A New Algorithm for Factorization of Logic Expressions

by

Farzan Fallah

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1996

© Massachusetts Institute of Technology 1996. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
Jan 19, 1996

Certified by ....
Srinivas Devadas
Associate Professor
Thesis Supervisor

Accepted by ....
F. R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUL 1 6 1996

LIBRARIES

Eng.

A New Algorithm for Factorization of Logic Expressions

by

Farzan Fallah

Submitted to the Department of Electrical Engineering and Computer Science
on Jan 19, 1996, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Rapid progress in technology has led to fabrication of millions of transistors on a single chip. This increase in complexity, makes the use of computer tools mandatory in the design process. Many tools have been developed to help designers optimize the circuit for performance, power, and area; to increase the reliability of the circuit; or to decrease its cost and design time.

In this project one of these problems, i.e., decreasing the number of transistors, or in other words area of a combinational logic circuit, is considered, and a new algorithm is proposed to solve the factorization problem. A computer program implementing the algorithm has been developed.

Thesis Supervisor: Srinivas Devadas
Title: Associate Professor

Acknowledgments

I would like to thank Professor Srinivas Devadas, my research supervisor, for introducing me to the Computer Aided Design of VLSI and its problems and supervising this thesis.

I would also like to thank Silvina Hanono for reading my thesis and for her valuable suggestions.

I dedicate this thesis to my parents and my sister, for the encouragement and support during my study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Research over the past thirty years has led to efficient methods for optimizing and implementing combinational logic in a two-level form. Although the minimization of combinational logic in two levels is a nondeterministic polynomial-time (NP)-complete problem [6], there are some exact minimizers which can minimize most encountered functions in a reasonable amount of Central Processing Unit (CPU) time, like McBOOLE [3] and espresso-exact [9]. Furthermore, some heuristic minimizers, such as MINI [8] , MIS [2], and espresso [1], can get faster results, for larger Boolean functions, which are close to the minimum.

Using Programmable Logic Arrays (PLAs) every function can be easily implemented in sum-of-products form (SOP) [5]. This is usually done with a NOR-NOR architecture. This approach may lead to wasting more than 50% of silicon area. To get better utilization, methods like PLA folding have been proposed [7].

With rapid progress in technology and the necessity of fabricating large systems in a single chip, there are many cases in which using a two-level form or PLAs is not efficient for implementing Boolean functions. One of the problems is that for certain functions the number of product terms in SOP form, or sum terms in product-of-sums (POS) form grows exponentially with respect to the number of inputs.

For instance, although the Achilles heel function with $2n$ inputs, i.e., $(a_1 + b_1) \times$

$(a_2 + b_2) \times \cdots \times (a_n + b_n)$ has n sum terms in POS form, it has 2^n product terms in SOP form. The parity function with n inputs, i.e., $a_1 \oplus a_2 \oplus \cdots \oplus a_n$ has 2^n product terms in SOP form, and 2^n sum terms in POS form. In this case there is no efficient way to represent the function in two-level form.

Adders, subtractors, multipliers and comparators are other heavily used functions which have exponential number of products or sums in SOP or POS form respectively, so it is necessary to implement these functions in multi-level form.

Other problems with implementing Boolean functions in two-level form include low performance and high energy consumption in comparison to multi-level form implementation.

As a result, much research has been done attempting to find algorithms for minimizing a Boolean function in more than two levels, or in other words factorizing Boolean functions. Because of the NP-complete nature of this problem, it is impossible to search all possible solutions to find the best one; such a search requires a lot of computer memory and CPU time, so a method of searching a subset of solutions is typically used to find a solution as close as possible to the best solution. Searching a larger subset of possible solutions will likely result in finding a better solution at the expense of increased CPU time and memory. Thus there is a trade-off between the quality of the solution and the required CPU time and memory.

There are different approaches to the factorization problem. They can be divided into three groups:

1. Rule-based methods
2. Algebraic methods and
3. Boolean methods.

In rule-based methods, there are a set of rules which are used to transform certain patterns in the function into other patterns. The local nature of these methods, and lack of global perspective, result in limited capability [4].

In algebraic methods, Boolean functions are treated as polynomial functions, so a and a' are two different variables, and equalities like $a \times a' = 0$ and $a + a' = 1$ are not used. Furthermore, any equalities such as $a + a = a$ or $a \times a = a$ are not used either. Treating Boolean functions as polynomials simplifies the problem, resulting in efficient algorithms, but compromises the quality of the solution.

For example, using algebraic methods for the following function,

$$G = ab' + ac' + a'b + bc' + a'c + b'c$$

we can find the solution,

$$G = a \times (b' + c') + b \times (a' + c') + c \times (a' + b')$$

But using $a \times a' = 0$, $b \times b' = 0$ and $c \times c' = 0$ equalities we can find,

$$G = (a + b + c) \times (a' + b' + c')$$

which is a better solution.

In Boolean methods, Boolean equalities like $a \times a' = 0$, $a + a' = 1$, $a + a = a$ and $a \times a = a$ are used, so better solutions can be achieved at the expense of more CPU time and memory.

1.1 Recursive factorization using kernel intersection and covering (RIC)

A popular means of algebraic factorization uses the notion of kernels [1]. In this method all the kernels of a logic expression are generated and the expression is factored by sequentially selecting kernels or kernel intersections.

We propose a more global approach to factorization consisting of the following steps:

1. All the kernels of the multiple output function are computed.

2. All the intersections of the kernels are computed.

3. The cost of the kernels and intersections are computed.

4. A subset of the kernels, intersections, and cubes are selected to cover all the cubes of the multiple output function.

5. Steps 1-4 are repeated to further factorize the functions.

The number of levels of factorization in this method can be controlled easily through step 5, unlike previous methods. Because kernel intersections are computed, this algorithm can get better results than algorithms which use only kernels.

1.2 Overview

Chapter 2 describes the basic definitions used in this thesis.
Chapter 3 describes the rectangle covering method for factorization [10] implemented in the SIS package [11].
Chapter 4 describes the method used for computing the kernel intersections in RIC (step 2 of the algorithm).
Chapter 5 describes the cost computation of Boolean expressions (step 3 of the algorithm).
Chapter 6 describes different methods used to select a set of kernels, intersections, or cubes (step 4 of the algorithm).
Chapter 7 describes the recursive factorization of Boolean functions (step 5 of the algorithm).
Chapter 8 describes a few modifications to the algorithm to use Boolean equalities.
Chapter 9 shows results achieved by this algorithm in comparison with other methods.

Chapter 2

Basic Definitions

Binary Variable:

A binary variable is a symbol representing one dimension of a Boolean space (e.g., a, b).

Literal:

A literal is a Boolean variable or its complement (e.g., a, b').

Cube:

A cube is a set of variables such that only one of v or v' is its member, not both. For example, $\{a, b, c'\}$ is a cube, but $\{a, a'\}$ is not a cube. Each cube can be shown as multiplication of its literals, like $a \times b \times c'$.

Cover:

A cover is a set of cubes, e.g. $\{\{a, b, c'\}, \{a', b'\}\}$. Each cover can be interpreted as a sum-of-products expression of a function, so the previous cover can be interpreted as, $abc' + a'b'$.

Cost:

Cost of a cube, cover, or Boolean function is its literal count.

Support of a function:

Support of f denoted as $sup(f)$ is the set of all variables v for which at least one of v or v' occurs in f.

For example, if $f = a'b + ac$, then $sup(f) = \{a, b, c\}$.
If $sup(f) \cap sup(g) = \phi$, we say f is orthogonal to g.

For example, the two functions $f = a'b + ac$ and $g = d + e$ are orthogonal.

Algebraic divisor:

g is an algebraic divisor of f, if there exist r and h such that $f = g \cdot h + r$ where g is orthogonal to h, $h \neq 0$, and r has as few cubes as possible.

For example, if $f = ac + bc + d$, then $a + b$ is an algebraic divisor of f, because $f = (a + b) \times c + d$.
The function g divides f evenly if $r = \phi$.

For example, function $g = a + b'$ divides function $f = ac + b'c$ evenly, because $f = (a + b') \times c$ and $r = \phi$.

Primary divisors:

Primary divisors of f are defined as,

$$P(f) = \{f/c \mid c \text{ is a cube } \}$$

Kernel and cokernel:

The kernels of f are defined as,

$$K(f) = \{K \mid k \in P(f),\ k \text{ is cube-free } \}$$

For example, $a + b$ is a kernel of function $f = ac + bc + d$, but $ac + bc$ is not a kernel, because it is not cube-free.

Cokernel of a kernel k is f/k. In the previous example, c is a cokernel of kernel $a+b$.

A co-kernel pair is the pair of kernel and its cokernel, for example, $(c, a+b)$.

Boolean network:

A Boolean network is a directed acyclic graph such that for each node i in it, there is an associated cover F_i and a Boolean variable y_i representing the output of F_i. There is a one-to-one correspondence between a node and its cover, so they can be used interchangeably.

ON-set and OFF-set:

For each output f_i of f, the ON-set can be defined to be the set of input values x such that $f_i(x) = 1$. Similarly, the OFF-set is the set of input values x such that $f_i(x) = 0$.

Chapter 3

Rectangle Covering Method For Factorization

We review the rectangle covering method for factorization originally presented in[10].

3.1 Definitions

A rectangle (R, C) of a matrix B, $B_{i,j} \in \{0, 1, *\}$ is a subset of rows R and subset of columns C such that $B_{i,j} \in \{1, *\}$ for all $i \in R$ and $j \in C$.

A rectangle (R_1, C_1) is said to strictly contain rectangle (R_2, C_2) if $R_2 \subseteq R_1$ and $C_2 \subset C_1$, or $C_2 \subseteq C_1$ and $R_2 \subset R_1$.

A rectangle (R, C) of matrix B is said to be a prime rectangle if it is not strictly contained in any other rectangle of B. For example, in the following matrix,

	1	2	3	4	5	6	7
1	1	1	0	1	*	0	1
2	1	1	0	0	0	*	0
3	1	1	1	*	0	*	*
4	0	0	1	*	1	0	1
5	1	1	1	0	0	1	0

$(\{2,3\},\{1,2\})$ and $(\{1,2,3,5\},\{1,2\})$ are rectangles.

$\{2,3\} \subset \{1,2,3,5\}$ and $\{1,2\} \subseteq \{1,2\}$, so rectangle $(\{1,2,3,5\},\{1,2\})$ strictly contains rectangle $(\{2,3\},\{1,2\})$. As as result, $(\{2,3\},\{1,2\})$ is not a prime rectangle, but $(\{1,2,3,5\},\{1,2\})$ is a prime rectangle of matrix B. Note that the rows and columns of a rectangle are not necessarily adjacent.

3.2 The co-kernel cube matrix

The co-kernel pairs of a multiple output function can be shown with a co-kernel cube matrix. In this matrix every row corresponds to one co-kernel pair and every column corresponds to one cube which is present in at least one kernel.

The entry $B_{i,j}$ in the matrix is 1 if the co-kernel pair associated with row_i contains the cube associated with column j, otherwise it is 0.

For example, given the following multiple output function,

$$F = abc + abd + ae + be$$
$$G = acd + cg + ch + de + dg$$
$$H = bc + bd + gf + fh$$

the co-kernel pairs of F are:

$$(a, bc + bd + e)$$
$$(b, ac + ad + e)$$
$$(e, a + b)$$
$$(ab, c + d)$$

and the co-kernel pairs of G are:

$$(c, ad + g + h)$$
$$(d, ac + e + g)$$
$$(g, c + d)$$

and the co-kernel pairs of H are:

$$(b, c + d)$$
$$(f, g + h)$$

These co-kernel pairs can be shown as the following matrix,

			a	b	c	d	e	g	h	ac	ad	bc	bd
			1	2	3	4	5	6	7	8	9	10	11
F	a	1	0	0	0	0	1	0	0	0	0	1	1
F	b	2	0	0	0	0	1	0	0	1	1	0	0
F	e	3	1	1	0	0	0	0	0	0	0	0	0
F	ab	4	0	0	1	1	0	0	0	0	0	0	0
G	c	5	0	0	0	0	0	1	1	0	1	0	0
G	d	6	0	0	0	0	1	1	0	1	0	0	0
G	g	7	0	0	1	1	0	0	0	0	0	0	0
H	b	8	0	0	1	1	0	0	0	0	0	0	0
H	f	9	0	0	0	0	0	1	1	0	0	0	0

Note that there is no one-to-one relation between entries $B_{i,j}$ of the co-kernel cube matrix and cubes of the multiple-output function. In other words, for some cubes in the multiple-output function more than one associated $B_{i,j}$ occurs in the co-kernel cube matrix.

To show which entries in the matrix have the same cube, all the cubes of multiple-output function can be numbered and instead of 1 for the entries in the matrix, the number of associated cubes can be used. For example, the numbers associated with abc and abd are 0 and 1, so $B_{1,10} = 0$ and $B_{1,11} = 1$. Using this method and using "." instead of 0, the co-kernel cube matrix can be updated as:

			a	b	c	d	e	g	h	ac	ad	bc	bd
			1	2	3	4	5	6	7	8	9	10	11
F	a	1	.	.	.	.	2	.	.	.	.	0	1
F	b	2	.	.	.	.	3	.	.	0	1	.	.
F	e	3	2	3	.	.	.	.	.	.	.	.	.
F	ab	4	.	.	0	1	.	.	.	.	.	.	.
G	c	5	.	.	.	.	.	5	6	.	4	.	.
G	d	6	.	.	.	.	7	8	.	4	.	.	.
G	g	7	.	.	5	8	.	.	.	.	.	.	.
H	b	8	.	.	9	10	.	.	.	.	.	.	.
H	f	9	.	.	.	.	.	11	12	.	.	.	.

Each prime rectangle of the co-kernel cube matrix identifies an intersection of some kernels which is a common factor between the kernels of one output or different outputs of a function. The columns of the rectangle represent the cubes of an expression which can be used to divide the output(s), and the rows represent the cubes resulting from division of the output(s) by that expression.

For example, in the previous matrix, $(\{4, 7, 8\}, \{3, 4\})$ is a prime rectangle, so $c + d$ is a common factor of F , G and H , and it can be extracted from them. This corresponds to factorization of the functions in the form:

$$F = abX + ae + be$$
$$G = acd + ch + de + gX$$
$$H = bX + gf + fh$$
$$X = c + d$$

The matrix should be updated to show the cubes which have been covered and the new variable X.

			a	b	c	d	e	g	h	ac	ad	bc	bd
			1	2	3	4	5	6	7	8	9	10	11
F	a	1	.	.	.	.	2	.	.	.	.	*	*
F	b	2	.	.	.	.	3	.	.	*	*	.	.
F	e	3	2	3	.	.	.	.	.	.	.	.	.
F	ab	4	.	.	*	*	.	.	.	.	.	.	.
G	c	5	.	.	.	.	.	*	6	.	4	.	.
G	d	6	.	.	.	.	7	*	.	4	.	.	.
G	g	7	.	.	*	*	.	.	.	.	.	.	.
H	b	8	.	.	*	*	.	.	.	.	.	.	.
H	f	9	.	.	.	.	.	11	12	.	.	.	.
X	1	10	.	.	13	14	.	.	.	.	.	.	.

This process can be repeated until no number remains in the table. The result will be a multiple-level form of the function. Different heuristics can find the rectangles which will give the best factorization.

3.3 Multiple covering

If rectangles are allowed to cover $*$'s as well, some cubes will be covered several times, which means the $a + a = a$ equality has been used. This can lead to a better factorization.

In the previous matrix, $(\{5,9\},\{6,7\})$ rectangle covers $B_{5,6}$ which is a $*$. Using this rectangle leads to factorization of the multiple output function as:

$$F = abX + ae + be$$
$$G = acd + de + cY + gX$$
$$H = bX + fY$$
$$X = c + d$$
$$Y = g + h$$

which means cube cg in G has been covered two times.

3.4 Deficiencies of the algorithm

After introducing a new variable, this method does not add its kernels to the co-kernel cube matrix. Because the algorithm omits this step, it finds no common factors between the kernels of the new variable and other expressions.

Also, no easy way exists to bound the number of levels of factorization in this method, so sometimes the algorithm finds a solution with a lot of levels, which may be an inappropriate implementation due to performance reasons.

Chapter 4

Kernel Intersection

In this chapter, we describe the method used for computing kernel intersections in resursive factorization.

4.1 A pair of sets

A pair of sets is defined as,

$$P_i = (X_i,\, Y_i) \quad \text{where } X_i \text{ is a set of sets, and } Y_i \text{ is a set.}$$

Intersection of two pairs of sets, $P_i = (X_i,\, Y_i)$ and $P_j = (X_j,\, Y_j)$ is defined as,

$$I_{1,k} = P_i \sqcap P_j = (X_i \uplus X_j,\, Y_i \cap Y_j)$$

where $X_i = \{a_{i,0}\ , a_{i,1}, a_{i,2}, \ldots, a_{i,N-1}\}$,

$a_{i,k}$ is a set for $0 \leq k \leq N-1$,

Y_i is a set,

$\cap$ is the intersection operator on the sets,

$\uplus$ is a binary operator defined as,

$$X_i \uplus X_j = \{a_{i,k} \cup a_{j,k}\}, \quad 0 \leq k \leq N-1.$$

Intersection of level l is defined as,

$$I_{l,k} = I_{l-1,i} \sqcap I_{l-1,j} \text{ where } I_{l-1,i} \text{ and } I_{l-1,j} \text{ are two intersections of level } l-1,$$
$$\text{with } l > 0 \text{ and } I_{0,k} = P_k$$

$\sqcap$ operator is reflexive, commutative and associative, in other words,

$$I_{l,i} \sqcap I_{l,i} = I_{l,i},\ \ I_{l,i} \sqcap I_{l',j} = I_{l',j} \sqcap I_{l,i},\ \ \text{and } I_{l,i} \sqcap (I_{l',j} \sqcap I_{l'',k}) = (I_{l,i} \sqcap I_{l',j}) \sqcap I_{l'',k}.$$

The order of an intersection $O(I)$ can be defined as the number of pairs that have I as their intersection.
For example, if $I_{1,i} = P_0 \sqcap P_1$, then $O(I_{1,i}) = 2$. For $I_{2,k} = I_{1,i} \sqcap I_{1,j}$, where $I_{1,i}$ is defined as before and $I_{1,j} = P_0 \sqcap P_2$, $O(I_{2,k}) = 3$, because $I_{2,k} = P_0 \sqcap P_1 \sqcap P_2$.

Computing all the level l intersections of a set of pairs, will result in all the intersections of order k, where $2^{l-1} < k \leq 2^l$. For example, computing intersections of level 3 will result in intersections of order 5, 6, 7 and 8.

Furthermore, intersecting all the intersections of level $l-1$ where their orders are $2^{(l-1)}$ will result in all the intersections of order k, where $2^{(l-1)} < k \leq 2^l$. This is so because each intersection of order k can be achieved by intersecting two intersections of order $2^{(l-1)}$ which have $2^l - k$ pairs in common. For example, if $l = 3$ all the intersections of order k, $4 < k \leq 8$ can be achieved using intersections of order 4. The intersection of 5 pairs, say P_0, P_1, P_2, P_3, P_4 can be achieved by intersecting $I_{3,i} = P_0 \sqcap P_1 \sqcap P_2 \sqcap P_3$ and $I_{3,j} = P_0 \sqcap P_1 \sqcap P_2 \sqcap P_4$.

4.2 Co-kernel and pair of sets

A co-kernel can be represented by a pair of sets, where Y is the kernel and for $0 \leq i \leq N-1$, a_i is the quotient of dividing $output_i$ by the kernel.
For example, $(a,\ bc + d)$ can be expressed by the following pair of sets,

$$(\{\,\{a\}\,\},\ \{bc,\ d\})$$

or its kernel set can be written in SOP form,

$$(\{\{a\}\},\ \{bc+d\})$$

Using the $\sqcap$ operator, the intersection of kernels can be defined as,

$$I_{l,k} = (cokernel_i,\ kernel_i) \sqcap (cokernel_j,\ kernel_j)$$

where $kernel_m,\ m \in \{i,j\}$ is a set of cubes,

$cokernel_k = \{a_{k,0},\ a_{k,1},\ a_{k,2},\ \ldots,\ a_{k,N-1}\},\ k \in \{i,j\}.$

where $a_{k,m}$ is a set of cubes for $0 \leq m \leq N-1$,

$N =$ Number of outputs.

For example, for the following multiple output function,

$$F = ad + ae + af + ag + i$$
$$G = bd + be + bf + bh + j$$
$$H = cd + ce + cg + ch + k$$

the co-kernels are,

$$I_{0,0} = (\{\{a\},\phi,\phi\},\ d+e+f+g)$$
$$I_{0,1} = (\{\phi,\{b\},\phi\},\ d+e+f+h)$$
$$I_{0,2} = (\{\phi,\phi,\{c\}\},\ d+e+g+h)$$

the result of the first level of intersection is,

$$I_{1,0} = I_{0,0} \sqcap I_{0,1} = (\{\{a\},\{b\},\phi\},\ d+e+f)$$
$$I_{1,1} = I_{0,0} \sqcap I_{0,2} = (\{\{a\},\phi,\{c\}\},\ d+e+g)$$
$$I_{1,2} = I_{0,1} \sqcap I_{0,2} = (\{\phi,\{b\},\{c\}\},\ d+e+h)$$

and the result of the second level of intersection is,

$$I_{2,0} = I_{1,0} \sqcap I_{1,1} = (\{\{a\},\{b\},\{c\}\},\ d+e)$$
$$I_{2,1} = I_{1,0} \sqcap I_{1,2} = (\{\{a\},\{b\},\{c\}\},\ d+e)$$
$$I_{2,2} = I_{1,1} \sqcap I_{1,2} = (\{\{a\},\{b\},\{c\}\},\ d+e)$$

Using kernel intersection, common factors between kernels of one function or different outputs of a multiple output function can be found and extracted. These common factors can be implemented only once to decrease the number of literals.

For example, using kernel intersection for two co-kernels, $(\{\{a\},\phi\},\ bcd+efg+h\ )$ and $(\{\phi,\{a'\}\},\ bcd+efg+i\ )$ of the following multiple output function,

$$F = abcd + aefg + ah$$
$$G = a'bcd + a'efg + a'i$$

the following pair will result,

$$(\{\{a\},\{a'\}\},\ bcd + efg)$$

so they can be factored as,

$$F = aX + ah$$
$$G = a'X + a'i$$
$$X = bcd + efg$$

while without using kernel intersection, the following less desirable solution is achieved,

$$F = a \times (bcd + efg + h)$$
$$G = a' \times (bcd + efg + i).$$

Each intersection in the l^{th} level is the intersection of at most 2^l kernels, so for a single output function, the cokernel has at most 2^l cubes. This means that in each level the maximum number of cubes of the cokernel of each intersection is at most 2 times greater than the maximum number of cubes of the cokernel of intersections of the previous level. Therefore, in computing the intersection of the l^{th} level of one function using intersection of $(l-1)^{th}$ level, we can use only intersections with no fewer than 2^l cubes in their kernel part. The reason is in other cases, the intersections obtained may have less than 2^l cubes in their kernel parts. For each intersection with less than 2^l cubes $(I1)$, there will be one intersection $(I2)$ with these properties:

1. Cokernel of $I2$ is the kernel of $I1$.

2. Kernel of $I2$ is a superset of the cokernel $I1$.

In other words, if there is an intersection,

$$I_{l,k} = (X_k,\, Y_k) \text{ where the number of members of } Y_k \text{ is less than } 2^l$$

then there is another intersection such that,

$$I_{l',k'} = (Y_k,\, X_{k'}) \text{ where } X_k \subseteq X_{k'} \text{ .}$$

This means $I_{l',k'}$ covers at least the same cubes as $I_{l,k}$ covers, so $I_{l',k'}$ is at least as good as $I_{l,k}$.

This can be shown by the following example, assuming the following function,

$$F = ae + af + ag + be + bf + bg + ce + cf + cg + de + df + dg$$

whose kernels are,

$$\begin{aligned}
I_{0,0} &= (\{\{a\}\},\, e + f + g) \\
I_{0,1} &= (\{\{b\}\},\, e + f + g) \\
I_{0,2} &= (\{\{c\}\},\, e + f + g) \\
I_{0,3} &= (\{\{d\}\},\, e + f + g) \\
I_{0,4} &= (\{\{e\}\},\, a + b + c + d) \\
I_{0,5} &= (\{\{f\}\},\, a + b + c + d) \\
I_{0,6} &= (\{\{g\}\},\, a + b + c + d)
\end{aligned}$$

the result of the first level of intersection is,

$$\begin{aligned}
I_{1,0} &= I_{0,0} \sqcap I_{0,1} = (\{\{a + b\}\},\, e + f + g) \\
I_{1,1} &= I_{0,2} \sqcap I_{0,3} = (\{\{c + d\}\},\, e + f + g) \\
I_{1,2} &= I_{0,4} \sqcap I_{0,5} = (\{\{e + f\}\},\, a + b + c + d) \\
I_{1,3} &= I_{0,4} \sqcap I_{0,6} = (\{\{e + g\}\},\, a + b + c + d) \\
&\vdots
\end{aligned}$$

the result of the second level of intersection is,

$$I_{2,0} = I_{1,0} \sqcap I_{1,1} = (\{\{a+b+c+d\}\},\ e+f+g)$$
$$I_{2,1} = I_{1,2} \sqcap I_{1,3} = (\{\{e+f+g\}\},\ a+b+c+d)$$
$$\vdots$$

Because the number of cubes in the Y part of $I_{2,0}$ is 3, which is smaller than 4 (2^l), there is another intersection ($I_{2,1}$) with its X part equal to the Y part of $I_{2,0}$, and its Y part is a superset of the X part of $I_{2,0}$. Therefore, it is not necessary to compute the intersection of $I_{1,0}$ and $I_{1,1}$ because their Y parts have 3 members, which is smaller than 4, and clearly the Y part of the intersection will have less than 4 members.

Using this analogy and the fact that the number of cubes in the X part times the number of cubes in the Y part cannot be greater than the number of cubes in the function, we have,

$$2^l \times 2^l \geq N_c$$

where the minimum of l in this inequality is the maximum level of intersection that is necessary and N_c is the number of cubes in the single output function.

For multiple output functions we can assume (just for computing the maximum level of intersections) that first, the intersections of kernels of each output are computed separately. Computing this needs l_1 levels of intersections, where l_1 is the smallest number satisfying,

$$2^{l_1} \times 2^{l_1} \geq Max(N_c)$$

where $MAX(N_c)$ is the largest number of cubes of a single output.

After that, l_2 levels of intersection should be computed to get the intersections of the intersections of single outputs together, where l_2 is the smallest number satisfying,

$$2^{l_2} \geq N$$

where N is the number of outputs.

So, all the intersections can be computed after $l_1 + l_2$ levels where,

$$2^{(l_1+l_2)} \geq N \times \sqrt{MAX(N_c)}.$$

The minimum of $(l_1 + l_2)$ in this inequality is a maximum bound on the number of levels of intersection that should be computed.

The number of kernels grows exponentially with the number of variables of the function, and for a function with K kernels the number of first level intersections can be as large as $\frac{K(K-1)}{2}$, so the number of intersections can be quite large. This is usually the case for the early levels of intersection and usually the number of intersections drops rapidly after this. The huge number of intersections makes the computation of intersections very expensive in terms of CPU time and memory. To solve this problem, different intersections with the same Y part can be merged together to make a single intersection, or $I_{l,i} = (X_i, Y_i)$ and $I_{l,j} = (X_j, Y_i)$ can be merged to form $I_{l,i} = (X_i \uplus X_j, Y_i)$.

In the previous example, $I_{1,0}$ with $I_{1,1}$ and $I_{1,2}$ with $I_{1,3}$ can be merged to make one intersection each,

$$I_{1,0} = (\{\{a + b + c + d\}\}, e + f + g)$$
$$I_{1,2} = (\{\{e + f + g\}\}, a + b + c + d)$$

Merging intersections helps to decrease the amount of memory and computation time of the next level of intersection and of the cost. It can decrease the number of required levels of intersection as well. It also simplifies the selection step. Table 4.1 shows the results of kernel intersection for several examples. Note that the actual number of levels of intersection is less than predicted by theory. The reason is the rapid drop of the number of intersections.

Merging several intersections means that in selection step, all of them should be selected or deselected together. This can degrade the final result in some cases.

Table 4.2 shows the number of intersection merges done in the first and second level of intersections for several examples.

	In/Out	Cubes	Kernels	Intersection Levels		Kernel Gen. Time (s)	Inter. Time (s)
				Actual	Theory		
ADD4	9/5	135	793	3	6	2.00	2.40
XOR5	5/1	16	146	1	2	0.02	0.06
INC	7/9	99	475	5	6	1.42	1.37
SQUARE5	5/8	85	412	5	5	1.05	5.17
BW	5/28	87	251	3	7	0.69	0.25
MISEX1	8/7	32	73	3	5	0.04	0.01
MISEX2	25/18	29	40	1	5	0.04	0.02
5XP1	7/10	75	215	3	6	0.23	0.72
RD53	5/3	32	214	3	4	0.10	0.19
RD73	7/3	141	2263	4	5	14.70	28.04
B12	15/9	82	176	4	7	0.16	0.10
SAO2	10/4	78	593	5	5	1.18	1.33
VG2	25/8	110	370	2	6	0.76	0.78

Table 4.1: CPU time for computing kernels, intersections, and levels of intersection for several examples.

	# Kernels	# intersections in level 1	# merges in level 1	# intersections in level 2	# merging in level 2
ADD4	793	237	4675	105	176
XOR5	146	40	100	0	0
INC	475	409	943	221	834
SQUARE5	412	358	1703	322	1809
BW	251	134	474	35	108
MISEX1	73	34	33	14	26
MISEX2	40	405	4	0	0
5XP1	215	100	169	28	24
RD53	214	119	444	68	2
RD73	2263	1212	25049	810	5442
B12	176	61	208	29	44
SAO2	593	240	245	68	95
VG2	370	41	7216	2	4

Table 4.2: Number of kernels, intersections, and intersection merges for several examples.

Chapter 5

Cost Computation

The computation of the cost of the co-kernels and intersections must follow the computation of the co-kernels and intersections. Cost can be computed by counting the literals in the SOP form of each set of co-kernels and intersections. Note that when computing the cost of an intersection between different outputs, the number of non-empty sets of the X part of each pair should be added to the cost, because in this case a new variable should be added to the network that is the input to each of those outputs.

For example, for the following multiple output function,

$$F = ab + ac + ad$$
$$G = a'b + a'c + a'd$$

the result of factorization will be,

$$F = a \times X$$
$$G = a' \times X$$
$$X = b + c + d$$

which means the cost of $(\{\{a\}, \{a'\}\},\ \{b+c+d\})$ is 7 (2 + 3 + 2 (because of variable X)).

Better results can be achieved by using the quick-factor or the good-factor algorithm, to factorize the SOP form, and by counting the literals in the factored form instead of the SOP form. Table 5.1 shows the effect on the number of literals after the first level of factorization when counting literals in the SOP form versus the factored forms. Table 5.2 shows the elapsed time for computing cost in the different methods.

	# literals Without Factoring	# literals Quick Factor	# literals Good Factor
ADD4	133	93	93
XOR5	32	28	28
BW	≤ 268	≤ 264	≤ 249
MISEX1	75	74	74
MISEX2	167	164	164
5XP1	177	159	159
RD53	75	68	68
RD73	197	≤ 192	≤ 188
B12	127	106	103
SAO2	243	184	183
VG2	208	182	182

Table 5.1: Number of literals for several examples.

	Without Factoring (sec)	Quick Factor (sec)	Good Factor (sec)
ADD4	0.30	4.58	7.96
XOR5	0.04	0.06	0.57
BW	0.13	0.41	0.63
MISEX1	0.03	0.12	0.19
MISEX2	0.02	0.07	0.09
5XP1	0.09	0.48	0.72
RD53	0.10	0.73	1.25
RD73	1.32	34.41	91.77
B12	0.08	0.43	0.71
SAO2	0.24	2.62	6.52
VG2	0.14	0.90	1.73

Table 5.2: Time for computing cost for several examples.

Chapter 6

Selection of the Best Cover

With the generation of all the kernels and their intersections and the computation of their costs, a table with kernels, intersections and cubes in its columns, and cubes of the function in its rows can be constructed. Each entry of $B_{i,j}$ in the table is X if the column j covers the cubes associated with row i, otherwise it is a "." .
Table 6.1 shows the related table for the function,

$$F = ab + ac + bc$$

To factorize the function, a subset should be selected, from the set of columns of the table. This subset should cover at least one X of each row and have the least cost.

For finding the best solution, the Branch and Bound method can be used. It has been shown that this covering problem is NP-complete [6], and hence the Branch and

	$a \times (b+c)$	$b \times (a+c)$	$c \times (a+b)$	$a \times b$	$a \times c$	$b \times c$
	1	2	3	4	5	6
$a \times b$ 1	X	X	.	X	.	.
$a \times c$ 2	X	.	X	.	X	.
$b \times c$ 3	.	X	X	.	.	X

Table 6.1: Kernels, intersections, and cubes of function F.

Bound method may take exponential time.

To decrease the amount of CPU time, a greedy method for selecting the best cover has also been tried. Since this greedy method is used in the first part of the Branch and Bound method, it will be explained first.

6.1 Greedy selection

In each step of the greedy algorithm, we select a column with the smallest f where f is a function of the cost of that column and the number of uncovered rows of the table which would be covered by this selection (N_{UR}).

The reason that we use N_{UR} instead of the number of cubes which will be covered, is that covering cubes which have already been covered (multiple covering) is usually redundant and we wish to prevent it. Of course in some cases, multiple covering can get better results, but not with greedy selection.

One simple and good function for f is $cost/N_{UR}$. Using this function leads to the selection of kernels, intersections, or cubes with a small number of literals that cover a large number of cubes.

Another function is $cost/N_{UR}^2$. This function is biased in favor of entries that cover many cubes.

This greedy method for selection is very fast because both functions can be computed quickly, and for each selection, only one search through all the columns is necessary and can be done in $O(N)$ time. This method usually gives good results. Table 6.2 shows the cost of the selected cover by the greedy method in comparison to the best solution. The subscript included in the name of example shows that the data is related to the the first or second or third level of factorization of function. For example, $ADD4_1$ shows that the data is related to the first level of factorization of $ADD4$ function.

As is obvious from the table, $cost/N_{UR}$ generally gives better results than

	Greedy ($Cost/N_{UR}$)	Greedy ($Cost/N_{UR}^2$)	Best solution
$ADD4_1$	93	93	93
$ADD4_2$	47	45	45
$XOR5_1$	28	28	28
BW_1	273	298	$\leq$ 264
$MISEX1_1$	80	80	74
$MISEX1_2$	33	30	30
$MISEX2_1$	173	272	164
$MISEX2_2$	67	100	65
$5XP1_1$	169	173	$\leq$ 159
$RD53_1$	72	70	68
$RD53_2$	39	39	39
INC_1	211	225	199
$B12_1$	124	106	106
$B12_2$	67	72	67
$SAO2_1$	206	199	184
$SAO2_2$	146	137	133
$VG2_1$	241	192	182
$VG2_2$	63	67	58
$VG2_3$	31	36	29
$RD73_1$	197	218	$\leq$ 192
TOTAL	2360	2480	$\leq$ 2179

Table 6.2: Number of literals for several examples.

$cost/(N_{UR}^2)$. The sum of the literals for the results achieved by $cost/N_{UR}$ is 8% higher than the best solution, while the number for $cost/N_{UR}^2$ is 14%. If the minimum of the results of two greedy methods is used, the total of the literals will be 2270, which is 4% more than the best solution.

Other functions for f have been tried as well, but the results were not better than the two mentioned functions.

6.2 The Branch and Bound method

The Branch and Bound method can find the best solution at the expense of a lot of CPU time.

Finding the best cover begins with deleting one column of the table and adding it to the set of selected columns and recurring for the subtable until all the rows of the table have been covered. The cost of this cover is the sum of the cost of the selected columns, and it can be compared to the cost of the best cover found thus far and updated, if necessary. The algorithm continues with deleting the added column from the set of the selected columns and recurring for the resulting subtable.
Figure 6-1 shows this algorithm in pseudo code.

This algorithm considers two cases for each column (selected or deselected) and computes the cost of all the possible covers of the table to find the best one. For a table with n columns this means considering 2^n different cases, hence the required CPU time may be large.

Sometimes it is not necessary to consider two different cases for some columns, and they can be selected or deselected without losing the best result. These are:

1. If there is a column in which there is no X (empty column), it can be deselected and deleted from the table.

```
best-cost = ∞
find-cover ( table ) {
            If there is no uncovered row in the table, return.

            Choose one column of the table ( C ).

            Delete C from the table.
            Add C to the set of selected columns.
            Find the best cover for resulting subtable.
            Compute the cost of the selected columns.
            If the computed cost is less than the best-cost, update it.

            Delete C from the set of selected columns.
            Find the best cover for resulting subtable.
            Compute the cost of the selected columns.
            If the computed cost is less than the best-cost, update it.
}
```

Figure 6-1: Branch and Bound Algorithm.

2. If there is a column whose X's are the subset of another column and the column's cost is not smaller than that column (dominated column), it can be deselected and deleted from the table.

3. If there is a row which has only one X, the column associated with that X should be selected and deleted from the table.

Furthermore if there is a row (r_1, dominating row) whose X's are a superset of another row (r_2), this row can be deleted from the table because covering r_2 will automatically cover r_1 as well.

Table 6.3 shows examples of these cases. In this table, column 7 is an empty column, so it can be deleted from the table.

The X's in column 6 are a subset of the X's in column 5. If the cost of column 5 is not greater than the cost of column 6, column 6 can be deleted from the table because selecting column 5 will cover more rows at the expense of less or equal cost.

There is only one X in row 1, so column 2 should be selected and deleted from

	1	2	3	4	5	6	7	8
1	.	X	.	.	.	.	.	.
2	X	.	X	.	.	.	.	.
3	X	.	X	X	.	.	.	.
4	.	.	.	.	X	.	.	.
5	.	.	.	.	X	X	.	.
6	.	.	.	.	X	X	.	X
7	.	X	.	.	.	.	.	X

Table 6.3: Example of empty column, dominated column, single-X row, and dominating row.

the table. After that row 1 and row 7, which have been covered by selecting column 2, should be deleted from the table.

The X's of row 2 are a subset of row 3, so selecting one column to cover row 2 will cover row 3 automatically. Therefore, row 3 can be deleted from the table.

Note that deleting a dominated column can create a single-X row or dominating row in the table. For example deleting column 6, makes row 5 a single-X row.

Deleting a single-X row can create a dominated column or empty column in the table. Deleting row 1 (selecting column 2) will delete row 7 from the table which makes column 8 dominated by column 5.

Deleting a dominating row can create an empty column or a dominated column in the table. Deleting row 3 from the table makes column 4 empty.

Therefore, it is necessary to iterate several times to delete all the new cases that have been created after deleting one column or row from the table.

Because the columns associated with the cubes have only one X and usually will be selected or deselected in the last steps of the covering procedure (columns 4, 5, and 6 in Table 6.1), there is usually no dominating row in the table. For the examples shown in Table 6.2 only one example had dominating rows and even that case it only occurred 4 times. Therefore, in the final implementation of the algorithm, the search for the dominating rows has been deleted to increase the speed of the program.

To accelerate the algorithm, in each step the minimum bound of the cost of the subtable can be found, and that level of recursion can be terminated if the sum of the bound and the already selected columns is more than the cost of the best solution found so far.

The cost of the solution resulting from greedy selection can be used as the best-cost at the start. Otherwise, a large number should be used to initialize best-cost, which means in the early steps of selection, every subtable will be considered for finding the best solution.

Using these facts, the algorithm can be updated. Figure 6-2 shows the updated algorithm in pseudo code.

The speed of the search depends on:

1. The quality of the first number used for initializing the best-cost variable (i.e., the quality of the result of greedy selection).

2. Sharpness of the minimum bound.

3. Effectiveness of the selected column for recursion.

To get a more accurate number for best-cost, greedy selection with two different functions was used and the lesser of the two costs is used to initialize the best-cost.

To find the minimum bound, two different methods were used:

a) If f is the $cost/N_{UR}$ for one column of the table, $MIN(f) \times N$ is a minimum bound, where N is the number of uncovered rows of the table.

b) If f_i is the smallest $cost/N_{UR}$ for the columns associated with row i, $\sum f_i$ over all the uncovered rows is a minimum bound.

Both of these minimum bounds can be computed quickly and some parts of their computation can be merged with other parts of the algorithm to speed up other parts.

```
best-cost = cost resulted by greedy selection.
find-cover ( table ) {
        Delete single-X rows from the table.
        Delete dominated columns and empty columns from the table.
        If table has been changed by deleting one column or row repeat
        the two previous steps.

        If there is no uncovered row in the table, return.
        Find a minimum bound on the cost of the subtable.
        If the cost of the subtable and the cost of the already selected
        columns is not less than best-cost, return.

        Choose one column of the table ( C ).

        Delete C from the table.
        Add C to the set of selected columns.
        Find the best cover for resulting subtable.
        Compute the cost of the selected columns.
        If the computed cost is less than the best-cost, update it.

        Delete C from the set of selected columns.
        Find the best cover for resulting subtable.
        Compute the cost of the selected columns.
        If the computed cost is less than the best-cost, update it.
}
```

Figure 6-2: Improved Algorithm.

At first method a is used since it is faster than method b. However, if the sum of the resulting minimum bound and the cost of the selected columns is smaller than the best-cost, method b is used to find a sharper minimum bound.

The speed of the algorithm also depends on how we choose the branching column. Selecting a column with many X's, means that a lot of rows will be deleted from the table. This can lead to deleting many columns from the table as well, thus making the table very small and simplifying the following steps of recursion.
Table 6.4 shows the results achieved by the Branch and Bound algorithm for several examples.

6.3 Implementation

Because the density of the matrix is usually small, a sparse-matrix structure has been used to store the matrix. A sparse-matrix only stores nonzero elements of the matrix and each element is linked to previous and next elements in the same row and the same column. This means that in a sparse-matrix each row or column can be traversed much faster than a non-sparse-matrix. This helps to find the empty columns, dominated columns, and single-X rows quickly. Also, instead of searching all the rows to find the single-X rows after selecting or deselecting some columns, only the rows which have elements in those columns are searched. Similarly, instead of searching all the columns to find the empty columns or dominated-columns after deleting some rows, only the columns which have elements in those deleted rows are searched.

	# matrix rows	# matrix columns	Density of matrix (%)	# literals of the cover	CPU time (sec)
$ADD4_1$	135	849	4	93	2.78
$ADD4_2$	26	66	9	45	≤ 0.01
$XOR5_1$	16	166	22	28	0.02
BW_1	113	367	2	≤ 264	...
$MISEX1_1$	32	98	7	74	0.48
$MISEX1_2$	8	14	18	30	≤ 0.01
$MISEX2_1$	22	37	8	164	0.01
$MISEX2_2$	10	16	16	65	≤ 0.01
$5XP1_1$	72	279	4	≤ 159	...
$RD53_1$	32	319	12	68	3.26
$RD53_2$	16	34	10	39	0.02
INC_1	99	807	5	≤ 199	...
$B12_1$	79	222	4	106	4.6
$B12_2$	46	118	8	67	0.11
$SAO2_1$	78	722	5	184	33.33
$SAO2_2$	48	139	5	133	0.23
$VG2_1$	110	393	4	182	0.2
$VG2_2$	28	47	7	58	0.01
$VG2_3$	16	19	12	29	≤ 0.01
$RD73_1$	141	3420	6	≤ 192	...

Table 6.4: Size of the matrix and CPU time of the Branch and Bound algorithm for several examples.

Chapter 7

Multiple levels of Factorization

After the first level of factorization, multiple-output functions will be converted into the following form:

$$F_1 = \sum_{0 \leq i \leq N_1} G_{i,1} \times H_{i,1} + R_1$$

$$F_2 = \sum_{0 \leq i \leq N_2} G_{i,2} \times H_{i,2} + R_2$$

$$\vdots$$

$$F_n = \sum_{0 \leq i \leq N_n} G_{i,n} \times H_{i,n} + R_n$$

where $G_{i,j}$, $H_{i,j}$ for R_j , $0 \leq i \leq N_j$, $0 \leq j \leq n$ are new Boolean functions.

Factorization can be continued to find common factors between these new functions, and to extract the common factors to get better results. In other words, a new multiple-output function F_{new} can be defined as,

$$\begin{aligned}
F_{new_1} &= G_{i,1} \\
F_{new_2} &= H_{i,1} \\
F_{new_3} &= R_1 \\
F_{new_4} &= G_{i,2} \\
&\vdots \\
F_{new_n} &= R_n
\end{aligned}$$

and is factorized. Note that although the number of outputs of this new function is much larger than the initial multiple-output function, the Boolean function related to each output is much smaller than the initial ones. This makes the factorization of succeeding levels faster than the first level.

The factorization can be continued until there are only non-trivial kernels in the multiple-output function. Another possibility is to terminate the factorization process after several levels. This can be done to get better performance. Other factorization algorithms lack this simple capability of limiting the number of levels.

During the recursion, sometimes one factor will be found in several different levels of factorization. This means that some factors will be implemented more than once in the final circuit, which is not desired. For the following multiple-output function,

$$\begin{aligned}
F_1 &= a \times (b + c) \times (d \times (e + f + g) + h) \\
F_2 &= b' \times (e + f + g) \\
F_3 &= c' \times (e + f + g)
\end{aligned}$$

after the first level of factorization the result will be,

$$\begin{aligned}
F_1 &= a \times [1] \\
F_2 &= b' \times [2] \\
F_3 &= c' \times [2] \\
[1] &= (b + c) \times (d \times (e + f + g) + h) \\
[2] &= e + f + g
\end{aligned}$$

Factorization will be continued on the following multiple-output function,

$$[1] = (b + c) \times (d \times (e + f + g) + h)$$
$$[2] = e + f + g$$

After the second level of factorization the result will be,

$$\begin{aligned} F_1 &= a \times [1] \\ F_2 &= b' \times [2] \\ F_3 &= c' \times [2] \\ [1] &= [3] \times [4] \\ [2] &= e + f + g \\ [3] &= b + c \\ [4] &= d \times (e + f + g) + h \end{aligned}$$

In the third level of factorization the following multiple-output function will be factorized,

$$\begin{aligned} [3] &= b + c \\ [4] &= d \times (e + f + g) + h \end{aligned}$$

and the result will be,

$$\begin{aligned} F_1 &= a \times [1] \\ F_2 &= b' \times [2] \\ F_3 &= c' \times [2] \\ [1] &= [3] \times [4] \\ [2] &= e + f + g \\ [3] &= b + c \\ [4] &= d \times [5] + h \\ [5] &= e + f + g \end{aligned}$$

As one can see $e + f + g$ has been extracted two times ([2] and [5]) in different levels of factorization, once in the first level and once in the third level. Note that it is impossible to extract $e + f + g$ from F_1 in the first level of factorization, because it will lead to a solution which will be worse than the one shown, after the first level of factorization.

To prevent this, it is necessary to search the network for multiple realizations of a factor and to delete redundant ones. Although this searching and deleting process can be done just once after ending recursion, it is better to do this after each level of factorization to prevent unnecessary processing on the same data several times (i.e., factorizing the expression which has been factorized in a previous level)

Another method to delete redundant nodes is factorizing all the nodes in each level, not just new nodes. Although this approach can solve the problem, it is slow and a lot of CPU time will be wasted in an unnecessary effort of searching for common factors between functions which usually do not have common factors.

Better results can be achieved by searching the network for the expressions which are complements and deleting one of them from the network. For example, in the following multiple output function,

$$F_1 = a \times (b + c + d)$$
$$F_2 = b' \times c' \times d' \times (a + e)$$

after the first level of factorization the result will be,

$$F_1 = a \times [1]$$
$$F_2 = [2] \times [3]$$
$$[1] = b + c + d$$
$$[2] = b' \times c' \times d'$$
$$[3] = a + e$$

as one can see $b + c + d$ and $b' \times c' \times d'$ ([1] and [2]) are complements and one of them can be deleted and be expressed as the inversion of the other. This will result in the following solution which is better than the first one,

$$F_1 = a \times [1]$$
$$F_2 = [1]' \times [3]$$
$$[1] = b + c + d$$
$$[3] = a + e$$

Although extracting common factors which have two literals from two outputs, will not improve the total number of literals, it is possible to find the same factor in another level of factorization, in this case it will decrease the number of literals. Therefore, the implemented program extracts two-literal factors as well.

Chapter 8

Boolean Factorization

With slight modifications to the methods of the Chapters 4 through 7, some Boolean equalities like $a \times a' = 0$ and $a \times a = a$ can be used. To use these Boolean equalities, instead of each kernel, a range should be used.

For two functions X and Y, the result of $X \times Y$ will not be changed, if we multiply Y by a function whose ON-SET is a superset of the ON-SET of X, or we add to Y a function whose OFF-SET is a superset of the ON-SET of X. In other words,

$$\begin{aligned} X \times Y &= X \times (f_1 \times (Y + f_2)) \\ &\text{where} \quad ON - SET_{f_1} \supseteq ON - SET_X, \\ &\qquad OFF - SET_{f_2} \supseteq ON - SET_X \end{aligned}$$

Using this fact, for each co-kernel pair a range can be defined.

1. Single output functions

   For a single output function, the range of a co-kernel can be defined as,

   $$R_i = \{cokernel_i \times kernel_i,\ cokernel_i' + kernel_i\}$$

   As one can see using equalities, $a \times a' = 0$ and $a \times a = a$ will result in,

$$\begin{aligned} X_i \times R_i &= cokernel_i \times \{cokernel_i \times kernel_i,\ cokernel_i' + kernel_i\} \\ &= \{cokernel_i \times kernel_i,\ cokernel_i \times kernel_i\} \\ &= cokernel_i \times kernel_i \\ &= X_i \times Y_i \end{aligned}$$

which means this transformation has not changed the cubes which will be covered by each co-kernel pair.

2. Multiple output functions

For a co-kernel pair of a multiple output function $(\{a_0, a_1, \ldots, a_{(n-1)}\}, Y_i)$ where a_i is a set for $0 \leq i \leq (n-1)$, the range can be defined as,

$$R_i = \{kernel_i \times f_1,\ kernel_i + f_2\}$$

$$\text{where}\quad ON - SET_{f_1} \supseteq ON - SET_{a_i}\ \text{, for } 0 \leq i \leq (n-1),$$
$$OFF - SET_{f_2} \supseteq ON - SET_{a_i}\ \text{, for } 0 \leq i \leq (n-1)$$

in other words,

$$ON - SET_{f_1} \supseteq (ON - SET_{a_0} \cup ON - SET_{a_1} \cup \ldots \cup ON - SET_{a_{n-1}}),$$
$$OFF - SET_{f_2} \supseteq (ON - SET_{a_0} \cup ON - SET_{a_1} \cup \ldots \cup ON - SET_{a_{n-1}}).$$

or

$$R_i = \{\ kernel_i \times (a_0 \cup a_1 \cup \ldots \cup a_{(n-1)})\ ,\ kernel_i + (a_0' \cap a_1' \cap \ldots \cap a_{(n-1)}')\ \}$$

Chapter 9

Results

The technique described in this thesis has been implemented as an option to the SIS package. The results and run-times were collected on a DEC ALPHA 3000 machine.

Table 9.1 shows the number of literals achieved by the RIC algorithm and CPU time for several examples in comparison with number of literals achieved by two different SIS algorithms, namely, fx (an algorithm for cube and kernel extraction), and gkx (an algorithm for kernel extraction).

As one can see, the results achieved by RIC algorithm are much better than gkx. The results are sometimes better than fx, but in cases with many common cubes between different outputs, it cannot find a solution as good as the one achieved by fx. Note that for arithmetic functions RIC algorithm usually produces better results than either fx or gkx.

The RIC algorithm can be used as a preprocessor before running fx to get better results. In other words, after running RIC algorithm, fx can be used to extract cubes between different outputs.

Table 9.2 shows the number of literals achieved by running fx after the RIC algorithm for some examples in comparison with number of literals achieved by running quick-factor after fx.

Table 9.3 shows the number of literals achieved by using different methods for

	In/Out	Cubes	CPU Time (sec)	RIC # literals	gkx # literals	fx # literals
ADD4	9/5	135	16.96	55	80	73
XOR5	5/1	16	0.76	16	28	16
INC	7/9	99	8.76	190	255	176
SQUAR5	5/8	32	8.39	103	132	110
BW	5/28	87	5.06	252	317	213
MISEX1	8/7	32	1.15	79	100	72
MISEX2	25/18	29	1.12	164	184	112
5XP1	7/10	75	415.49	136	176	138
RD53	5/3	32	6.04	57	73	52
RD73	7/3	141	150.4	144	179	130
B12	15/9	82	7.21	101	122	105
SAO2	10/4	78	82.41	161	287	160
VG2	25/8	110	4.66	121	133	94
T481	16/1	481	877.62	62	74	36

Table 9.1: CPU time and the number of literals in comparison with gkx and fx.

	In/Out	Cubes	greedy with fx # literals	B and B with fx # literals	fx with Quick factor # literals
ADD4	9/5	135	54	54	66
XOR5	5/1	16	16	16	16
INC	7/9	99	190	—	171
SQUAR5	5/8	32	95	—	106
BW	5/28	87	216	—	208
MISEX1	8/7	32	72	68	70
MISEX2	25/18	29	110	117	110
5XP1	7/10	75	136	128	123
RD53	5/3	32	57	53	51
RD73	7/3	141	142	—	120
B12	15/9	82	94	96	97
SAO2	10/4	78	148	147	153
VG2	25/8	110	95	100	92
T481	16/1	481	56	—	36

Table 9.2: Number of literals using fx after RIC algorithm in comparison with using quick-factor after fx.

	In/Out	Cubes	Without Factoring # literals	Quick Factor # literals	Good Factor # literals
ADD4	9/5	135	55	55	55
XOR5	5/1	16	16	16	16
INC	7/9	99	212	190	190
SQUAR5	5/8	32	101	103	103
BW	5/28	87	250	252	252
MISEX1	8/7	32	74	79	79
MISEX2	25/18	29	164	164	164
5XP1	7/10	75	148	143	136
RD53	5/3	32	54	57	57
RD73	7/3	141	135	144	144
B12	15/9	82	100	102	101
SAO2	10/4	78	160	165	161
VG2	25/8	110	120	121	121
T481	16/1	481	100	62	62

Table 9.3: Number of literals for different methods of counting the literals of expressions.

computing the cost of kernels, intersections, and cubes on the results generated by RIC. As one can see, the differences in the results are generally small. Note that for some functions, using quick-factor and good-factor have given worse results than simply counting the literals without factoring. The reason is that this simple estimation of literals does not take into account that there will be some common factors between different expressions, and simply considers each expression separately.

Table 9.4 shows the results achieved using the minimum of the greedy algorithm with two different cost functions in comparison with the Branch and Bound algorithm. In both cases, good factor was used to factorize each expression before computing its cost. Note that for function VG2, the result achieved by the greedy method is better than the Branch and Bound method. The reason is that minimizing the number of literals in one level of factorization does not necessarily guarantee the minimization of the final result.

	In/Out	Cubes	Greedy # literals	B and B # literals
ADD4	9/5	135	55	55
XOR5	5/1	16	16	16
MISEX1	8/7	32	79	72
MISEX2	25/18	29	164	164
5XP1	7/10	75	142	136
RD53	5/3	32	57	54
B12	15/9	82	101	96
SAO2	10/4	78	161	154
VG2	25/8	110	121	126

Table 9.4: Number of literals for the greedy method and the Branch and Bound method.

9.1 Conclusion

In this thesis a new method for factorization of Boolean functions using kernel intersection has been proposed. The proposed method has the possibility of limiting the levels of factorization which does not exist in other methods. The results achieved by this method are much better than kernel extraction method implemented in the SIS package (gkx). In some cases it is better than kernel and cube extraction method implemented in the SIS package (fx).

9.2 Future work

The presented algorithms can be implemented using Binary Decision Diagram (BDD) data structures. This can halp to decrease the memory requirement and increase the speed of the method.

Bibliography

[1] R. Brayton, Gary D. Hachtel, Arthur Richard Newton, and A. L. Sangiovanni-Vincentelli, editors. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, Norwell, MA, 1979.

[2] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits, CAD-6(6)*, pages 1062–1081, November 1987.

[3] M. Dagenais, V. K. Agarwal, and N. Rumin. McBOOLE: A Procedure for Exact Boolean Minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits, CAD-5(1)*, pages 229–237, January 1986.

[4] J. Darringer, W. Joyner, L. Berman, and L. Trevillyan. Logic Synthesis through Local Transformations. *IBM Journal of Research and Development*, 25(4):272–280, July 1981.

[5] H. Fleisher and L. I. Maissel. An Introduction to Array Logic. *IBM Journal of Research and Development, 19(3)*, pages 98–109, March 1975.

[6] M. R. Garey and D. S. Johnson, editors. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[7] Gary D. Hachtel, Arthur Richard Newton, and Alberto L. Sangiovanni-Vincentelli. An Algorithm for Optimal PLA Folding. *IEEE Transactions on Computer-Aided Design of Integrated Circuits, CAD-1(2)*, pages 63–76, April 1982.

[8] S. J. Hong, R. G. Cain, and D. L. Ostapko. Mini: A Heuristic Approach for Logic Minimization. *IBM Journal of Research and Development*, 18(4):443–458, September 1974.

[9] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-Valued Minimization for PLA Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits, CAD-6(5)*, pages 727–751, September 1987.

[10] Richard L. Rudell. *Logic Synthesis for VLSI Design.* PhD dissertation, University of California, Berkeley, College of Engineering, April 1989.

[11] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, and R. K. Sequential Circuit Design Using Synthesis and Optimization. In *Proceedings of the Int'l Conference on Computer Design: VLSI in computer*, pages 328–333, October 1992.