarity vector then CPReg is loaded with the next polarity vector.

PReg register holds the value of the polarity which produces FPRM coefficients with least number of 1's. The process of keeping record of the polarity vector is the same as the process which stores the FPRM coefficients with least number of 1's in RReg.

The register *SReg* holds the value of number of 1's in the FPRM coefficients. Initially this register is loaded with the value of 2^n, as the FPRM coefficients can have at most 2^n 1's for an n variable function. Again the process of storing this value is same as the process which stores the FPRM coefficients with least number of 1's in RReg.

The *cmp* module used in the design is a comparator. This module compares the number of 1's in the FPRM coefficients produced by the polarity vector being used presently (cs) with the FPRM coefficients' number of 1's (s) produced by some other polarity vectors previously. If $cs < s$ then the output of cmp (the line *lt*) goes high. This *lt* enables the registers *RReg*, *PReg* and *SReg*.

After the computation of FPRM coefficients for 2^n polarity vectors we get the FPRM coefficients with the minimum number of 1's (r) in register *RReg*, the polarity vector (p) which produced r in register *PReg* and we get the number of 1's in r in register *SReg*.

All of the flip-flops used here are positive edge-triggered. And the total process is *reset* with the negative edge of *start*.

The transition between states is shown in Figure 4.5.

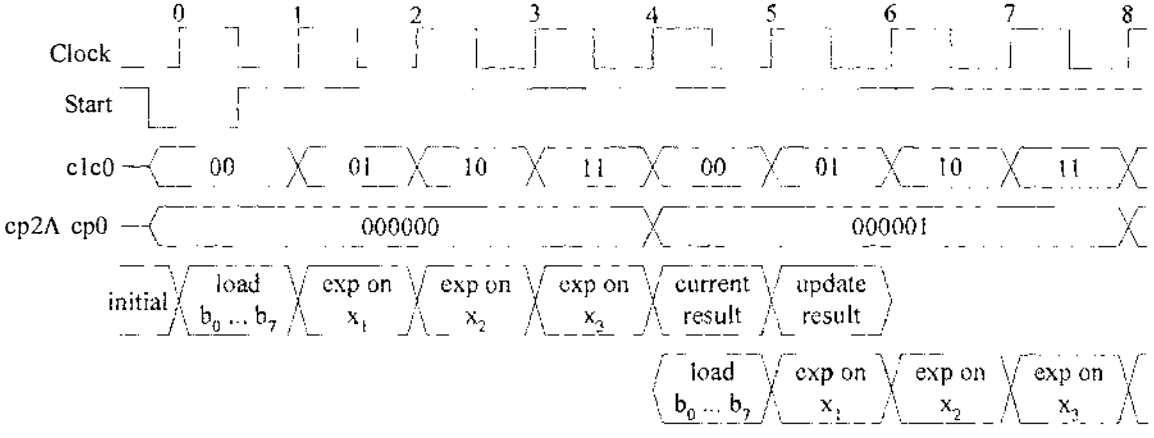


Figure 4.5: The pipelining of the minimization process

4.4 The ASM Chart of the developed design for FPRM Minimization

The ASM chart of the developed design for FPRM minimization is shown in Figure 4.6. As long as the circuit is in the initial state and the start signal is in the logic high, no action occurs and the system remains in the initial state. When the start signal goes to logic low then with the sensing of the negative edge of this signal the system starts to operate and goes to the next state. In this state the polarity counter, state counter, CRReg, SReg, the output register of the converter tr, and the input register of the converter b, are initialized.

In each state the state counter is incremented and the input of the converter is loaded with the coefficients of the input function. In the next three states the expansion on variables x_1, x_2 and x_3 is done respectively. In the next state the polarity counter is incremented; the output of the converter tr is loaded to the register cr and the register tp is loaded with the polarity value of the previous polarity counter.

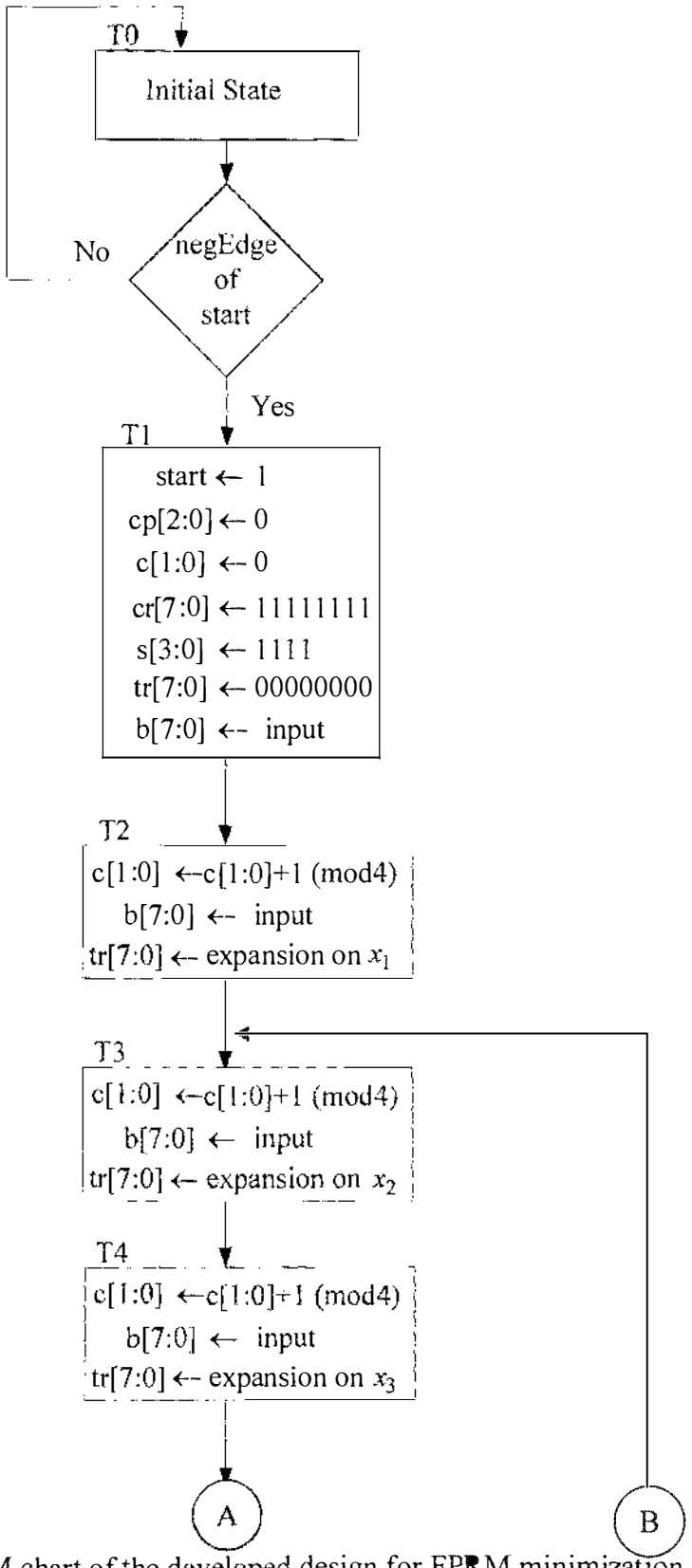


Figure 4.6: The ASM chart of the developed design for FPRM minimization

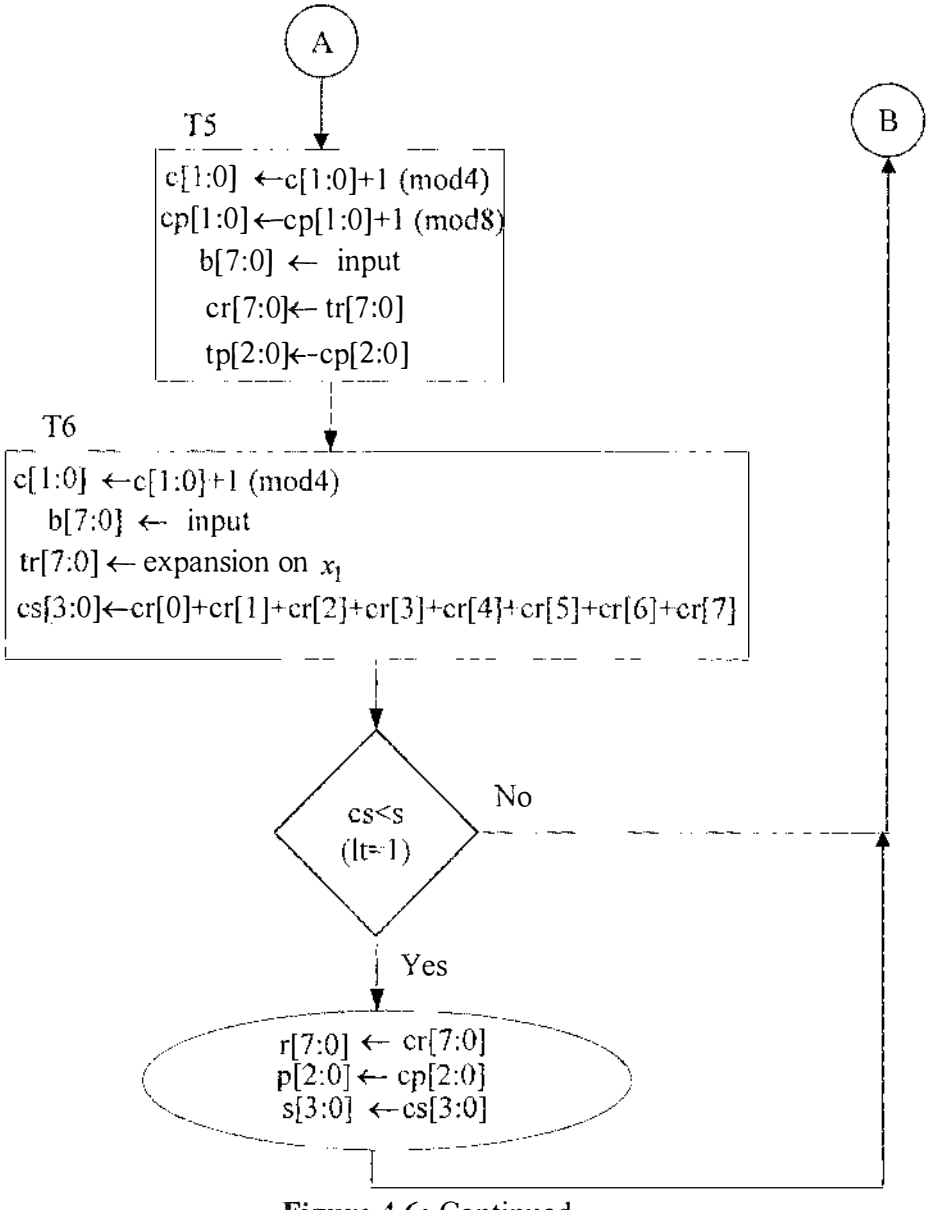


Figure 4.6: Continued

In the next state the expansion on x_1 for the current polarity vector is done and the FPRM coefficients generated by the counter are added to get the no of 1s in the FPRM coefficients and the value is stored in cs. The cs value is compared with the value of the SReg (s). If the comparison gives true value then the registers r (RReg), p (PReg) and s (SReg) are updated with the FPRM coefficients, the previous polarity and the value of number of 1's in the FPRM coefficients, respectively. If the comparison gives the false value then the control goes to state T3.

Chapter 5
FPGA Implementation of the ASIC for FPRM Minimization

5.1 Aspects of using FPGA

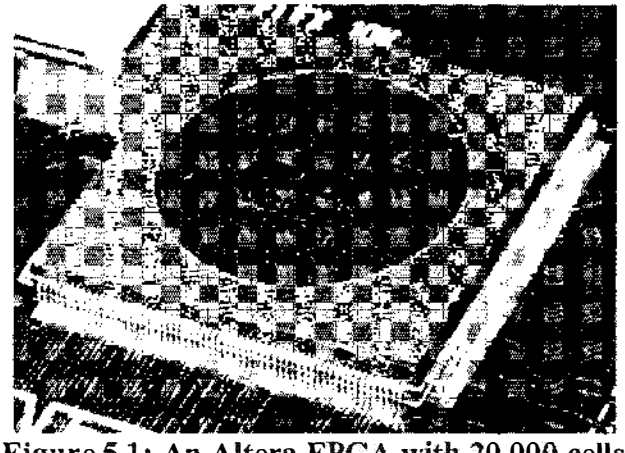


Figure 5.1: An Altera FPGA with 20,000 cells

We have discussed different types of ASICs in chapter 3 elaborately.

An FPGA is similar to a Programmable Logic Device, but whereas PLDs are generally limited to hundreds of gates, FPGAs support thousands of gates. They are especially popular for prototyping integrated circuit designs. Additionally, they take shorter time to market, ability to re-program in the field to fix bugs, and lower non-recurring engineering costs.

5.2 How FPGAs work

To define the behavior of the FPGA the user provides a hardware description language (HDL) or a schematic design. Common HDLs are VHDL and Verilog. Then, using an electronic design automation tool, a technology-mapped netlist is generated. The netlist can then be fitted to the actual FPGA architecture using a process called place-and-route, usually performed by the FPGA company's proprietary place-and-route software. The user will validate the map, place and route results via timing analysis, simulation, and other verification methodologies. Once the design and validation process is complete, the binary file generated (also using the FPGA company's proprietary software) is used to (re)configure the FPGA device.

n a typical design flow, an FPGA application developer will simulate the design at multiple tages throughout the design process. Initially the RTL description in VHDL or Verilog is imulated by creating test benches to stimulate the system and observe results. Then, after the ynthesis engine has mapped the design to a netlist, the netlist is translated to a gate level lescription where simulation is repeated to confirm the synthesis proceeded without errors. Finally the design is laid out in the FPGA at which point propagation delays can be added and the simulation run again with these values back-annotated onto the netlist.

To describe the developed design we have used Verilog as the HDL and the Quartus II 4.2 software to synthesize the design.

Compilation Reports:

Family	Device	Total logic elements	Clock Setup time (*fmax*)
Cyclone	EP1C6Q240C8	89 / 5,980(1 %)	129.62 MHz (period = 7.715 ns)
Cyclone	EP1C6Q240I7	89 / 5,980 (1 %)	140.17 MHz (period = 7.134 ns)
Cyclone II	EP2C5Q208C6	86 / 4,608 (1 %)	187.79 MHz (period = 5.325 ns)
Cyclone II	EP2C5T144C6	85 / 4,608 (1 %)	182.08 MHz (period = 5.492 ns)

The Compilation Report provides a lot of information that may be of interest to the designer. It indicates the speed of the implemented circuit. A good measure of the speed is the maximum frequency at which the circuit can be clocked, referred to as *fmax*. This measure depends on the longest delay along any path between two registers clocked by the same clock. The maximum frequency for our circuit implemented on the specified chip for the device **EP1C6Q240C8** is 129.62 MHz.

The simulation results for two input functions are shown below.

Here, the signal ***A*** is the minterm coefficients of the input function
signal ***r*** is the FPRM coeffcients
signal ***p*** is the polarity vector corresponding to *r* that produces *A*
signal *s* shows the value of no. of 1's in *r*

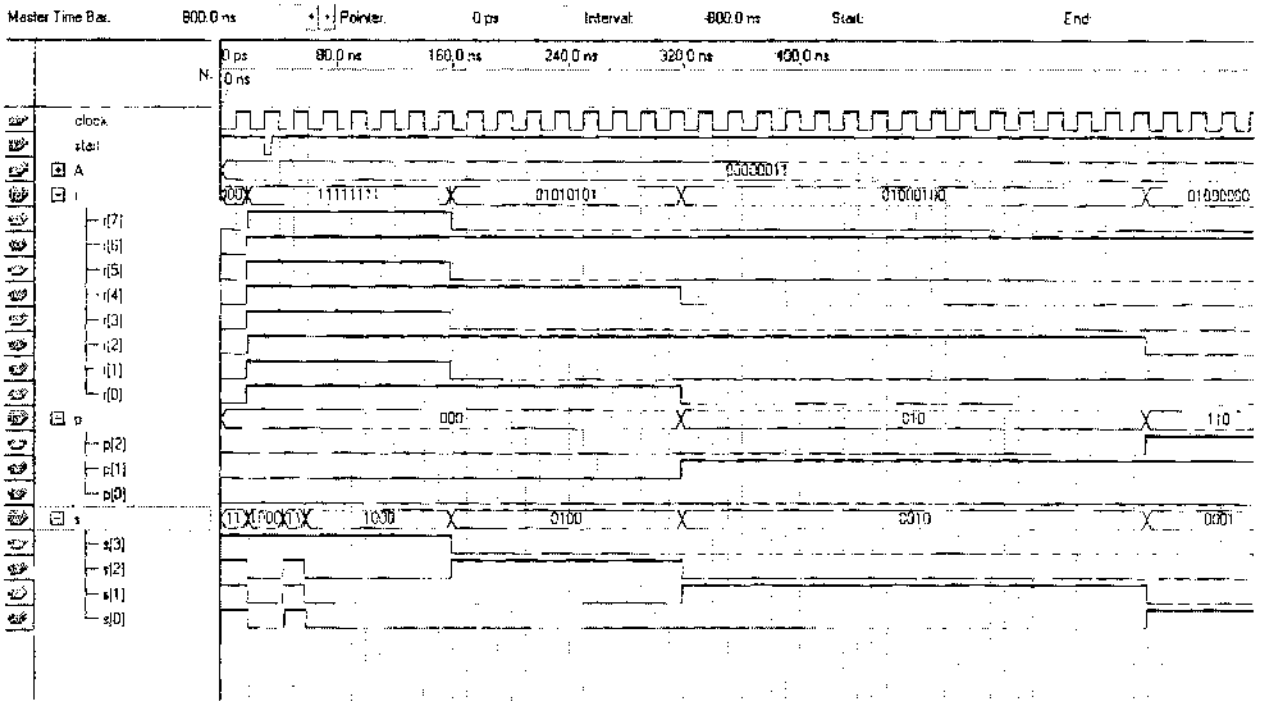


Figure 5.2: The timing simulation waveform for input function $[1, 1, 0, 0, 0, 0, 0, 0]^T$

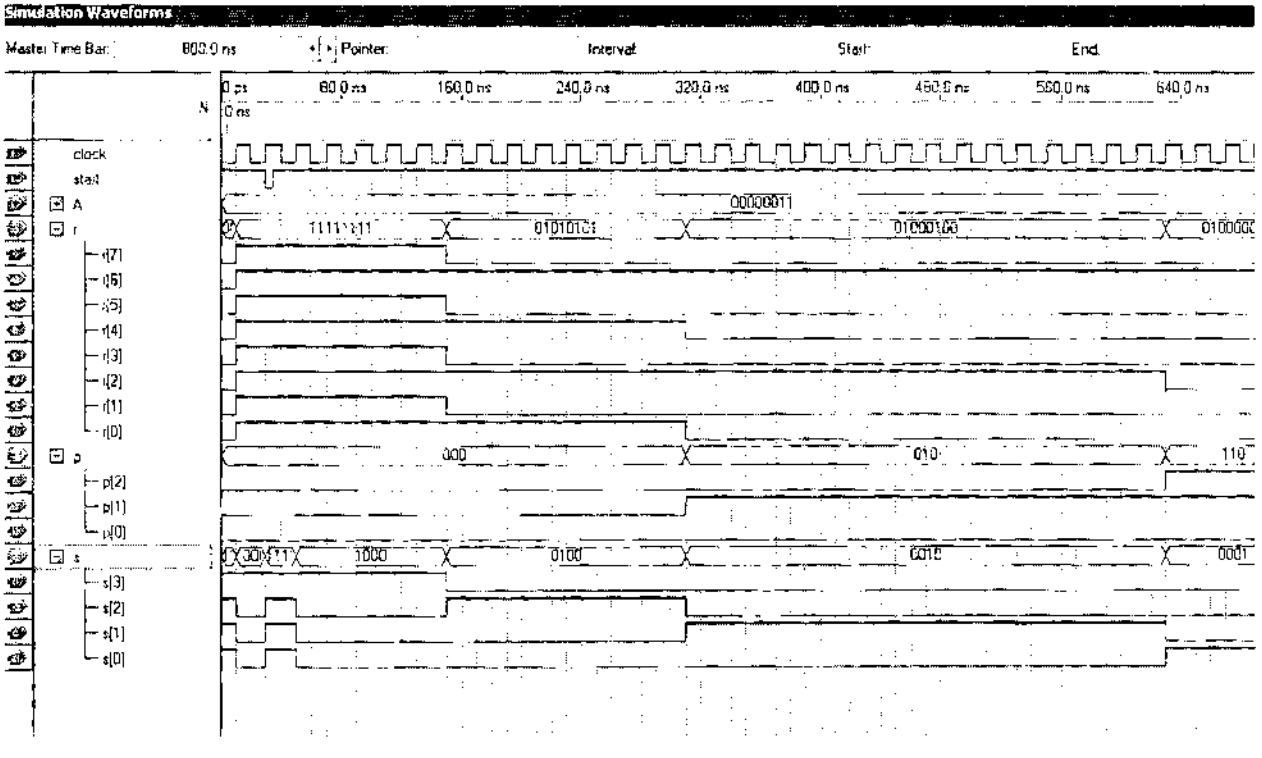


Figure 5.3: The functional simulation waveform for input function $[1, 1, 0, 0, 0, 0, 0, 0]^T$

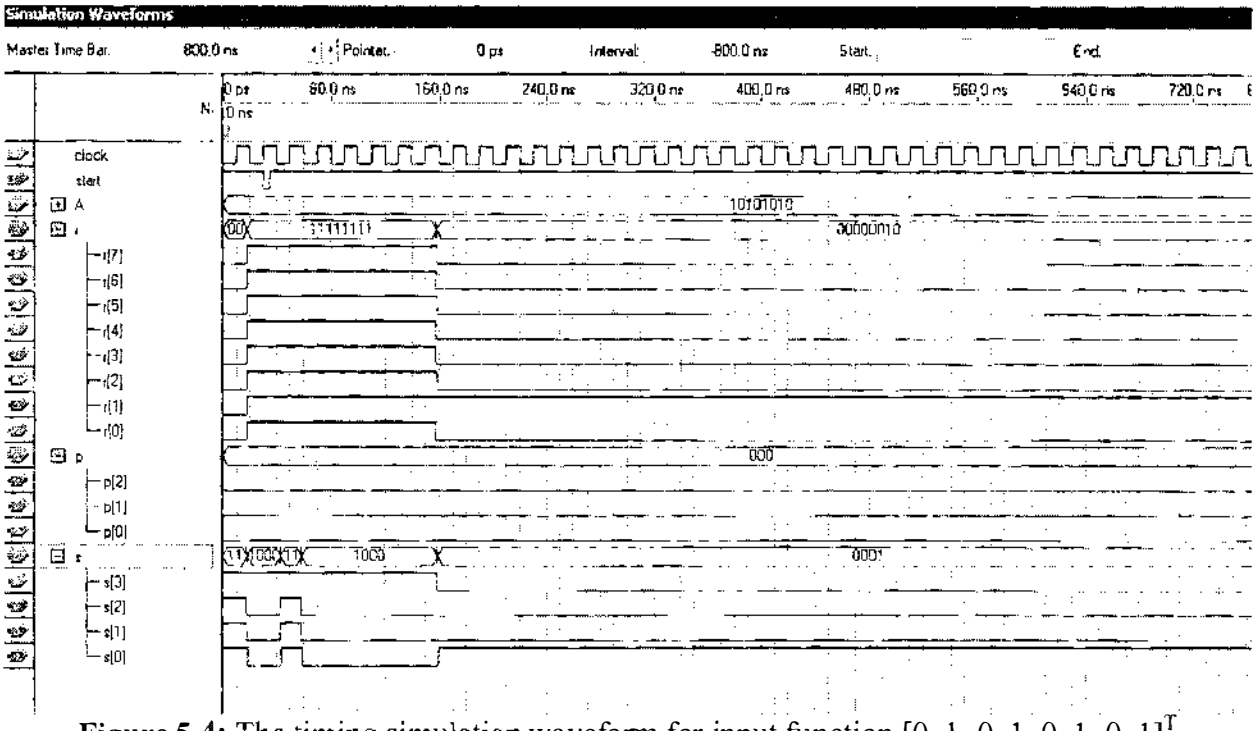


Figure 5.4: The timing simulation waveform for input function $[0, 1, 0, 1, 0, 1, 0, 1]^T$

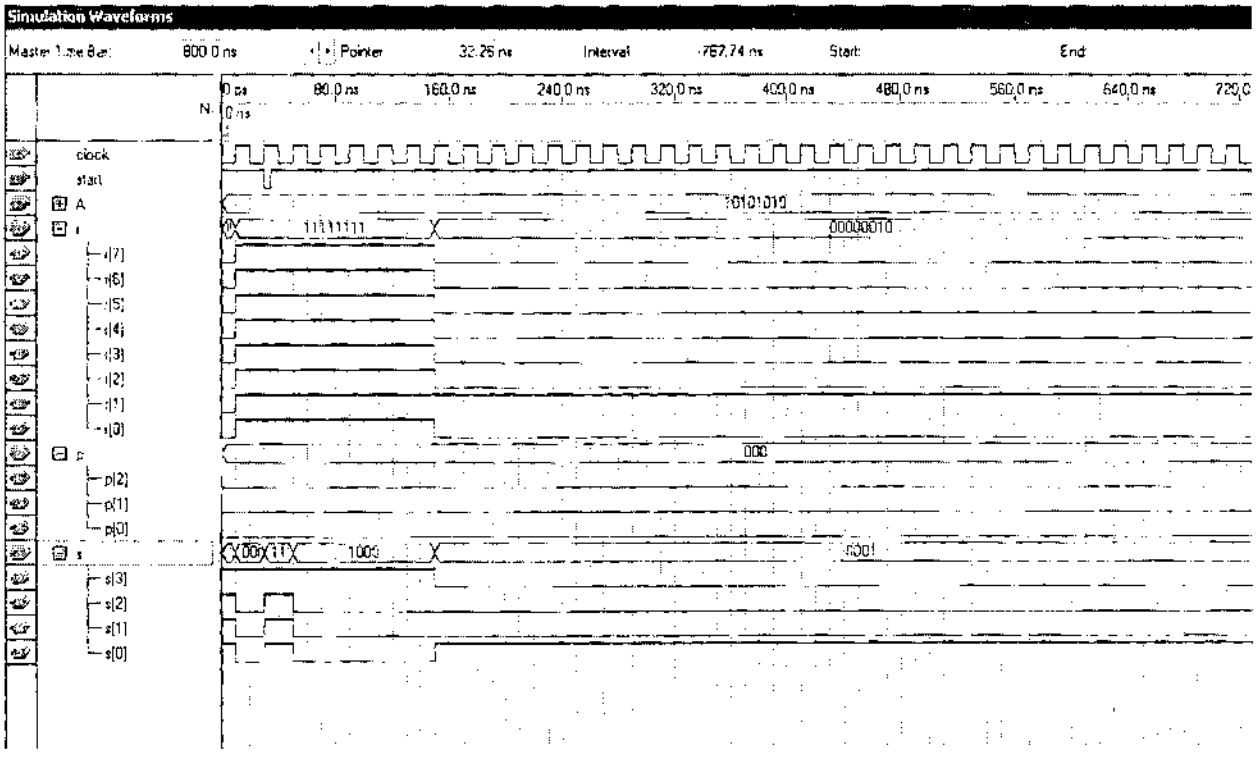


Figure 5.5: The functional simulation waveform for input function $[[0, 1, 0, 1, 0, 1, 0, 1]^T$

Chapter 6
Discussion and Conclusion

6.1 Discussion and Conclusion

AND-EXOR logic, which is also known as Reed-Muller logic, is now a days very popular for many reasons. There are seven types of AND-EXOR logic expressions, among them Fixed Polarity Reed-Muller (FPRM) expression is one type. This type has the property that the polarity of a variable remains same throughout the expression, which eases the implementation of the expression in VLSI. For an n-variable function, there are 2^n possible FPRM expressions having different number of products and number of literals. So, finding out the minimum FPRM expression for a given Boolean function is very important.

There are many software methods for FPRM minimization. In this work an approach to minimize FPRM expressions using hardware and implemented in FPGA has been outlined. In real life problems finding Boolean equivalence of a function is important. The FPRM representation is a tool to find Boolean equivalence or Boolean matching. Our designed ASIC will provide the FPRM coefficients and the optimum polarity vector for a particular Boolean function of three variables.

Many software approaches are available to minimize FPRM expressions. But for real time problems the software approaches are not applicable as they all take exponential time for computation. The ASIC will take constant time to generate FPRM coefficients. For this reason researchers focused on minimizing FPRM expressions using hardware.

6.2 Further Works

This design is able to minimize FPRM expressions of three variable single-output, fully-specified functions. Further works may include,

i. To parameterize the design so that the number of variables the ASIC can handle is n.

ii. To develop the design for handling multi-output and incompletely specified functions.

iii. To develop the design for optimization of other AND-EXOR expressions such as pseudo Reed-Muller, Kronecker, pseudo Kronecker, etc.

References

[Almaini 1996] A. E. A. Almaini and K. Burnside, October 1996, "Generalised Reed-Muller ASIC converter", *The 2nd Int. Conf. on ASIC*, China.

[Almaini 1997] A. E. A. Almaini, 1997, "A semicustom IC for generating optimum generalized Reed-Muller expansions", *Microelectronics Journal*, 28(2), 129-142.

[Besslich 1985] Ph. W. Besslich, 1985, "Spectral processing of switching functions using signal-flow transformations", *in (M. Karpovsky ed.) Spectral Techniques and Fault Detection, (Orlando, FL: Academic Press)*, 91-141.

[Bryant 1986] R. E. Bryant, Aug. 1986 "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, Vol. C-35, No. 8, pp. 677–691.

[Clarke 1993] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, June 1993, "Spectral transforms for large Boolean functions with applications to technology mapping," *Proc. Design Automation Conference*, pp. 54–60.

[Damarla 1989] T. R. Damarla and M. Karpovsky, 1989, "Fault detection in combinational networks by Reed-Muller transformations", *IEEE Transaction on Computers*, C-38, 788-797.

[Davio 1978] M. Davio, J. P. Deschamps, and A. Thayse, 1978, "Discrete and Switching Functions", (*McGraw-Hill International*).

[Davis 1991] L. Davis, 1991, "Handbook of Genetic Algorithms", *van Nostrand Reinhold, New York.*

[Debnath 1995] D. Debnath and T. Sasao, 1995, "GRMIN: a heuristic simplification algorithm for generalized Reed-Muller expressions", *IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design.*

[Debnath 2000] D. Debnath and T. Sasao, 2000, "Exact minimization of fixed polarity Reed-Muller expressions for incompletely specified functions," *Asia and South Pacific Design Automation Conference (ASP-DAC'2000)*, Yokohama, Japan, pp.247-252.

[Drechsler 1994] R. Drechsler, M. Theobald, and B. Becker, 1994, "Fast OFDD based minimization of fixed polarity Reed-Muller expressions" *European Design Automation Conf.*, pp. 2-7.

[Drechsler 1995] R. Drechsler, B. Becker, and N. Göckel, April, 1995 "A Genetic Algorithm for minimization of Fixed Polarity Reed-Muller expressions", *International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 392-395, Ales.

[Drechsler 1996] R. Drechsler, M. Theobald, and B. Becker, Nov., 1996, "Fast OFDD based minimization of fixed polarity Reed-Muller expressions" *IEEE Trans. Comput.*, Vol. C-45, No. 11, pp. 1294-1299.

[Drechsler 1997] R. Drechsler and B. Becker, Jan. 1997, "Sympathy: Fast Exact Minimization of Fixed Polarity Reed-Muller Expressions for Symmetric Functions." *IEEE Trans. CAD*, Vol. 16, #1, pp. 1-5.

[Fisher 1974] L. T. Fisher, 1974, "Unateness properties of AND-EXCLUSIVE-OR logic circuits", *IEEE Transaction on Computers*, 23, 166-172.

[Fleisher 1983] H. Fleisher, M. Tavel, and J. Yeager, 1983, "Exclusive-OR representations of Boolean functions", *IBM Journal of Research and Development*, 27, 412-416.

[Fujiwara 1985] H. Fujiwara, 1985, "Logic testing and Design for Testability", (*The MIT Press, Cambridge*).

[Goldberg 1989] D. E. Goldberg, 1989, "Genetic Algorithms in Search, Optimization, and Machine Learning", *Reading, MA: Addison-Wesley.*

[Green 1976] D. H. Green and I. S. Taylor, 1976, "Multiple-valued switching circuit design by means of generalized Reed-Muller expansions", *Digital Process*, 2, 63-81.

[Green 1987] D. H. Green, 1987, "Reed-Muller expansions of incompletely specified functions", *IEEE Proceedings on Computers and Digital Techniques*, 134, 228-236.

[Green 1991] D. H. Green, 1991, "Families of Reed-Muller canonical forms", *International Journal of Electronics*, 63(2), 259-280.

[Harking 1990] B. Harking, 1990, "Efficient algorithm for canonical Reed-Muller expansions of Boolean functions", *IEEE Proceeding on Computers and Digital Techniques*, 137(5), 366-370.

[Helliwell 1988] M. Helliwell and M. Perkowski, 1988, "A fast algorithm to minimize multi-output mixed-polarity generalized Reed-Muller forms", *Proceeding of the 25th Design Automation Conference*, 427-432.

[Hirayama 2001] T. Hirayama, K. Nagasawa, Y. Nishitani and K. Shimizu, 2001, "Double fixed-polarity Reed-Muller expressions: a new class of AND-EXOR expressions for compact and testable realization", *IPSJ Journal*, vol. 42 no. 4, pp. 983-991.

[Khan 1997] M. M. H. A. Khan and M. S. Alam, 1997, "Mapping of fixed polarity Reed-Muller coefficients from minterms, and the minimization of fixed polarity Reed-Muller expressions", *International Journal of Electronics*, 83(2), 235-247.

Kebschull 1993] U. Kebschull and W. Rosenstiel, Feb. 1993, "Efficient Graph-Based Computation and Manipulation of Functional Decision Diagrams", *Proc. European Design Automation Conf. '93*, pp. 278-282.

Mukhopadhyay 1970] A. Mukhopadhyay and G. Schmitz, 1970, "Minimization of exclusive OR and logical equivalence of switching circuits", *IEEE Transaction on Computer*, C-19, 132-140.

[Reddy 1972] S. M. Reddy, 1972, "Easily Testable realization for logic functions", *IEEE Transaction on Computer*, C-21(11), 1182-1188.

[Rollwage 1993] U. Rollwage, 1993, "The complexity of mod-2 sum PLA's for symmetric functions", *IFIP 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design*.

[Saluja 1975] K. K. Saluja and S. M. Reddy, 1975, "Fault detecting test set for Reed-Muller canonic networks", *IEEE Transaction on Computers*, C-24(10), 995-998.

[Sarabi 1992] Sarabi and M. A. Perkowski, 1992, "Fast exact and quasi-minimal minimization of highly testable fixed polarity AND/XOR canonical networks", *Proceeding of the 29th Design Automation Conference 1992*, Anaheim, CA, 30-35.

[Saul 1992] J. Saul, 1992,"Logic Synthesis for Arithmetic Circuits Using the Reed-Muller Representation", *Proc. EDAC-EuroASIC*, pp.109-113.

[Schafer 1991] I. Schafer and M. A. Perkowski, May 1991, "Multiple-Valued Input Generalized Reed-Muller Forms", *Proc. Of ISMVL '91*, pp. 40-48.

[Sasao 1991] T. Sasao, 1991, Oct. 1991, "On the complexity of some classes of AND-EXOR expressions", *IECE Technical Report FTS* 91-35.

[Sasao 1993a] T. Sasao, 1993a, "AND-EXOR Expressions and their Optimization", in "Logic Synthesis and Optimization", *(Kluwer Academic Publisher)*, 287-312.

[Sasao 1993b] T. Sasao, 1993b, "EXMIN2: A simplification algorithm for exclusive-OR sum of products expressions for multiple-valued input two-valued output functions", *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, 12(5), 621-632.

[Sasao 1994] T. Sasao, 1994, "Easily testable realization for generalized Reed-Muller expression", *IEEE the 3rd Asian Test symposium*, November 15-17, 1994, Nara, Japan, 157-162.

[Sasao 1995] T. Sasao, August 1995, "Representation of Logic Functions using EXOR Operations", *Proceeding of IFIP WG 10.5 Workshop on the Applications of the Reed-Muller Expansion in Circuit Design*, Chiba, Japan, 11-20.

[Sasao 1996] T. Sasao and F. Izuhara, 1996, "Exact minimization of FPRMs using multi-terminal EXOR TDDs," in T. Sasao and M. Fujita, eds., *Representations of Discrete Functions*, Kluwer Academic Publishers.

[Sasao 1997] T. Sasao, 1997, "Easily testable realizations for generalized Reed-Muller expressions", *IEEE Transaction on Computers*, 46(6), 709-716.

[Smith 1997] M. J. S. Smith, June 1997, "Application-Specific Integrated Circuits", *Addison-Wesley Publishing Company VLSI Design Series.*

[Toida 1992] S. Toida and N. S. V. Rao, 1992, "On test generation for combinational circuits consisting of AND and XOR gates", *Digest of Papers of 1992 IEEE VLSI Test Symposium. Design, Test and Application: ASICs and Systems-on-a-chip*, 113-118.

[Tsai 1994a] C. -C. Tsai and M. Marek-Sadawska, 1994a, "Boolean Matching using generalized Reed-Muller forms", *Proceeding of the 31st ACM/IEEE Design Automation Conference*, 339-344.

[Tsai 1994b] C. -C. Tsai and M. Marek-Sadawska, 1994b, *Minimization of fixed polarity AND/XOR canonical networks*, IEEE Proceedings on Computer and Digital Techniques, 141, 369-374.

[Tsai 1996] C. -C. Tsai and M. Marek-Sadawska, 1996, "Generalized Reed-Muller forms as a tool to detect symmetries", *IEEE Transaction on Computers*, C-45, 33-40.

[Tsai 1997] C. -C. Tsai and M. Marek-Sadawska, 1997, "Boolean functions classifications via fixed polarity Reed-Muller forms", *IEEE Transaction on Computers*, C-46(2), 173-186.

[Varma 1991] D. Varma and E. A. Trachtenberg, 1991, "Computation of Reed-Muller Expansions of Incompletely Specified Boolean Functions From Reduced Representations", *Proc. of IEE*, Vol. 138, Part E, No. 2, pp. 85-92.

[Wu 1982] X. Wu, X. Chen and S.L. Hurst, January 1982, "Mapping of Reed-Muller Coefficients and the Minimisation of Exclusive OR Switching Functions", *Proc IEE*, vol.129, Pt.E, No.1.