

Portland State University

PDXScholar

Dissertations and Theses

Dissertations and Theses

5-8-1992

A New Approach to the Decomposition of Incompletely Specified Functions Based on Graph Coloring and Local Transformation and Its Application to FPGA Mapping

Wei Wan

Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds



Part of the [Electrical and Computer Engineering Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Wan, Wei, "A New Approach to the Decomposition of Incompletely Specified Functions Based on Graph Coloring and Local Transformation and Its Application to FPGA Mapping" (1992). *Dissertations and Theses*. Paper 4698.

<https://doi.org/10.15760/etd.6582>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

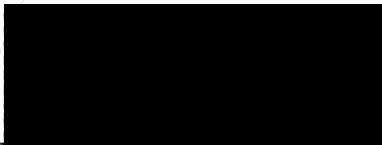
AN ABSTRACT OF THE THESIS OF Wei Wan for the Master of Science in Electrical and Computer Engineering presented May 8, 1992.

Title: A New Approach to the Decomposition of Incompletely Specified Functions Based on Graph Coloring and Local Transformation and Its Application to FPGA Mapping.

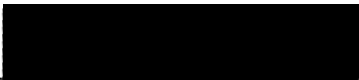
APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:



Marek A. Perkowski, Chair



Malgorzata Chrzanowska-Jeska



Martin Zwick

The thesis presents a new approach to the decomposition of incompletely specified functions and its application to FPGA (Field Programmable Gate Array) mapping. Five methods: Variable Partitioning, Graph Coloring, Bond Set Encoding, CLB Reusing and Local Transformation are developed in order to efficiently perform decomposition and FPGA (Lookup-Table based FPGA) mapping.

- 1) Variable Partitioning is a high quality heuristic method used to find the "best" partitions, avoiding the very time consuming testing of all possible decomposition charts, which is impractical when there are many input variables in the input function.
- 2) Graph Coloring is another high quality heuristics used to perform the quasi-optimum don't care assignment, making the program possible to accept incompletely specified function and perform a quasi-optimum assignment to the unspecified part of the function.
- 3) Bond Set Encoding algorithm is used to simplify the decomposed blocks during the process of decomposition.
- 4) CLB Reusing algorithm is used to reduce the number of CLBs used in the final mapped circuit.
- 5) Local Transformation concept is introduced to transform nondecomposable functions into decomposable ones, thus making it possible to apply decomposition method to FPGA mapping.

All the above developed methods are incorporated into a program named *TRADE*, which performs global optimization over the input functions. While most of the existing methods recursively perform local optimization over some kinds of network-like graphs, and few of them can handle incompletely specified functions. Cube calculus is used in the *TRADE* program, the operations are global and very fast. A short description of the *TRADE* program and the evaluation of the results are provided at the end of the thesis. For many benchmarks the *TRADE* program gives better results than any program published in the literature.

A NEW APPROACH TO THE DECOMPOSITION OF INCOMPLETELY
SPECIFIED FUNCTIONS BASED ON GRAPH COLORING AND
LOCAL TRANSFORMATION AND ITS APPLICATION
TO FPGA MAPPING

by

WEI WAN

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1992

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Wei Wan
presented May 8, 1992.



Marek A. Perkowski, Chair



Malgorzata Chrzanowska-Jeske



Martin Zwick

APPROVED:



Rolf Schaumann, Chair, Department of Electrical Engineering



C. William Savery, Interim Vice Provost for Graduate Studies and Research

ACKNOWLEDGEMENTS

A number of people have assisted me in the research reported in this thesis. I would like to take this opportunity to thank them.

I would like to thank Marek A. Perkowski, Chair of the Committee, for his initial inspiration and professional guidance throughout the research.

I would also like to thank other Committee members: Malgorzata Chrzanowska-Jeske and Martin Zwick for their assistance and numerous suggestions in the preparation of the thesis.

I am grateful to Shirley Clark for her provision of all kinds of help during my research period.

I am grateful to Ingo Schäfer for helping me to start the *TRADE* program.

I gratefully acknowledge the financial support of NSF and lastly,

I am deeply appreciative of the patience, encouragement and companionship of my wife, Hui. Without her inspiration and love, motivation would be lacking to complete this project.

TABLE OF CONTENTS

	PAGE
ACKNOWLEDGEMENTS	iii
LIST OF TABLES	vi
LIST OF FIGURES	vii
CHAPTER	
I INTRODUCTION	1
II XILINX ARCHITECTURE	4
II.1. CLBs	4
II.2. IOBs	5
II.3. Interconnect	6
II.4. Combinatorial Functions	7
III CURRENT RESEARCH VERSUS OUR APPROACH	11
III.1. Current Research on Lookup-Table Based FPGA Mapping	11
III.2. Our Approach	14
IV BASIC DEFINITIONS	16
V HOW TO PERFORM DECOMPOSITIONS?	24
V.1. Decomposition of the Incompletely Specified Functions	24
V.2. Bond Set Encoding	29
VI THREE BASIC SPEEDUP APPROACHES	35
VI.1. Graph Coloring	35
VI.2. Variable Partitioning	41
VI.3. Local Transformation	50

	v
VII PROGRAM <i>TRADE</i> AND ITS EVALUATIONS	59
VII.1. Procedure of <i>TRADE</i>	59
VII.2. CLB Merging	62
VII.3. CLB Reusing	64
VII.4. Evaluation of the Results	70
VIII CONCLUSIONS AND FUTURE WORK	73
VIII.1. Conclusions	73
VIII.2. Future Work	74
REFERENCES	75

LIST OF TABLES

TABLE	PAGE
I Time (T) vs. node Number (N) and edge Percentage (P)	38
II Comparisons among <i>TRADE</i> , <i>MIS-PGA(phase 1)</i> and <i>MIS-PGA(new)</i>	72

LIST OF FIGURES

FIGURE	PAGE
1 Physical structure of a Xilinx chip	5
2 CLB internal logic	6
3 IOB internal logic	7
4 Interconnect resources	8
5 FG mode	9
6 F mode	9
7 FGM mode	10
8 Cubes and cube sets	19
9 Intersection operation	19
10 Decomposition	20
11 Karnaugh map and decomposition chart	22
12 Example of an incompatibility graph	23
13 Curtis decomposition	25
14 Application of the Curtis decomposition to FPGA mapping	26
15 Karnaugh map, decomposition chart and the expected decomposition	27
16 Incompatibility graph	28
17 Final don't care assignment	28
18 Similarity Factor Table	30
19 Decomposed Karnaugh maps	31
20 Pseudo-code of bond set encoding	34
21 Graph coloring using Color Influence Method	37

22	Time vs. node Number and edge Percentage	39
23	Pseudo-code of graph coloring	40
24	Cube arrangements	42
25	Relative positions between two cubes	43
26	Karnaugh map of function f	43
27	Karnaugh maps for the 'best' partitions	45
28	Variable partitioning example	46
29	ON-Y, ON-N, OFF-Y and OFF-N Tables	48
30	Partitions after variable partitioning	49
31	Pseudo-code of variable partitioning	50
32	Local transformation	52
33	Application of local transformation to FPGA mapping	54
34	Four different columns	55
35	Modification Factor Table	56
36	Pseudo-code of local transformation	58
37	Before and after decomposition	60
38	Pseudo-code of <i>TRADE</i> program	61
39	Pseudo-code of CLB merging	64
40	CLB reusing concept	65
41	CLB reusing example	66
42	After graph coloring	67
43	The final coding	69
44	Pseudo-code of CLB reusing	70
45	Verifying example	71

CHAPTER I

INTRODUCTION

The modern digital logic designers have the opportunity to design a circuit from any of the six categories: Standard SSI/MSI devices, Standard LSI/VLSI devices, Gate array devices, Standard-cell devices, Full-custom devices and Programmable Logic Devices (PLDs). A PLD is a digital integrated circuit capable of being programmed to provide a variety of different logical functions. It offers a wide variety of complexities, architectures and configurations. Since they function differently depending on how they are programmed, PLDs belong to the Application-Specific Integrated Circuits (ASICs) family. PLD NRE (Nonrecurring Engineering) charges are either nonexistent or very small. Making design changes is very simple, fast and cost-free. PLD "turn-around" time (the time from design completion to having usable devices available) is very short, ranging from a few minutes to a few hours, which results in a significant reduction in the time-to-market of a product. All of those benefits have made the PLD technology receive great expectations in the electronic industry. FPGA (Field Programmable Gate Array) is one member of the PLD family, and it is the subject of this thesis.

PLDs have been around since the 1970's. In 1970 Harris introduced the first PLD device, Programmable Read-Only Memory (PROM), which had a structure of a fixed "AND" array followed by a programmable "OR" array. In 1975 Signetics introduced its Programmable Logic Array (PLA), which was in a structure of a programmable "AND" array followed by a programmable "OR" array. In 1978 MMI introduced its Programmable Array Logic (PAL), in a structure of a programmable "AND" array followed by a

fixed "OR" array. In 1985 Lattice introduced its Generic Array Logic (GAL), which was similar to a PAL, but had the electrically erasable capability. As the new devices emerged, the new features, such as programmable output polarities, feed backs, registers, buried registers, macro-cells, in-system programming, fold backs and so forth were introduced. There were so many kinds of PLD devices, and their architectures became so complex that the universal and advanced design tools were greatly in demand. In 1983 Assisted Technology released version 1.01a of its CUPL (Universal Compiler for Programmable Logic) PLD compiler. In 1984 Data I/O released its ABEL (Advanced Boolean Expression Language) PLD compiler. Logic minimization and device simulation were standard features of both compilers.

In 1985 Xilinx introduced its Logic Cell Array (LCA), now called FPGA, which had a very different architecture from the previous "AND-OR" array PLD architectures. The LCA consisted of a matrix of Configurable Logic Blocks (CLBs) surrounded by a ring of Input/Output interface Blocks (IOBs), and an interconnect network for connecting blocks. A CLB can be configured to function as one or two Lookup Tables (LUTs). With the introduction of the hardware, Xilinx released its own design tool XACT (Xilinx Advanced CAD Technology). There was also a similar kind of FPGA introduced by Actel. Instead of using LUTs, it used a multiplexer architecture in the basic logic blocks. Later on, gate array-like and PAL-like complex devices were introduced, they were also called FPGAs. The purpose of this thesis is to present the technology mapping techniques for Xilinx's lookup-table based FPGAs.

Good design tools can greatly improve the quality of the resulting circuit. For instance, the very first FPGA mapping program produced the results that the number of CLBs necessary for the MCNC benchmark examples alu2 and 9symml were 157 and 74 respectively, but the program based on the theories presented in this thesis results in 22

and 6 only. This is almost a ten times improvement. The benchmarks are same, Xilinx architecture remains unchanged. But the inherent characteristics of the input function is gradually unveiled, and the characteristics of the architecture of the chip is more perfectly matched with those of the input function. It seems there is no universal architecture that can fit all kinds of functions. Hopefully in the future there will be more new architectures to match the various kinds of logic circuits.

The next chapter presents the features of Xilinx architecture. Chapter III addresses the current research on FPGA mapping techniques and our approach. Chapter IV defines some basic terminologies that will appear in later chapters. Chapter V presents the decomposition method for incompletely specified functions and describes the bond set encoding algorithm. Chapter VI introduces several techniques related to decomposition and FPGA mapping problems. They include graph coloring, variable partitioning and local transformation. Chapter VII describes the *TRADE* FPGA mapping program and the CLB reusing algorithm used in the program, and gives the evaluation of the results. Finally, Chapter VIII summarizes our work and addresses the future work.

CHAPTER II

XILINX ARCHITECTURE

FPGA combines the high density and the versatility of gate arrays with the time-to-market advantages and off-the-shelf availability of user programmable standard parts. Xilinx's FPGA architecture [1] has an interior matrix of Configurable Logic Blocks (CLBs) and a surrounding ring of I/O interface Blocks (IOBs). Interconnect resources occupy the channels between the rows and columns of CLBs, and between the CLBs and IOBs. The functions of the CLBs, IOBs and their interconnection are controlled by a configuration program stored in an on-chip memory. The configuration program is loaded automatically from an external memory on power-up or on command, or is programmed by a microprocessor as a part of the system initialization. Figure 1 shows the physical structure of a Xilinx chip.

II.1. CLBS

The core of the FPGA device is a matrix of identical CLBs as shown in Figure 2. Each CLB contains a programmable combinatorial logic section and two storage registers. The combinatorial logic section is capable of implementing any Boolean function of its input variables. The registers can be loaded from the combinatorial logic or directly from a CLB input. The register outputs can drive the combinatorial logic directly via an internal feedback path.

II.2. IOBS

The periphery of the FPGA device is made up of user programmable IOBs as shown in Figure 3. Each block can be programmed independently to be an input, an output with 3-state control or a bidirectional pin. Inputs can be programmed to recognize either TTL or CMOS thresholds. Each IOB also includes flip-flops that can be used to buffer inputs and outputs.

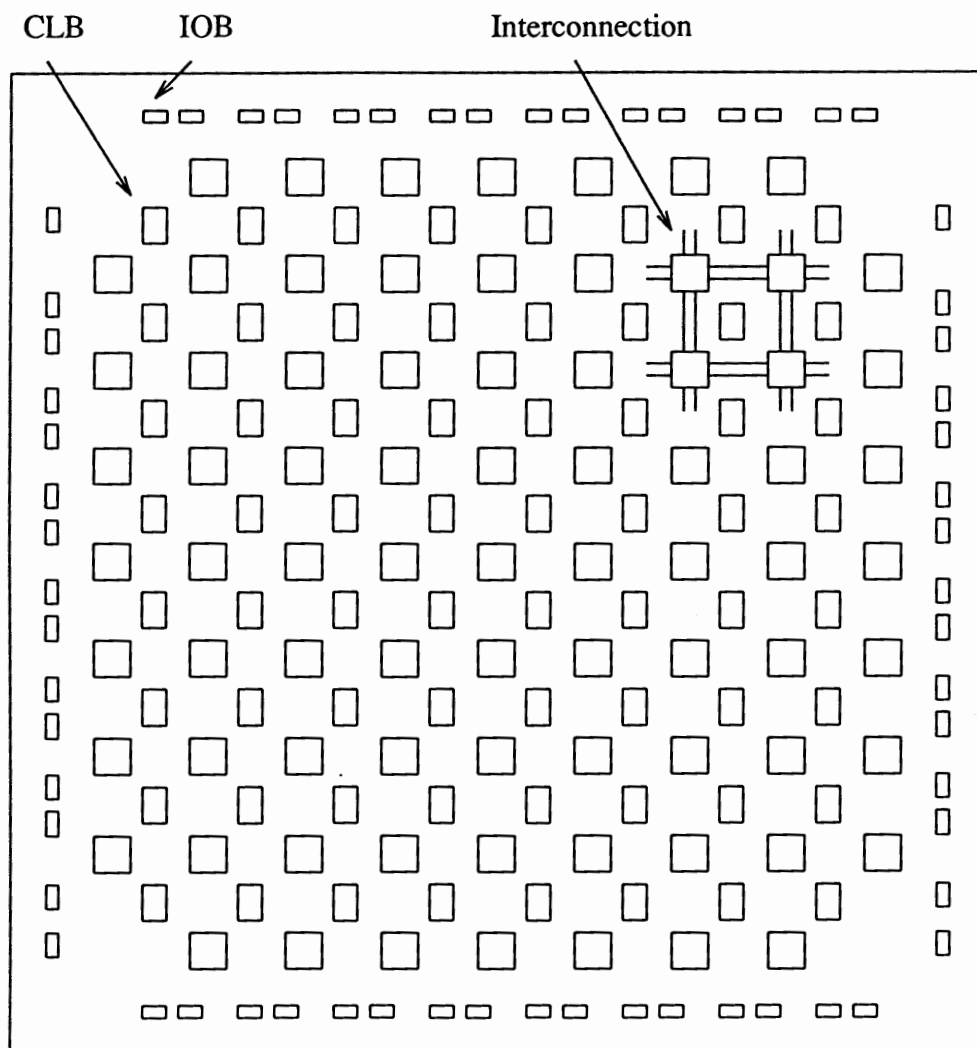


Figure 1. Physical structure of a Xilinx chip [1].

II.3. INTERCONNECT

The flexibility of the FPGA devices is due to the programmable resources that control the interconnection of any two points on the chip. The FPGA interconnection resources, as is shown in Figure 4, include a two-layer metal network of lines that run horizontally and vertically in the rows and columns between the CLBs. Programmable switches connect the inputs and outputs of IOBs and CLBs to nearby metal lines. Crosspoint switches and interchanges at the intersections of rows and columns can switch the signal from one path to another. Long lines run the entire length or breadth of the chip, bypassing interchanges to provide the distribution of critical signals with minimum delay or skew.

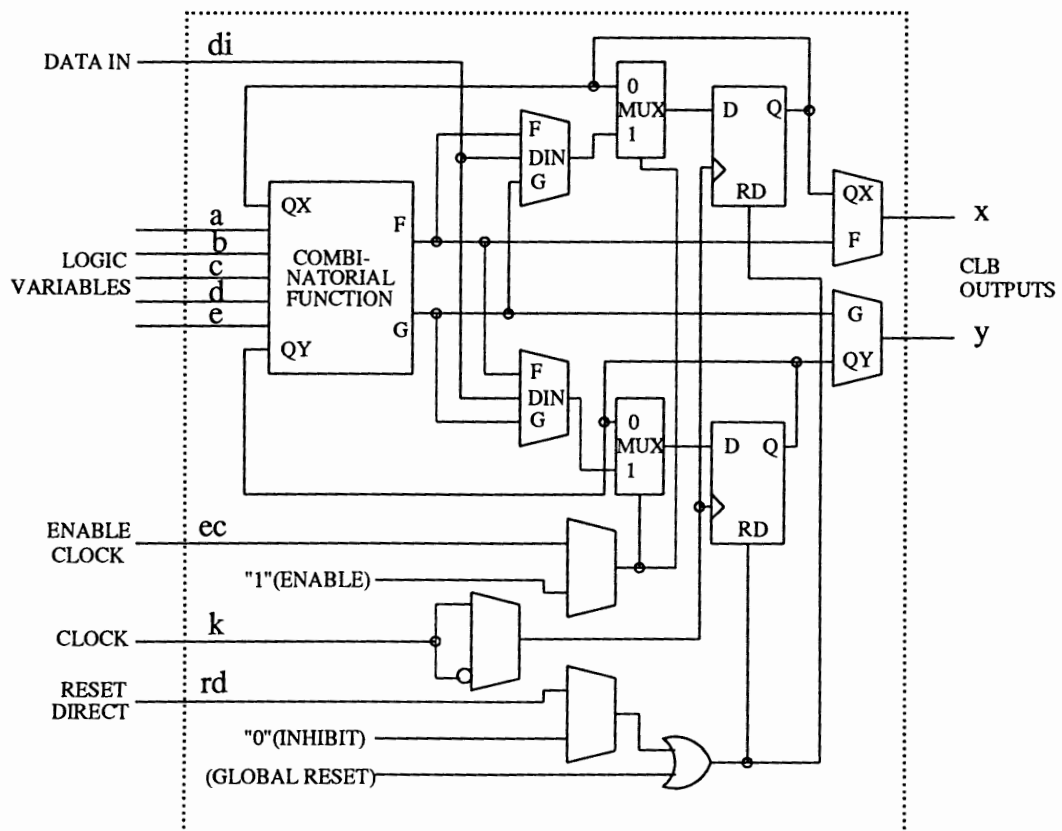


Figure 2. CLB internal logic [1].

II.4. COMBINATORIAL FUNCTIONS

The combinatorial logic portion of the CLB uses a 32 by 1 lookup-table to implement Boolean function. A K-input ($K = 5$ for Xilinx architecture) lookup-table is a digital memory with K address lines and a one-bit output. This memory contains 2^K ($2^5 = 32$) bits and is capable of implementing any Boolean function of K input variables.

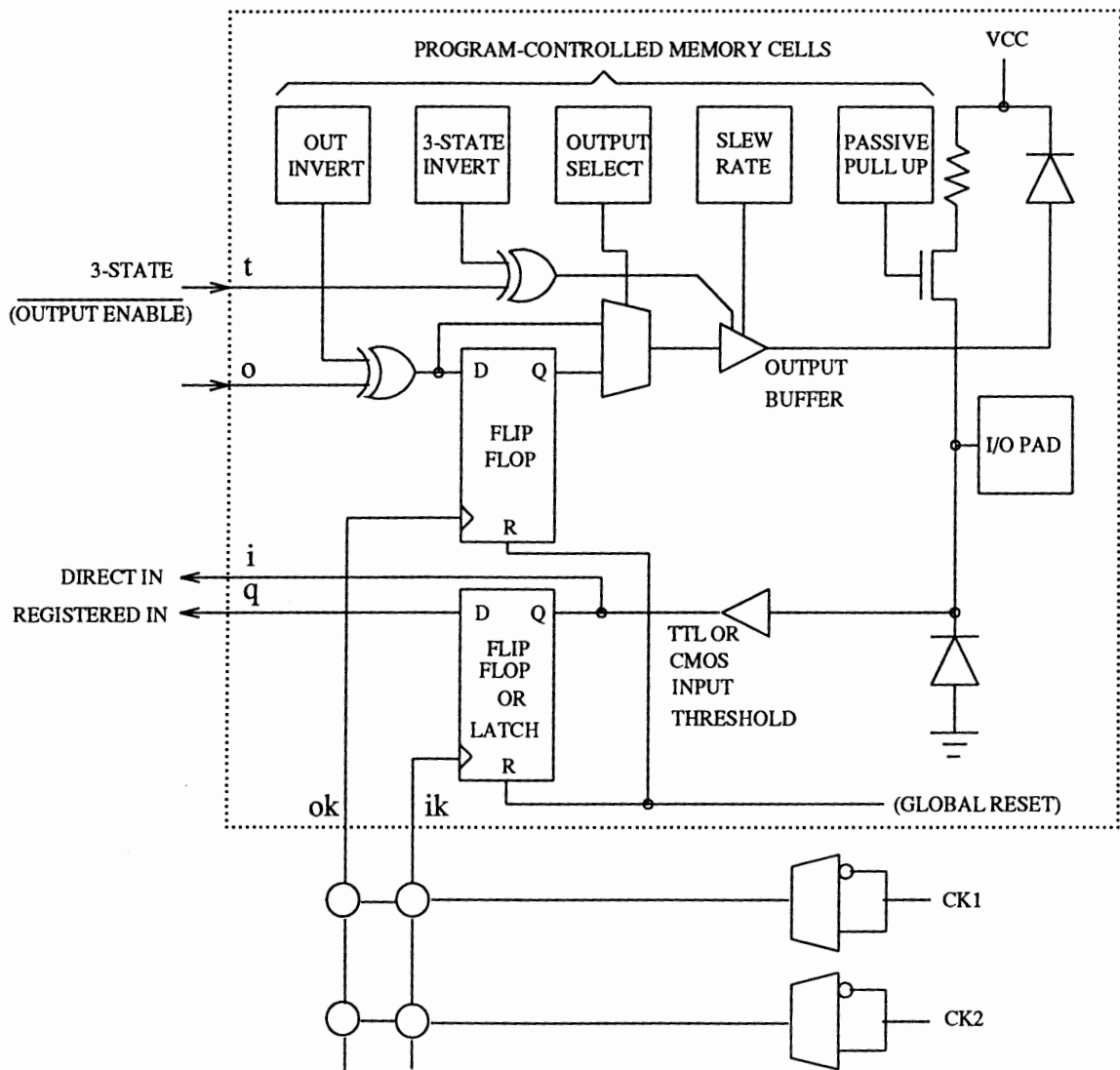


Figure 3. IOB internal logic [1].

The combinatorial logic can be configured into one of the three modes (FG, F and FGM modes).

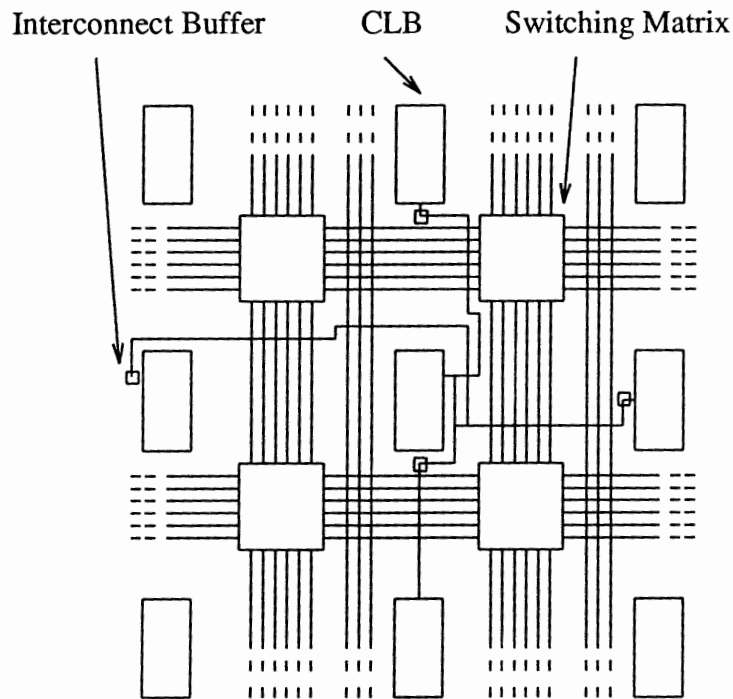


Figure 4. Interconnect resources [1].

The FG mode, as is shown in Figure 5, generates two functions of four variables each. One variable, A, must be common to both functions. The second and the third variable can be any choice of B, C, Q_x and Q_y . The fourth variable can be any choice of D or E.

The F mode, as is shown in Figure 6, generates any function of five variables: A, D, E and two choices out of B, C, Q_x and Q_y .

The FGM mode, as is shown in Figure 7, allows variable E to select between two functions of four variables: Both have common inputs A and D and any choice out of B,

C, Q_x and Q_y for the remaining two variables. This mode might then implement some

functions of six or seven variables. In Figure 7 $\begin{matrix} \boxed{M} \\ \boxed{U} \\ \boxed{X} \end{matrix}$ stands for a multiplexer.

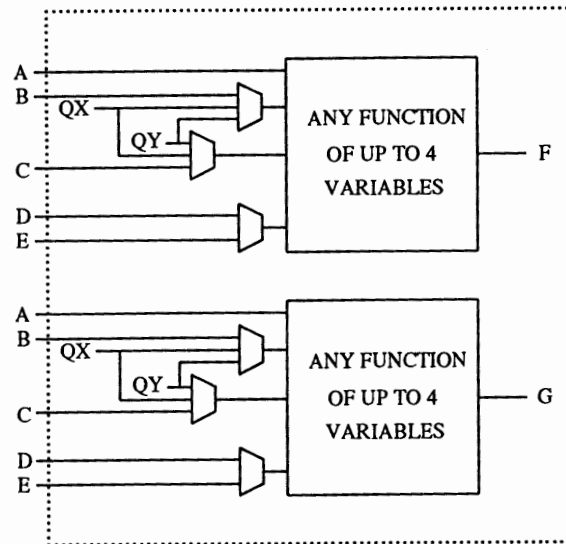


Figure 5. FG mode [1].

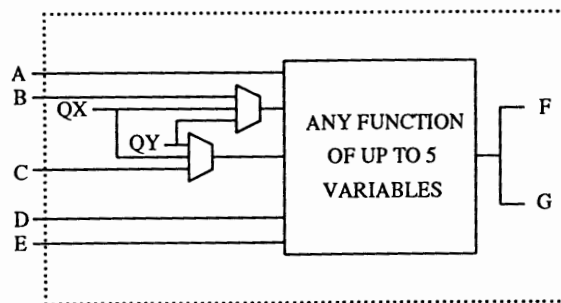


Figure 6. F mode [1].

We are concerned only with the F and FG modes in this thesis.

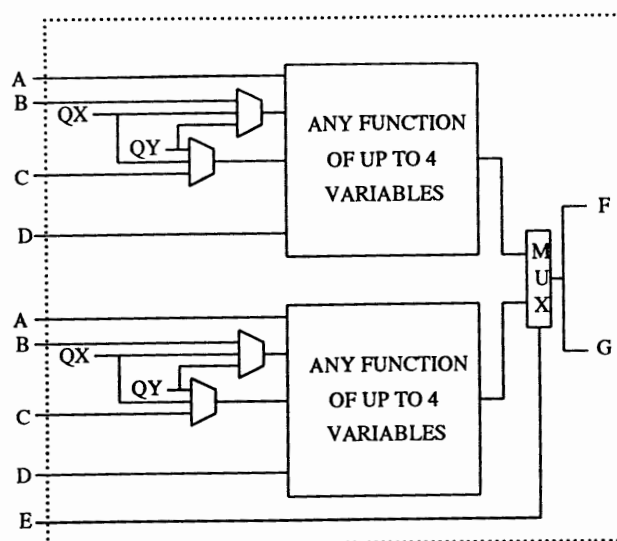


Figure 7. FGM mode [1].

CHAPTER III

CURRENT RESEARCH VERSUS OUR APPROACH

Technology mapping is a process of transforming a technology independent Boolean network (or function) into a technology-based circuit. For lookup-table based FPGAs, the technology-based circuit is a network of basic logic blocks. The basic logic block can implement any Boolean function of up to five input variables. The traditional library-based technology mapping techniques can not be used because the size of the library increases exponentially with the number of inputs of the component in the library. For example, there will be $2^{2^5} = 42,494,967,295$ components in the library if the number of inputs of each component is five.

III.1. CURRENT RESEARCH ON LOOKUP-TABLE BASED FPGA MAPPING

Several technology mapping approaches for lookup-table based FPGAs have been reported.

MIS-PGA(new) [2] starts from a tree-like network. First, it applies a variety of decomposition methods to decompose the input network into a feasible network. The *feasible network* is a network in which the number of inputs of each node is limited. For Xilinx architecture, this input number is up to five. Compared with its predecessor *MIS-PGA* [3], more decomposition techniques are incorporated. The decomposition methods that *MIS-PGA(new)* employs include *cube packing* which works well for functions with

more or less *mutually orthogonal cubes*. *Roth-Karp decomposition* is suitable to *symmetric functions* but doesn't work with nondecomposable functions. *AND-OR decomposition* can break an infeasible node into several feasible nodes. The generated feasible node is either an inverter, a two-input AND or a two-input OR gate. *Cofactoring decomposition* uses the concept of Shannon expansion to expand the network into a feasible network, in which all nodes have up to three input variables. *decomp -d** partitions the cubes of the input network into a set of cubes having disjoint variable support, and creates a node for each partition of cubes and a node which is the OR of all these partitions. The resulting subnetworks may not be feasible, and neither are those from the *kernel extraction decomposition*. Other decomposition techniques will be used to make the network feasible. After decomposition, *MIS-PGA(new)* uses a *maxflow algorithm* to generate all possible *supernodes* and solves the *binate covering problem* to minimize the cardinality of the supernode set which covers the entire network. Finally, by solving the *maximum matching problem*, it merges all possible nodes into the FG mode CLBs.

Hydra [4] uses an approach similar to *MIS-PGA*, but puts more attention on the FG mode CLBs.

Chortle-crf [5] starts from a *Directed Acyclic Graph* (DAG). It first divides the DAG into a *forest of trees*. Then, by using the *dynamic programming approach*, it carries out technology mapping on each tree to find the *minimum cost circuit*. Several techniques are used, such as *two_level decomposition* which uses a *bin packing algorithm*, *multi-level decomposition*, *exploiting reconvergent paths* and *replication of logic at fanout nodes*. These make the *Chortle-crf* get a significant improvement over its predecessor *Chortle* [6] in both the quality of solutions and the running time.

* There is no special name for this decomposition in [2].

X-map [7] converts Blif format into an *if-then-else* DAG, which is a network with the number of inputs of each node less than or equal to three. Then it goes through a *marking* and a *reduction process* to minimize the network. Finally, a simple merging algorithm is applied to merge all possible nodes into the FG mode CLBs.

VISMAP [8] introduces the concept of invisible edges. The *invisible edge* is a edge which doesn't appear in the resulting network after mapping. It starts from a feasible network (in DAG format), partitions this network into several subgraphs of reasonable size and goes through a *pre-processing* and a *main processing* step to determine the invisible edges to reduce each subgraph. A merge algorithm is used to merge all possible nodes into the FG mode CLBs.

The main objective of above FPGA technology mapping approaches is to minimize the area. There are several other approaches: *MIS-PGA(d)* [9], *Chortle-d* [10] [11] and *DAG-MAP* [12] which aim at the delay optimization.

The FPGA mapping approaches mentioned above consist, in general, of four major steps: graph construction, decomposition, reduction and packing. In the *graph construction* step, the very first step, a special kind of network-like graph or a set of subgraphs is created. The graph (or network) can be feasible or infeasible. Several specific FPGA mapping techniques will be applied to it. In the *decomposition* step, the most important step in the mapping process, a variety of decomposition methods are applied to transform an infeasible network into a feasible one. During the process, the decomposition algorithms try to minimize the number of nodes in the decomposed network as well as the number of input variables per node. In the *Reduction* step, generally more computationally expensive, some covering algorithms are applied in order to find a set of minimum number of CLBs which can cover the entire network. In the *Packing* step, according to

the specific FPGA architecture, some algorithms are used to merge the possible nodes to further decrease the area. Most of the operations used in the above four steps are local operations. The dynamic programming algorithm ensures the local operation to traverse across the network. The program is recursively invoked until a satisfactory result is reached.

III.2. OUR APPROACH

We developed several FPGA mapping techniques which apply global operations. We used a special memory storage technique to store ON and OFF sets and applied Cube calculus to them. Our program accepts incompletely specified functions and performs a quasi-optimum don't care (DC) outputs assignment. The *don't care assignment* is to assign don't care outputs as 0 or 1 to make the function more simple. With respect to the algorithms used in the program, only ON and OFF sets are stored and the DC set is not needed to process. (ON set, OFF set, DC set and Cube calculus will be explained in the next chapter). Our FPGA mapping techniques try to reconstruct the input network in such a way that:

- the decomposed network is technology feasible (for Xilinx architecture, the input variables of each node is up to five),
- the number of nodes in the network is as small as possible,
- the connections between the nodes are as simple as possible, and
- the path from the input to output, which is measured by the number of CLB layers, is as short as possible.

This approach generates circuits which fit better to Xilinx technology and have less CLBs, less connections and less layers. Thus the circuit is faster and easier to place and route.

The presented methods have the following assets:

- The input data to the program is an incompletely specified Boolean function described by the sets of ON and OFF cubes. It is the property of this method that the more DC cubes exist, the more efficient the method becomes. This makes our approach particularly powerful for strongly unspecified Boolean functions.
- The decomposition methods are specifically adapted to the lookup-table based FPGA architecture.
- A fast variable partitioning method is used to quickly find the good quality partitions for decomposition, avoiding the thorough test of all possible decomposition charts.
- In order to simplify the decomposed blocks, the column multiplicity minimization and the quasi-optimum don't care assignment are performed, they are achieved through a fast graph coloring algorithm. A bond set encoding algorithm is used to further simplify the decomposed blocks.
- A local transformation method is used to make the decomposition possible for all Boolean functions.
- A CLB reusing algorithm is used to decrease the number of CLBs used in the final mapped circuit.

CHAPTER IV

BASIC DEFINITIONS

Suppose that one intends to decompose an incompletely specified function consisting of twenty-five inputs and twenty outputs into several smaller logic blocks. This function is given in Espresso format as follows:

```
.i    25
.o    20
.ilb  i1 i2 i3 i4 i5 i6 i7 i8 i9 i10 i11 i12 i13 i14 i15
      i16 i17 i18 i19 i20 i21 i22 i23 i24 i25
.ob   o1 o2 o3 o4 o5 o6 o7 o8 o9 o10 o11 o12 o13 o14 o15
      o16 o17 o18 o19 o20
.type fr

10-01-010101-01-01010-10-    10-10010-1010-01-10-
1-11-111-1--1100000-01-1-    01-01-00101010-----1
00000001-01010101-11-0110    01-0-1-010101-1101-0
..
..
00100101010101010-1         1-110101001---010101

.end
```

Espresso format is a two-level description of the Boolean function. It is a character matrix with keywords embedded in the input to specify the size of the matrix and the logical format of the input function. In the above file:

.i 25 Specifies the number of input variables (25).

.o 20 Specifies the number of output functions (20).

`.ilb i1 i2 i25`

Gives the name of input variables. `i1` is the name of the input variable corresponding to the first column of the input cube array (left matrix of the above file), `i2` to the second column, and so forth.

`.ob o1 o2 o20`

Gives the name of output functions. `o1` is the name of the output function corresponding to the first column of the output array (right matrix of the above file), `o2` to the second column, and so forth.

`.type fr` Sets the logical interpretation of the character matrix of output array. `fr` specifies that: A 1 in the output array means that the corresponding cube in the input cube array belongs to the ON set. A 0 in the output array means that the corresponding cube in the input cube array belongs to the OFF set. The rest (-'s in the output array) means that the corresponding cube in the input cube array belongs to the DC set.

`.end` Marks the end of the input logic.

With respect to the algorithms used in the program, DC cubes are not needed to store at all in the program when the respective output function is minimized. This decreases the memory demand and becomes more efficient when there exist many DC cubes in the input function.

How do we deal with such a decomposition problem? By now there are very few of CAE tools for the general Boolean decompositions [13] [14] [15] [16]. In most of the existing systems, during the process of constructing the Truth-Tables (or Boolean equations) from the practical problems, DC outputs are mistreated as 0's (or 1's) because of

the lack of the tools that can handle DC outputs. This happens anywhere, and leads to the results which are far from the optimum ones. Let's first give the fundamental definitions.

Definition 1. *Cube*

A *Cube* is a compact expression of a set of minterms. For example, minterms 11010 and 11000 can be expressed as a cube 110-0. "-" means it takes the value of both 0 and 1.

Definition 2. *ON Cube*

If the output of a cube is 1, it is called the *ON cube*.

Definition 3. *OFF Cube*

If the output of a cube is 0, it is called the *OFF cube*.

Definition 4. *DC Cube*

If the output of a cube is - (don't care. It can be either 0 or 1), it is called the *DC cube*.

Definition 5. *ON Set*

The *ON set* is the collection of all ON cubes.

Definition 6. *OFF Set*

The *OFF set* is the collection of all OFF cubes.

Definition 7. *DC Set*

The *DC set* is the collection of all DC cubes.

Cube, ON cube, OFF cube, DC cube, ON set, OFF set and DC set are showed in Figure 8.

$$\begin{array}{l}
 \text{Cube} \\
 \downarrow \\
 \text{ON cube} \rightarrow \left. \begin{array}{l} 10110- \\ 1--000 \\ 111010 \end{array} \right\} \begin{array}{l} 1 \\ 1 \\ 1 \end{array} \leftarrow \text{ON set} \\
 \text{OFF cube} \rightarrow \left. \begin{array}{l} 0-1001 \\ 01101- \\ 0-1001 \end{array} \right\} \begin{array}{l} 0 \\ 0 \\ 0 \end{array} \leftarrow \text{OFF set} \\
 \text{DC cube} \rightarrow \left. \begin{array}{l} 01010- \\ 0100-1 \\ --0101 \end{array} \right\} \begin{array}{l} - \\ - \\ - \end{array} \leftarrow \text{DC set}
 \end{array}$$

Figure 8. Cubes and cube sets.

Definition 8. *Cube Calculus*

The *Cube calculus* is a set of operations applied to cubes. The Intersection (\bullet) operation is used in our program. Figure 9 shows the rules of Intersection operation. The ϵ is the result of the Intersection of 0 with 1 or 1 with 0. x has the same meaning as $-$.

\bullet	0	1	x
0	0	ϵ	0
1	ϵ	1	1
x	0	1	x

Figure 9. Intersection operation.

From Figure 9, the rules are: $0 \bullet 0 = 0$, $0 \bullet 1 = \epsilon$, $0 \bullet x = 0$, $1 \bullet 0 = \epsilon$, $1 \bullet 1 = 1$, $1 \bullet x = 1$, $x \bullet 0 = 0$, $x \bullet 1 = 1$, and $x \bullet x = x$. For example,

$$\begin{array}{r} 1010xx \\ 100x1x \\ \hline 10\epsilon 01x \end{array}$$

Definition 9. *Decomposition*

The *decomposition* means to decompose a large block of logic, which is difficult to analyze and implement, into several relatively smaller blocks which are easier to implement.

Figure 10 shows a general diagram of decomposition. Boolean decomposition uses Boolean representation. In some cases, decomposition is a must. For instance, some kinds of decompositions must be done in order to get a feasible network. We are not aware of any efficient tools for the general Boolean decomposition of the incompletely specified functions.

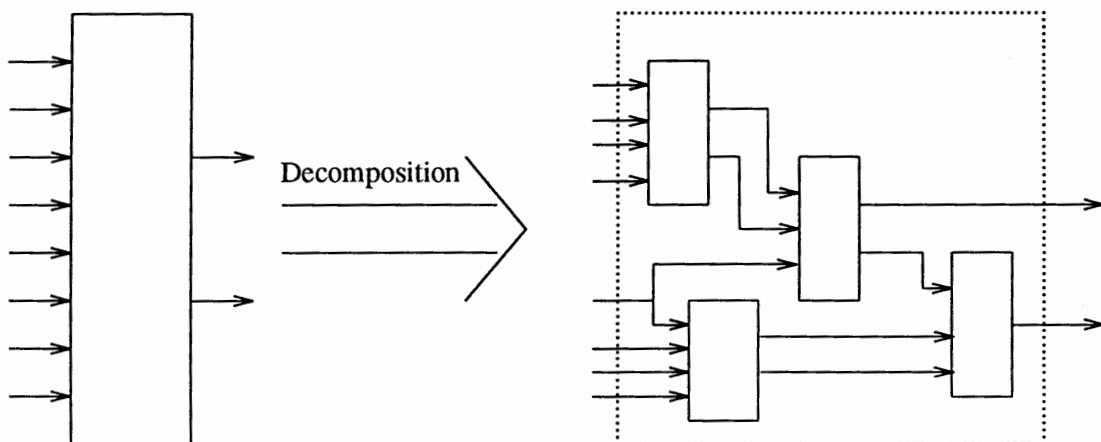


Figure 10. Decomposition.

Unfortunately, not every function is decomposable. How can we test whether a function is decomposable or not? If it is, how can we decompose it? If not, how can we make it decomposable? These questions will be answered in this thesis.

Definition 10. *Decomposition Chart*

The *decomposition chart* [17] [18] is a chart that is similar to the Karnaugh map with the only difference being that the column and row indexes of the decomposition chart are in the straight binary order, while that of the Karnaugh map are in the Gray code order.

Figure 11(b) shows an example of a decomposition chart. The corresponding Karnaugh map is shown in Figure 11(a). The column of the chart is denoted as a vector of its successive minterms. For example, column 1 in Figure 11(b) is denoted as a vector [1, 1, 0, 1]. Because there is no essential difference between the Karnaugh map and the decomposition chart, The Karnaugh map will be used instead of decomposition chart for illustration later in this thesis.

Definition 11. *Bond Set*

The *bond set* is a set of variables forming the columns of the decomposition chart. In Figure 11(b), { c, d, e } is a bond set.

Definition 12. *Free Set*

The *free set* is a set of variables forming the rows of the decomposition chart. In Figure 11(b), { a, b } is a free set.

Definition 13. *Column Multiplicity*

The *column multiplicity*, denoted by $\nu(B|A)$, is the number of different columns in a decomposition chart. In $\nu(B|A)$, B stands for the free set, A stands for the bond set.

For example, in Figure 11(b), $B = \{ a, b \}$, $A = \{ c, d, e \}$ and $v(B|A) = v(ab|cde) = 3$.

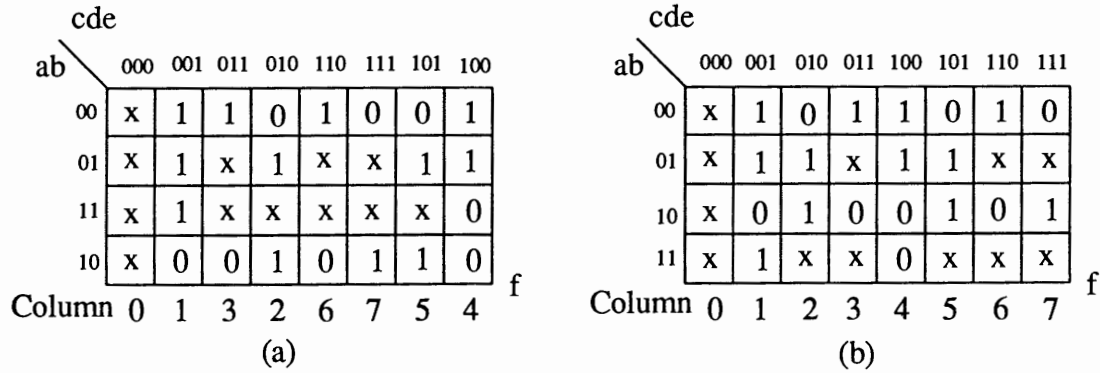


Figure 11. Karnaugh map and decomposition chart.

Definition 14. Compatible and Incompatible

If the two horizontally corresponding cells in two columns of the decomposition chart are (0,0), (1,1), (0,x), (1,x), (x,0), (x,1) or (x,x), these two cells are called *compatible*. If all the corresponding cells in two columns are compatible, these two columns are called *compatible*. Otherwise, *incompatible*.

In Figure 11(b) columns 1 ([1, 1, 0, 1]) and 6 ([1, x, 0, x]) are compatible, while columns 5 ([0, 1, 1, x]) and 6 ([1, x, 0, x]) are incompatible. In the program, we use Cube calculus to test whether two columns are compatible or not. The formula [19] to test the compatibility of two columns (columns i and j) is:

$$\& \left. \begin{array}{l} \text{ON}(i) \bullet \text{OFF}(j) = \phi \\ \text{ON}(j) \bullet \text{OFF}(i) = \phi \end{array} \right\} \implies \text{i-th and j-th column are compatible}$$

The ϕ stands for empty set. The formula states that if the Intersection of the ON set of column i and the OFF set of column j is empty, and the Intersection of the ON set of column j and the OFF set of column i is empty as well, these two columns are

compatible. Otherwise, they are incompatible.

Definition 15. *Incompatibility Graph*

The *incompatibility graph* is a graph which illustrates the relationship among columns of the decomposition chart. Each node in the incompatibility graph corresponds to a column in the decomposition chart. If two columns are incompatible, there is an edge between the corresponding nodes. If they are compatible, there is no edge. Figure 12 shows a incompatibility graph corresponding to the decomposition chart in Figure 11(b).

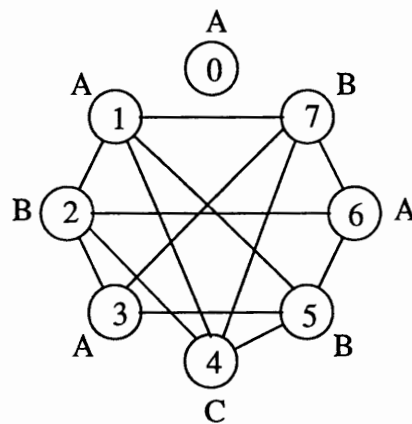


Figure 12. Example of an incompatibility graph.

In Figure 12, the number in each node (denoted by a circle) is the column number. The letter beside the circle is the color assigned to the node (column) after graph coloring. Graph coloring will be discussed in Chapter VI.

CHAPTER V

HOW TO PERFORM DECOMPOSITIONS?

In this chapter, the generalized Boolean decomposition of incompletely specified functions, and the bond set encoding algorithm are presented. The basic ideas follow [17] [18] [20] [21] and the general approach based on graph coloring is patterned after [19] [22].

V.1. DECOMPOSITION OF THE INCOMPLETELY SPECIFIED FUNCTIONS

Curtis has described the decomposition of completely specified functions in [18] [20]. Curtis proved the *fundamental theorem*:

$$v(B | A) \leq 2^k$$

$$\Leftrightarrow$$

$$f(A, B) = F[\phi_1(A), \phi_2(A), \dots, \phi_k(A), B]$$

It states that if the column multiplicity $v(B|A)$ (under the partition of the bond set A and free set B) is less than 2^k , then the function $f(A, B)$ can be decomposed into the form:

$$f(A, B) = F[\phi_1(A), \phi_2(A), \dots, \phi_k(A), B]$$

The graph representation of this theorem is shown in Figure 13. From Figure 13 we observe that, after decomposition, the big block f is broken into several smaller sub-

blocks $\phi_1, \phi_2, \dots, \phi_k$ and F . If we restrict the variable number in the bond set A to be less than or equal to five, $\phi_1, \phi_2, \dots, \phi_k$ can be implemented by CLBs of the Xilinx chip. If we further decompose subblock F until the input variables of each subblocks are less than or equal to five, the function f would be realized by the Xilinx chip. This is shown in Figure 14.

The generalization of the Ashenhurst decomposition for incompletely specified functions based on proper graph coloring was presented in [19]. Perkowski first used graph coloring to minimize the column multiplicity, then used multiplexers to realize the circuit.

The essential problem of the decomposition of incompletely specified function is how to assign DC outputs as 0 or 1 to minimize the column multiplicity. Because the number of colors in a properly colored incompatibility graph is the same as the number of different columns (column multiplicity) in a decomposition chart [19], we can transfer

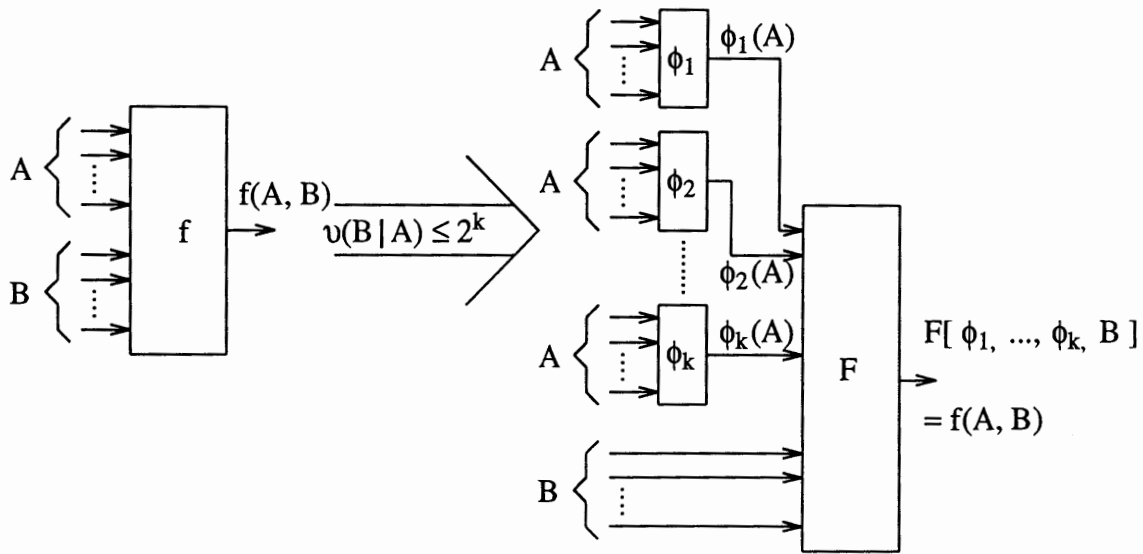


Figure 13. Curtis decomposition.

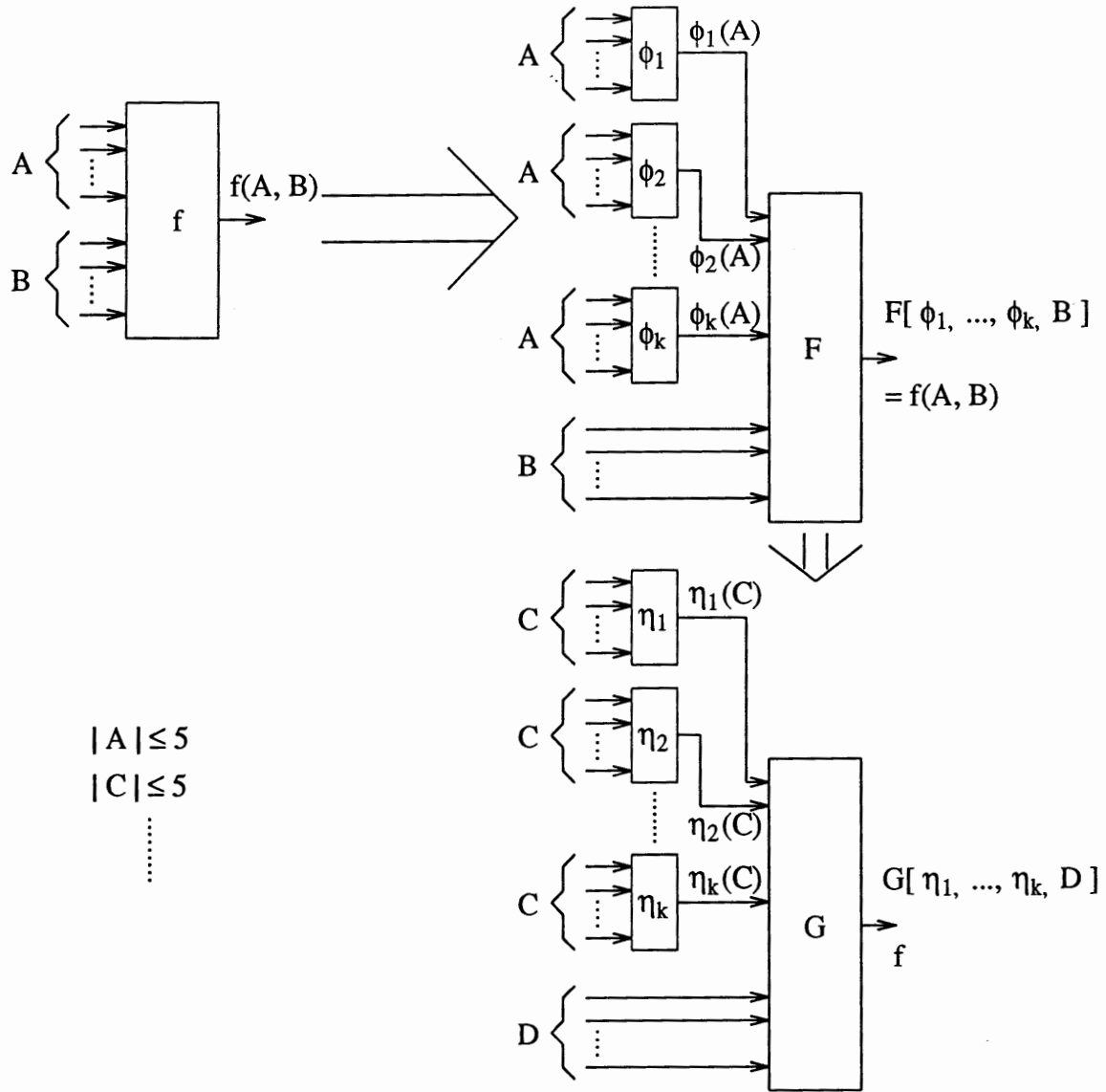


Figure 14. Application of the Curtis decomposition to FPGA mapping.

the problem of finding the smallest column multiplicity into the one of performing the proper graph coloring to find the smallest number of colors. We use the following criterion:

Set a expected number (n) of output variables from the bond set, n is less than the number of variables in the bond set. If the column multiplicity is equal to or less

than 2^k , and k is less than or equal to n , the decomposition is *successful* (or the function is *decomposable*) for this bond set under this expected value of n . Otherwise, the function is *nondecomposable* for this bond set under this expected value of n .

After a successful decomposition, the number of input variables of each subfunction (decomposed blocks, like $\phi_1, \phi_2, \dots, \phi_k$ and F in Figure 13) is decreased, and the complexity of each subfunction is decreased as well. This will be illustrated with an example. Figure 15(a) is the Karnaugh map of function f with don't care outputs. We intend to decompose the function f into several subfunctions (may be L, M and N as shown in Figure 15(c)) with the input variables of each subfunction less than or equal to four.

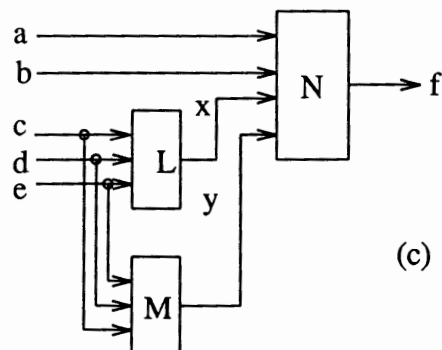
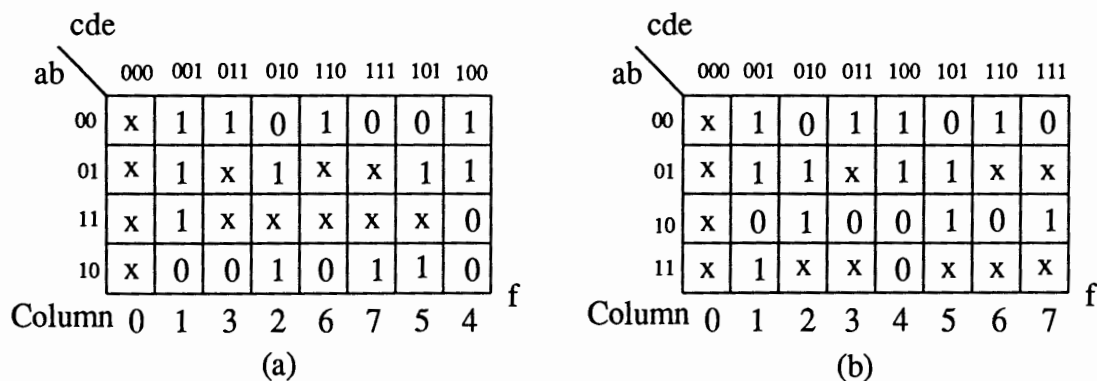


Figure 15. Karnaugh map, decomposition chart and the expected decomposition.

According to the rules presented above, the incompatibility graph is created as shown in Figure 16. After graph coloring, three colors (which means $v = 3$) which group nodes as $A = \{0, 1, 3, 6\}$, $B = \{2, 5, 7\}$ and $C = \{4\}$ are obtained. The columns with the same color are combined horizontally by the rules: $(0, 0) \rightarrow 0$, $(0, x) \rightarrow 0$, $(x, 0) \rightarrow 0$, $(1, 1) \rightarrow 1$, $(1, x) \rightarrow 1$, $(x, 1) \rightarrow 1$ and $(x, x) \rightarrow x$. For example, columns 0, 1, 3 and 6 in Figure 15(a) are combined and replaced by a new vector $[1, 1, 1, 0]$ as shown in the final don't care assignment in Figure 17.

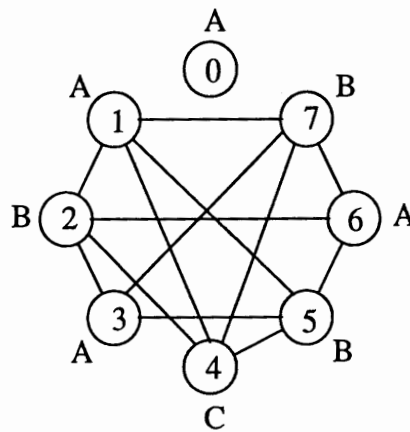


Figure 16. Incompatibility graph.

		cde							
ab		000	001	011	010	110	111	101	100
	00	1	1	1	0	1	0	0	1
	01	1	1	1	1	1	1	1	1
	11	1	1	1	x	1	x	x	0
	10	0	0	0	1	0	1	1	0
Column		0	1	3	2	6	7	5	4
Color		A	A	A	B	A	B	B	C

Figure 17. Final don't care assignment.

In the above example, we have chosen the variables a and b as the free set and variables c , d and e as the bond set. This partition results in a successful decomposition in sense of the column multiplicity less than or equal to three. How to chose the partition will be discussed in Chapter VI. In Figure 15(c), x and y are the encoded outputs of the bond set, two variables are enough for three different columns ($3 \leq 2^2 = 4$).

V.2. BOND SET ENCODING

There are many methods [23] [24] to implement the decomposed blocks (blocks L, M and N in Figure 15(c)). Here the author introduces an algorithm to encode the bond set. This algorithm aims at simplifying the block N. Block L and M will be implemented by CLBs. It doesn't matter then how complex these two blocks are as long as the number of their input variables are less than or equal to four (we assume that the CLBs have up to four inputs for this example). The encoding algorithm assigns adjacent codes (Gray code) to the similar columns. This increases the number of large cubes in the block N. The similarity between two columns is measured by the so-called Similarity Factor. The more similar the two columns, the lower the value of the Similarity Factor. The *Similarity Factor* is the number of minterms which cause the two columns not identical. The Similarity Factor between the i -th and j -th columns is:

$$\text{Similarity Factor} = \text{minterm}(\text{ON}(i) \bullet \text{OFF}(j)) + \text{minterm}(\text{OFF}(i) \bullet \text{ON}(j))$$

In equation, "minterm()" calculates the number of minterms. " $\text{ON}(i) \bullet \text{OFF}(j)$ " is the Intersection of array of ON cubes of i -th column with array of OFF cubes of j -th column. " $\text{OFF}(i) \bullet \text{ON}(j)$ " is the Intersection of array of OFF cubes of i -th column with array of ON cubes of j -th column.

In this example, bond set $\{c, d, e\}$ forms eight columns as shown in Figure 15(a). After graph coloring, it is found that there are only three different columns out of these eight columns. These three columns are with the vector $[1, 1, 1, 0]$ corresponding to color A, $[0, 1, x, 1]$ corresponding to color B and $[1, 1, 0, 0]$ corresponding to color C as shown in Figure 17. We introduce two new variables x and y to encode the bond set $\{c, d, e\}$. First calculate the Similarity Factors. The Similarity Factor between columns corresponding to color A and B has a value of 2. The value of the Similarity Factor between columns corresponding to color A and C is 1. And the value of the Similarity Factor between columns corresponding to color B and C is 2. A Similarity Factors Table is created as shown in Figure 18.

	B	C
A	A-B 2	A-C 1
	B	B-C 2

Figure 18. Similarity Factor Table.

Because the Similarity Factor between columns corresponding to color A and C is smaller (with a value of 1), these two columns are put in adjacent position as shown in Figure 19(c). Code the column corresponding to color C as 00 , the column corresponding to color A as 01 and the column corresponding to color B as 11 as shown in Figure 19(c), which is the Karnaugh map of the block N. Color A has the code 01 , this means that x is equal to 0 and y is equal to 1 for all columns with the color A. These columns are 000, 001, 011 and 110 in Figure 17, therefore the cells 000, 001, 011 and 110 of the Karnaugh map in Figure 19(a), which is the Karnaugh map of the block L, are filled with

0 because x is equal to 0. The same cells in Figure 19(b), which is the Karnaugh map of the block M, are filled with 1 because y is equal to 1. The same way, color B has the code 11, this means that both x and y are equal to 1. Columns 010, 111, and 101 correspond to color B, therefore the cells 010, 111 and 101 of the Karnaugh maps in both Figure 19(a) and (b) are filled with 1. Color C has the code 00, both x and y are equal to 0, column 100 correspond to color C, the cell 100 of the Karnaugh maps in both Figure 19(a) and (b) are filled with 0.

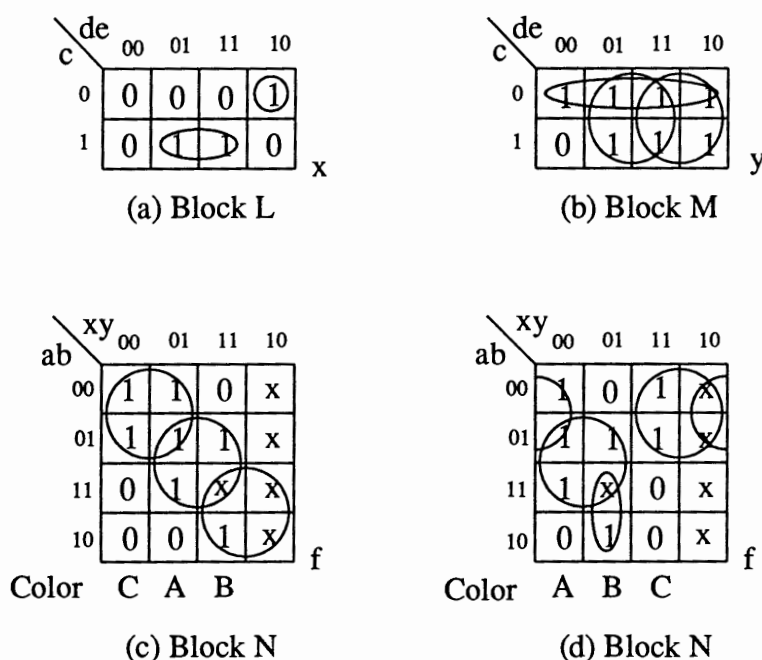


Figure 19. Decomposed Karnaugh maps.

Two variables can encode up to four columns ($2^2 = 4$). There are only three columns, corresponding to color A, B and C, that need to be encoded in our example. We fill the remaining column (column 10 in Figure 19(c)) with don't cares (*DC column*). The existence of this newly introduced DC column will further simplify the block N. This example illustrated that even the input function is completely specified, our algo-

rithm may introduce DCs in the middle of the process, which are very useful for the simplification of the later stages.

Figure 19(d) shows a Karnaugh map of an alternative implementation for the function f . Which uses the natural order of the colors. Clearly, the Karnaugh map in Figure 19(c) is more simple than that in Figure 19(d). The pseudo-code for bond set encoding is shown in Figure 20. The Blif format of the result is as follows:

```
.model example
.inputs a b c d e
.outputs f
.names c d e x
  1 - 1 1
  0 1 0 1
.names c d e y
  0 - - 1
  - - 1 1
  - 1 - 1
.names a b x y f
  0 - 0 - 1
  - 1 - 1 1
  1 - 1 - 1
.end
```

Blif format is a multi-level description of the Boolean network. Each node in this representation has a single output. Therefore, each net (or signal) has only a single driver, and one can therefore name either the signal or the gate which drives the signal without ambiguity.

`.model example` Specifies the name of the model (example).

`.inputs a b c d e` Gives the name of the input variables (a, b, c, d, e).

`.outputs f` Gives the name of the output function (f).

.names c d e x With the following ON set describes the logic of a node (subblock L in Figure 15(c)). The input variables to this node are c, d, e, and the output variable is x.

.names c d e x With the following ON set describes the logic of a node (subblock M in Figure 15(c)). The input variables to this node are c, d, e, and the output variable is y.

.names a b x y f With the following ON set describes the logic of a node (subblock N in Figure 15(c)). The input variables to this node are a, b, x, y, and the output variable is f.

.end Marks the end of this model.

```

bond_set_encoding( )
{
    create Similarity Factor Table;
    sort Similarity Factor Table in increasing order;

    l_c = one column of the column pair at position 0 of the queue;
    mark l_c as used;
    r_c = another column of the column pair at position 0 of the queue;
    mark r_c as used;
    put l_c and r_c in line;      /* l_c at left, r_c at right */
    c_n = 2;

    while ( c_n < column_multiplicity )
    {
        for ( i = 1; i <  $\frac{\text{column\_multiplicity} * (\text{column\_multiplicity} - 1)}{2}$ ; i++ )
        {
            if ( (c_i = one of the pair at position i) == l_c )
            {
                mark c_i as used;
                put c_i at the left of l_c;
                l_c = c_i;
                c_n++;
                break;
            }
            else if ( (c_i = one of the pair at position i) == r_c )
            {
                mark c_i as used;
                put c_i at the right of r_c;
                r_c = c_i;
                c_n++;
                break;
            }
        }
    }
}

```

Figure 20. Pseudo-code of bond set encoding.

CHAPTER VI

THREE BASIC SPEEDUP APPROACHES

There are three fundamental problems in the efficient implementation of the FPGA mapping program which is based on the Boolean decomposition of incompletely specified function:

- How to choose the bond set to minimize the column multiplicity?
- How to minimize the column multiplicity for a given bond set and
- how to transform a nondecomposable function into a decomposable one?

These questions will be discussed in this chapter.

VI.1. GRAPH COLORING

We have reduced the problem of finding the smallest column multiplicity to the one of performing proper graph coloring with the minimum number of colors. *Graph coloring* [25] [26] [27] [28] is one in which every two nodes linked by an edge are assigned different colors. *Minimum graph coloring* is one with the minimum number of colors.

Graph coloring is an NP-hard problem. There has been a substantial research on it in order to find the algorithms for a quasi-optimum solution with the fastest possible speed. The author presents here a fast graph coloring method. This method has been

programmed and tested on many examples, it resulted in excellent colorings. The method found exact colorings for graphs in [25], and even found better coloring than that that was claimed to be minimal in the book. We call our method the "*Color Influence Method*".

The main idea of this method is to evaluate the influence of the color assignment to a node over the entire graph, and chose the color which results in a minimum influence. The *minimum influence* means that the color assignment to a node will produce a minimum increase of color-in-bar's. The *color-in-bar's* (restrictions) are the colors that the node cannot be assigned with, which are denoted by \bar{A} , \bar{B} ..., \bar{AB} , \bar{AC} , ... as in Figure 21. After each color assignment to a node, the complexity of the graph is decreased. This is a greedy method with global evaluation. The next example is used to illustrate this method.

Figure 21(a) shows a graph that need to be colored. Start from the node with the most number of edges, that is node 2, assign color A to it. This color assignment results in that nodes 1, 3, 5 and 6 cannot be assigned with color A, denote this restriction on those nodes by color-in-bar \bar{A} 's, and remove all corresponding edges as shown in Figure 21(b). Then, color the node with the most number of color-in-bar's. If there are more than one node with the same number of color-in-bar's, chose the node with the most number of edges. If there are still more than one node, evaluate the influence of each color assignment, and assign the node with a color which results in a minimum influence. If a node can be assigned with more than one color, the evaluation of the influence of each color assignment is also required. According to the rules stated above, nodes 5 and 6 are selected because they have the same number of color-in-bar's and the same number of edges. Assigning color B to node 6 will result in a restriction \bar{AB} on node 1 and a restriction \bar{AB} on node 5. While assigning color B to node 5 will result in a restriction \bar{AB}

on node 6 and a restriction \bar{B} on node 4. Because one $\bar{A}\bar{B}$ restriction and one \bar{B} restriction result in less influence than two $\bar{A}\bar{B}$ restrictions, assigning a color to node 5 produces less influence than assigning a color to node 6. Node 5 is selected, give it color B as shown in Figure 21(c). The same way, assign color C to node 6 as shown in Figure 21(d), color B to node 1 as shown in Figure 21(e). Nodes 3 and 4 are in the same condition, and have the same influence to the graph. If node 3 is assigned with color B, node 4 can be assigned with color A or C. The final color assignment is shown in Figure 21(f).

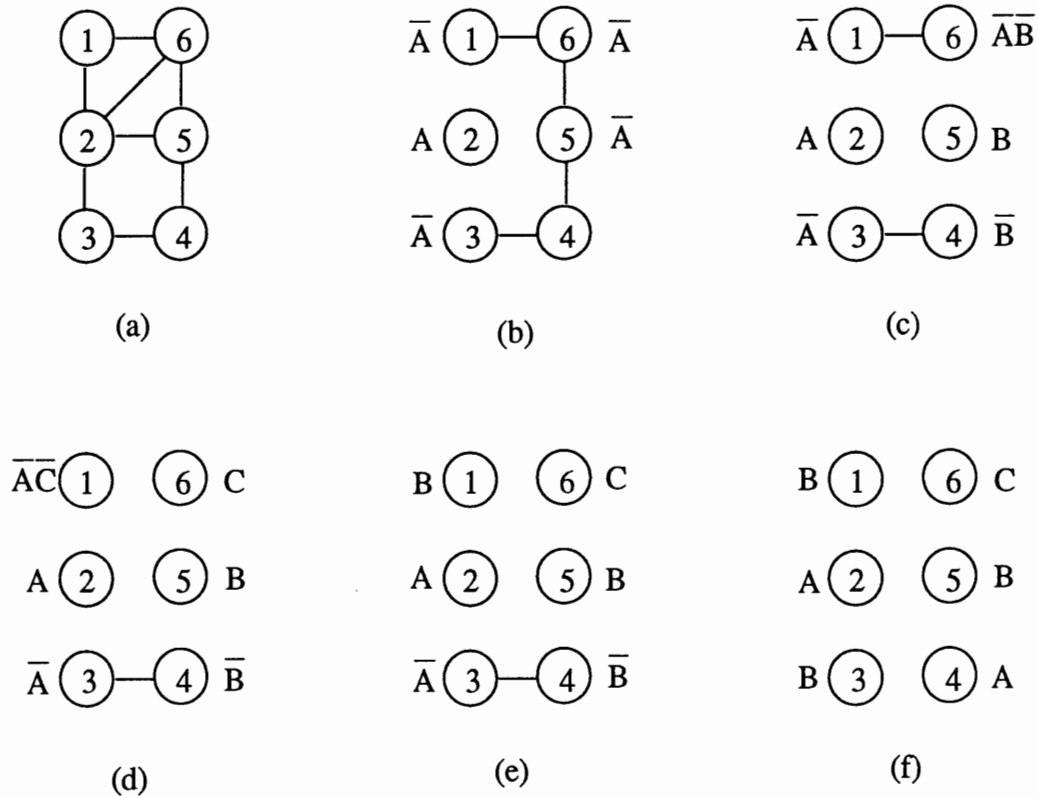


Figure 21. Graph coloring using Color Influence Method.

The above algorithm was incorporated into a program, named *COLOR*, and was run on a networked SUN 4/670MP Workstation. The results are listed in Table I. The program was tested on graphs with different number of nodes ($N = 100 \rightarrow 1000$) and

different edge percentages ($P = 10\% \rightarrow 90\%$). The maximum number of edges in a graph is $\frac{N(N-1)}{2}$, N is the number of nodes in the graph. The edge percentage (P) is simply the percentage of this maximum number of edges. Edges in the graph are randomly generated.

TABLE I
TIME (T) VS. NODE NUMBER (N) AND EDGE PERCENTAGE (P)

N → ↓ P		100	200	300	400	500	600	700	800	900	1000
10%	T	0.3	1.9	5.2	10.4	18.6	28.6	42.9	59.9	81.8	107.9
	C	6	9	12	14	16	19	21	23	25	28
20%	T	0.5	3.0	8.0	16.9	29.2	45.5	68.2	97.2	132.3	170.4
	C	9	15	19	22	28	32	36	41	44	48
30%	T	0.7	4.0	10.7	22.1	39.2	62.9	93.6	133.7	181.4	238.1
	C	12	20	26	34	39	45	51	58	63	69
40%	T	0.9	4.9	14.5	28.4	51.0	80.7	123.8	174.7	232.1	300.5
	C	16	24	36	44	52	61	70	76	84	91
50%	T	1.1	6.1	16.6	35.6	62.1	100.6	154.4	213.1	294.5	393.5
	C	20	33	44	56	67	77	88	97	108	117
60%	T	1.3	7.5	20.9	43.4	78.8	122.9	186.5	268.0	368.8	480.6
	C	23	37	52	68	81	94	107	120	135	147
70%	T	1.5	9.0	25.0	52.2	93.9	148.9	222.1	322.5	441.8	579.0
	C	27	47	67	84	104	118	136	154	164	186
80%	T	1.9	11.2	31.3	66.9	116.4	194.8	295.3	413.5	558.0	748.5
	C	34	61	85	104	129	149	172	194	212	231
90%	T	2.4	14.7	42.8	86.0	164.3	262.7	389.1	561.3	766.2	1003.2
	C	45	79	111	136	169	198	230	255	279	306

In Table I, N is the number of nodes in the graph. P is the edge percentage. T is the running time of the program which is measured by *time* command of UNIX system. The unit of T is second. C is the number of colors after graph coloring. Figure 22 shows a graphical representation of this table.

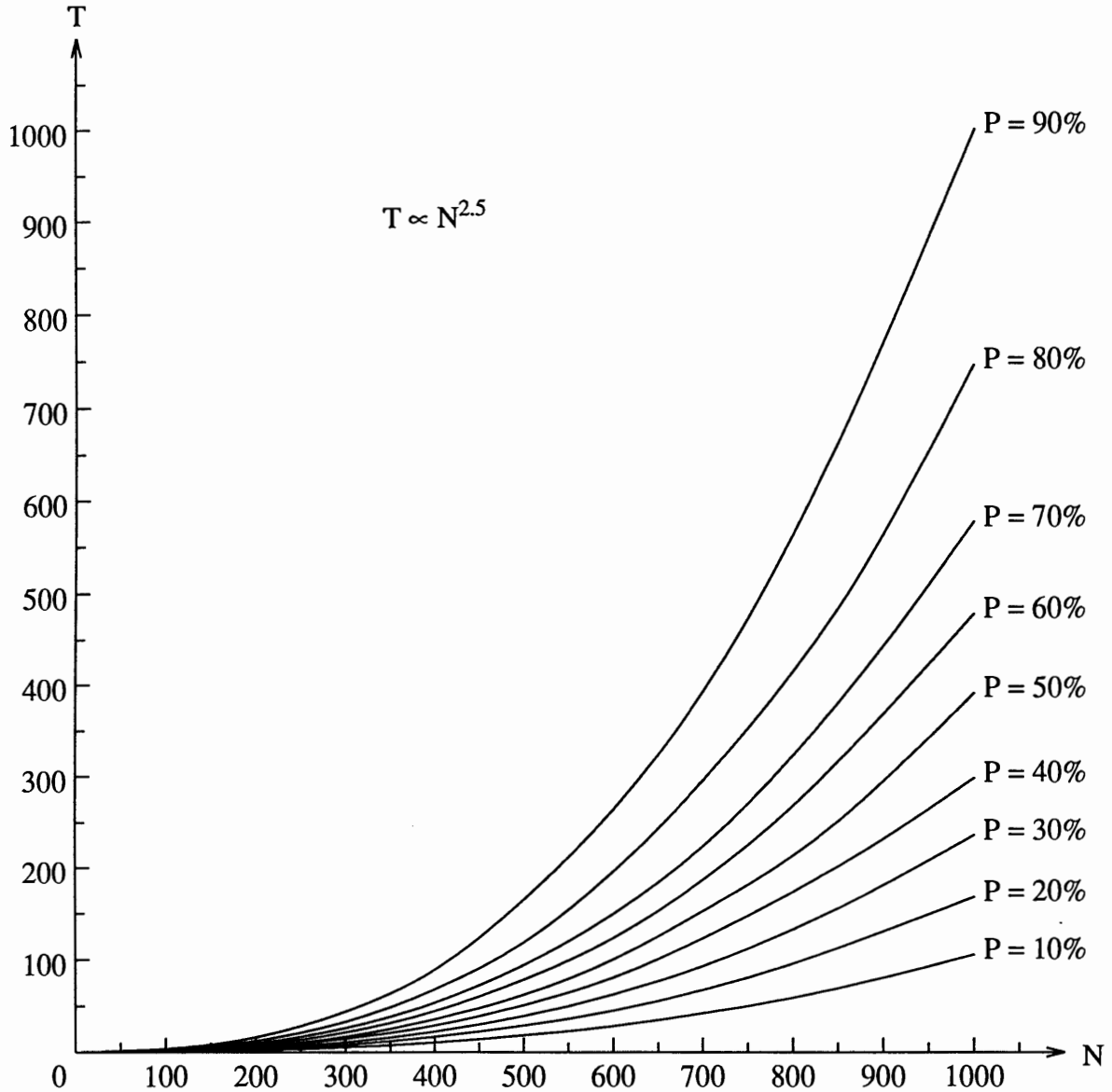


Figure 22. Time vs. node Number and edge Percentage.

By statistic analysis, it is found that the time (T) is proportional to the number of nodes (N) in a polynomial form $T \propto N^{2.5}$. So, we solve the graph coloring problem in polynomial time (not in exponential time). For small graphs, we are able to verify that the algorithm gives the minimum solutions. Therefore we hope that it gives good results for larger graphs as well. But we are not able to verify this claim since we couldn't access an exact minimal optimizer. The pseudo-code for graph coloring is shown in Figure 23.

```

graph_coloring( )
{
    sort nodes in decreasing order by the number of edges;

    color first node;
    remove it;
    mark restrictions;
    remove edges;

    while ( (cib_set = largest_number_color_in_bar( node_queue )) != empty )
    {
        e_set = largest_number_edge( cib_set );

        if ( |e_set| > 1 )
            f_n = minimum_influence( e_set );
        else
            f_n = e_set;

        color f_n;
        remove f_n;
        mark restrictions;
        remove edges;
    }
}

```

Figure 23. Pseudo-code of graph coloring.

VI.2. VARIABLE PARTITIONING

Variable partitioning is the separation of the input variables into two sets, the bond set and the free set. Each partition corresponds to an individual decomposition chart which is going to be used to calculate the column multiplicity. In order to find the decomposition that corresponds to the smallest column multiplicity, one needs to go through all possible decomposition charts. If there are total m input variables and n variables in the bond set, the number of all possible partitions is $\binom{m}{n}$. For example, if $m = 64$ and $n = 5$, then $\binom{64}{5} = 7,624,512$. If the time required to calculate the column multiplicity of a decomposition chart is 0.01 second, one would need more than 20 hours to complete all calculations. This 20 hours will be repeated thousands of times to get the FPGA mapping done. Therefore, it is impractical to try all possible partitions to find the best one.

Here the author presents a method called the *Pair Weighting Method* to quickly find the "best" partitions. This heuristic method will produce as many as four "best" partitions which are to be used for decomposition.

The basic idea of this method is to arrange cubes (minterms) in the Karnaugh map in such a way that they become more concentrated in either certain columns or rows, like the arrangements in Figure 24(a) and (b), but not like that in Figure 24(c).

For a given function, the number of minterms is fixed, the number of cubes after minimization is fixed as well. There are six minterms in Figure 24(a). Under that partition, the minterms are concentrated in two columns. In Figure 24(b) it is the same function. Under that partition, the minterms are concentrated in two rows. In Figure 24(c) it is the same function again, but the minterms are diverged across the Karnaugh map.

Clearly, the column multiplicities under the partitions shown in Figure 24(a) and (b) are less than that in Figure 24(c).

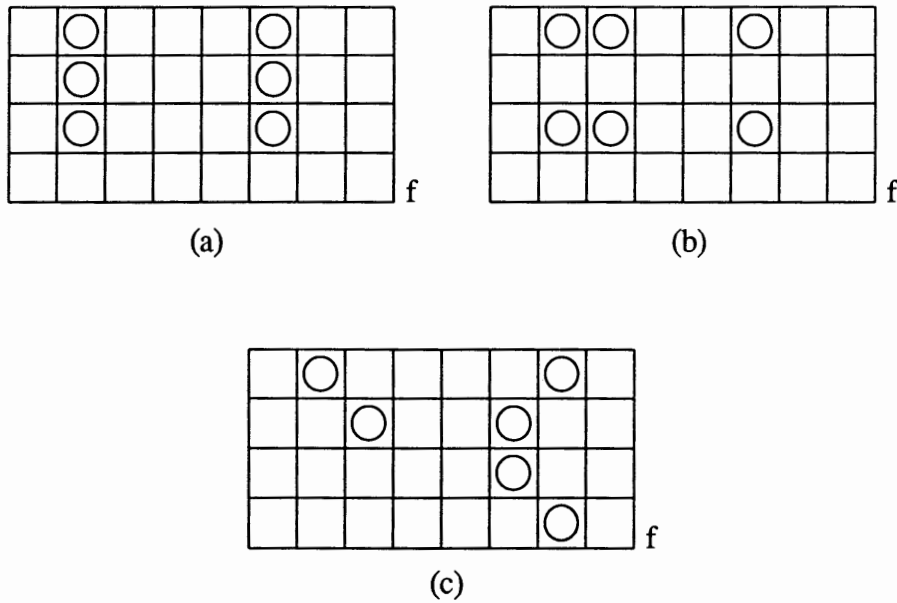


Figure 24. Cube arrangements.

How can we put more cubes in some certain columns or rows? We first analyze the relative positions between two cubes in the Karnaugh map. There are four possible relative positions between two cubes as shown in Figure 25. The rectangles stand for cubes.

The shaded areas from both cubes are the possible parts that can be put into the same columns or rows in the Karnaugh map. The question then arises, under what partition do the shaded areas from both cubes reside in the same columns or rows? The next example is used to show how to find that partition. The ON set of function f in Figure 26 is:

	abcd
cube1	0-0-
cube2	1110

The remaining cubes belong to the OFF set.

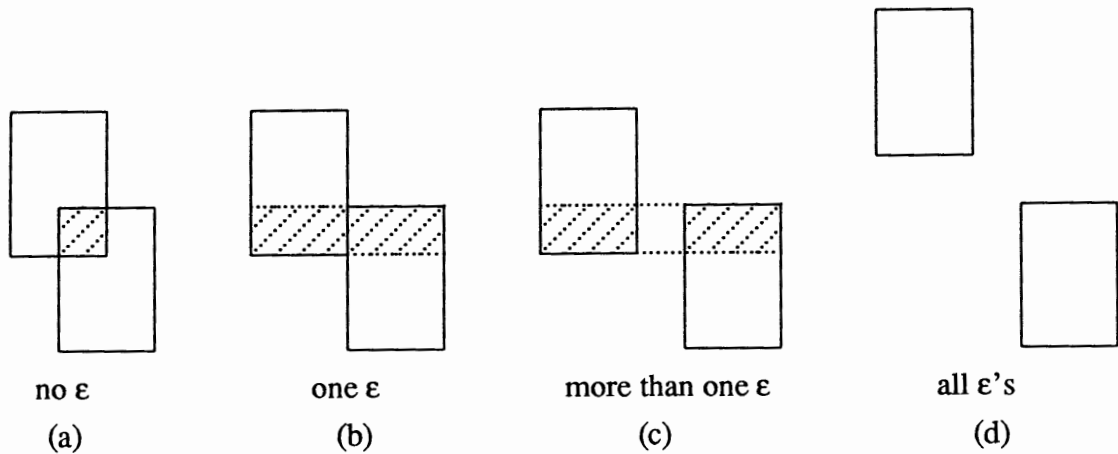


Figure 25. Relative positions between two cubes.

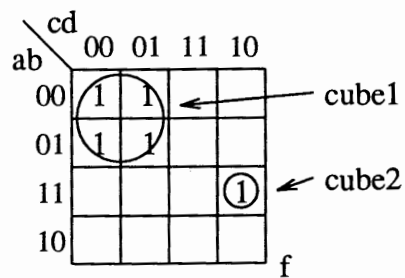


Figure 26. Karnaugh map of function f .

The column multiplicity is three under this partition ($ab|cd$, bond set $\{c, d\}$, free set $\{a, b\}$). The Intersection of the two cubes results in the "Intersection Cube" cube0.

	abcd
cube1	0-0-
cube2	1110
cube0	$\epsilon 1 \epsilon 0$

Because there are two ϵ 's, these two cubes have the relative position as shown in Figure 25(c). The comparisons of the cube0 with cube1 and the cube0 with cube2 result in the Partition Cubes C1 and C2, respectively:

	abcd
cube0	$\epsilon 1 \epsilon 0$
cube1	0-0-
C1	NNN

	abcd
cube0	$\epsilon 1 \epsilon 0$
cube2	1110
C2	NYNY

The *Partition Cube* is formed as follows:

If the corresponding bits of the two cubes are the same, there is a Y in the Partition Cube. Otherwise N.

Next, according to the Partition Cubes form the partitions which try to put as many shaded areas from both cubes into the same columns or rows as possible. We ignore the Partition Cubes if they have the value of all Y's or all N's because they have nonsense. Therefore, C1 is ignored. Group variables corresponding to Y in C2 into a group, group $\{b, d\}$, forming a partition $ac|bd$. Under this partition, part of cube1 and part of cube2 reside in column 10 ($b = 1, d = 0$) of the Karnaugh map. These are minterms 0010 and 1110 as shaded in Figure 27(a). This is just what we want that both cubes have a part in one column. Group variables corresponding to N in C2 into a group, group $\{a, c\}$, forming another partition $bd|ac$. Under this partition, part of cube1 and part of cube2 reside in row 10 ($b = 1, d = 0$) of the Karnaugh map. They are minterms 1000 and 1011 (shaded as well) in Figure 27(b). Again, this is what we want that both cubes have a part in one row.

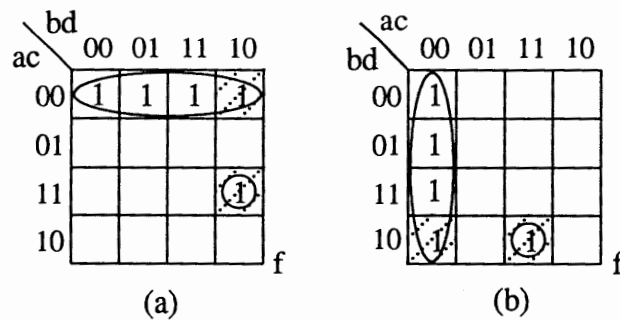


Figure 27. Karnaugh maps for the 'best' partitions.

Clearly, the bond set $\{b, d\}$ is a better partition which produces a column multiplicity of two, while bond set $\{a, c\}$ results in a column multiplicity of three. In summary, we first calculate all Partition Cubes for each pair of variables. Each Partition Cube will form a partition. We count the appearance number of each pair and according to this number form the final partitions.

Next is a more complex example used to show the detailed procedure of variable partitioning. Figure 28 shows the Karnaugh map of function f with the column multiplicity $v(ab|cde)$ of five. Its Espresso format input file is as follows:

```
.i 5
.o 1
.ilb a b c d e
.ob f
.type fr

00-01 1
-0101 1
01-11 1
-1111 1
----0 0
1-0-- 0
-0-1- 0
-1-0- 0

.end
```


		cde							
ab		000	001	011	010	110	111	101	100
	00	0	1	0	0	0	0	1	0
	01	0	0	1	0	0	1	0	0
	11	0	0	0	0	0	1	0	0
	10	0	0	0	0	0	0	1	0
		f							

Figure 28. Variable partitioning example.

(1) Calculate the Intersection Cubes. For ON set, they are:

00-01	00-01	00-01	-0101	-0101	01-11
-0101	01-11	-1111	01-11	-1111	-1111
$\overline{00101}$	$\overline{0\varepsilon-\varepsilon1}$	$\overline{0\varepsilon1\varepsilon1}$	$\overline{0\varepsilon1\varepsilon1}$	$\overline{-\varepsilon1\varepsilon1}$	$\overline{01111}$

For OFF set, they are:

---0	---0	---0	1-0-	1-0-	-0-1-
1-0-	-0-1-	-1-0-	-0-1-	-1-0-	-1-0-
$\overline{1-0-0}$	$\overline{-0-10}$	$\overline{-1-00}$	$\overline{1001-}$	$\overline{1100-}$	$\overline{-\varepsilon-\varepsilon-}$

(2) Calculate the Partition Cubes. For ON set, they are:

00101	00101	0\varepsilon-\varepsilon1	0\varepsilon-\varepsilon1	0\varepsilon1\varepsilon1	0\varepsilon1\varepsilon1
00-01	-0101	00-01	01-11	00-01	-1111
\overline{YNYNY}	\overline{NYYYY}	\overline{YNYNY}	\overline{YNYNY}	\overline{YNNNY}	\overline{NNYNY}
0\varepsilon1\varepsilon1	0\varepsilon1\varepsilon1	-\varepsilon1\varepsilon1	-\varepsilon1\varepsilon1	01111	01111
-0101	01-11	-0101	-1111	01-11	-1111
\overline{NNYNY}	\overline{YNNNY}	\overline{YNYNY}	\overline{YNYNY}	\overline{YNYNY}	\overline{NYYYY}

For OFF set, they are:

1-0-0	1-0-0	-0-10	-0-10	-1-00	-1-00
---0	1-0-	---0	-0-1-	---0	-1-0-
\overline{NYNYNY}	\overline{YYYYN}	\overline{YNYNY}	\overline{YYYYN}	\overline{YNYNY}	\overline{YYYYN}

1001 -	1001 -	1100 -	1100 -	- ε - ε -	- ε - ε -
1 - 0 - -	- 0 - 1 -	1 - 0 - -	- 1 - 0 -	- 0 - 1 -	- 1 - 0 -
<u>YNYNY</u>	<u>NYNYN</u>	<u>YNYNY</u>	<u>NYNYN</u>	<u>YNYNY</u>	<u>YNYNY</u>

(3) Calculate the Relationship Factors. Create four triangle tables as shown in Figure 29, call them ON-Y Table, ON-N Table, OFF-Y Table and OFF-N Table. Initiate all their cells to zero. The value in each cell represents a weighted *Relationship Factor* between the variables corresponding to the row and column labels of the table. Later on, we will sort the Relationship Factor Table. In order to keep the correct correspondence between the value and the variable pair it represents, we attach two more storage units to each cell to store the two variables that the Relationship Factor corresponds to. So the Relationship Factor Table is, in fact, a 3-tuple list. For example, the ON-Y Table in Figure 29 is a list: $\{(a, b, 2), (a, c, 4), (a, d, 2), (a, e, 8), (b, c, 2), (b, d, 4), (b, e, 4), (c, d, 2), (c, e, 8), (d, e, 4)\}$. Weight the Partition Cubes obtained in the second step in the following way:

Group variables corresponding to Y's in the Partition Cube. Select a pair of variables in the group and increase the value of the corresponding cell of the ON-Y Tables by 1. Group variables corresponding to N's in the Partition Cube, select a pair of variables in the group and increase the value of the corresponding cell of the ON-N Tables by 1. Execute the calculations for all pairs in the group. Perform the same operations for the OFF-Y and OFF-N Tables. The formula of Relationship Factor is:

$$\text{Relationship Factor} = \sum ((\text{pair between variables } i \text{ and } j) ? 1 : 0)$$

The summation is over all Partition Cubes and all pairs of variables in the Partition Cubes. "(pair between variables i and j)? 1 : 0" in the equation means that if variables i and j are a pair, take the value 1. Otherwise 0.

For example, if a Partition Cube C1 from the ON set is

$$C1 = \begin{matrix} & a & b & c & d & e & f & g \\ Y & N & Y & N & N & N & Y & Y \end{matrix}$$

the cells $a-c$, $a-f$, $a-g$, $c-f$, $c-g$ and $f-g$ of the ON-Y Table and the cells $b-d$, $b-e$ and $d-e$ of the ON-N Table will be increased by 1.

According to above rules, we fill the ON-Y, ON-N, OFF-Y and OFF-N Tables as shown in Figure 29.

	b	c	d	e
a	a-b 2	a-c 4	a-d 2	a-e 8
b		b-c 2	b-d 4	b-e 4
c			c-d 2	c-e 8
d				d-e 4

ON-Y Table

	b	c	d	e
a	a-b 2	a-c 0	a-d 2	a-e 0
b		b-c 2	b-d 8	b-e 0
c			c-d 2	c-e 0
d				d-e 0

ON-N Table

	b	c	d	e
a	a-b 3	a-c 9	a-d 3	a-e 6
b		b-c 3	b-d 6	b-e 3
c			c-d 3	c-e 6
d				d-e 3

OFF-Y Table

	b	c	d	e
a	a-b 0	a-c 4	a-d 0	a-e 0
b		b-c 0	b-d 5	b-e 0
c			c-d 0	c-e 0
d				d-e 0

OFF-N Table

Figure 29. ON-Y, ON-N, OFF-Y and OFF-N Tables.

(4) Form the "best" partitions. Sort Relationship Factor Table in decreasing order by the value of the Relationship Factors. Collect variable pairs with larger values in the ON-Y Table until the required number of variables for the bond set is reached. Both cells $a-e$ and $c-e$ have the values of 8, so select these two pairs and form the bond set $\{a, c, e\}$. For the ON-N Table, cell $b-d$ has the largest value of 8. All the rest are with the same value of 2. Cell $a-b$ has the value of 2 and it shares the common variable b with set $\{b, d\}$, so chose a, b and d as bond set $\{a, b, d\}$. Perform the same operations for the OFF-Y and OFF-N Tables to obtain another two bond sets $\{a, c, e\}$ and $\{b, d, e\}$. The Karnaugh maps under these four partitions are shown in Figure 30. In fact, we have only three Karnaugh maps because there are only three different partitions out of this four partitions. Partition $bd|ace$ and $ac|bde$ result in a column multiplicity of two. While partition $ce|abd$ results in a column multiplicity of three.

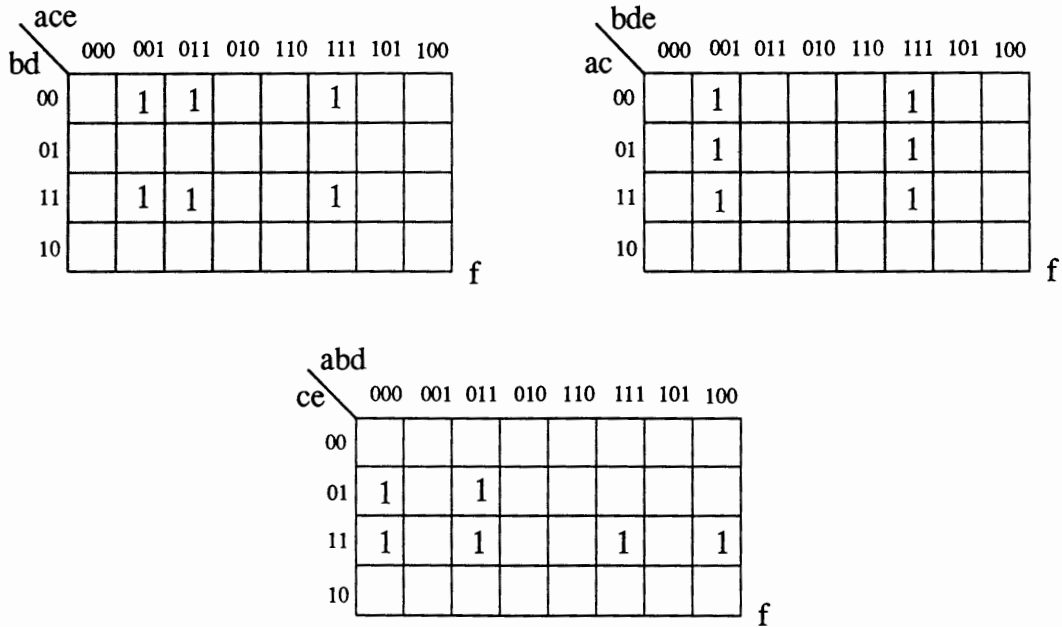


Figure 30. Partitions after variable partitioning.

We have checked that only partitions $bd|ace$ and $ac|bde$ can produce the minimum column multiplicity of two. There are ten $\binom{5}{3} = 10$ possible partitions out of these five-input function. The pseudo-code for variable partitioning is shown in Figure 31.

```

variable_partitioning( )
{
    repeat for ON-Y, ON-N, OFF-Y and OFF-N Tables
    {
        create Relationship Factor Table;
        sort Relationship Factor Table in decreasing order;
        bond_set = first pair in the queue;

        while ( |bond_set| < maximum_bond_set_number )
            bond_set = bond_set  $\cup$  pair in the next position;
    }
}

```

Figure 31. Pseudo-code of variable partitioning.

VI.3. LOCAL TRANSFORMATION

The known decomposition methods are passive in the sense that they only test whether a function is decomposable or not. If it is, the decomposition is carried out. But what do we do if the function is not decomposable?

Here the author presents a new approach to the decomposition. It is called the *Local Transformation Method*. This method can transform a nondecomposable function into several decomposable ones.

The basic idea of this method is to transform some columns in the original Karnaugh map to make them identical to some other columns in order to decrease the

column multiplicity. It is carried out in two steps:

- (1) The output values of the Conflict Cubes in one column are complemented in order to make two columns identical, creating the Modified and Modifying Karnaugh Maps.
- (2) EXOR operation of the functions described by the Modified Karnaugh Map and the function described by the Modifying Karnaugh Map is executed, creating the original function described just by the original Karnaugh map.

An attempt is made at the first step to make the Modified Cubes as big as possible, and at the same time keep the number of the Modified Cubes as small as possible. The *Conflict Cubes* are the cubes that make the two columns different. Like the cube -1 in columns 000 and 001 in Figure 32(a), it makes these two columns different. The formula for calculating the Conflict Cubes between i-th and j-th columns is:

$$\text{Conflict Cube Set} = \text{ON}(i) \bullet \text{OFF}(j) \cup \text{ON}(j) \bullet \text{OFF}(i)$$

ON(i) and OFF(i) are the ON and OFF sets of i-th column. ON(j) and OFF(j) are the ON and OFF sets of j-th column. The formula states that the Conflict Cubes between the i-th and j-th column are the union of the Intersection of the ON set of i-th column with the OFF set of j-th column and the Intersection of the ON set of j-th column with the OFF set of i-th column.

The following example is used to illustrate this method and above terminologies. Figure 32(a) is an original Karnaugh map with the bond set $\{c, d, e\}$ and a column multiplicity of four. We intend to decompose this function into two subfunctions, and each of them has a column multiplicity of no more than two.

We perform the local transformation to decrease the column multiplicity. First, the output value of the cube -10-1 and -1101 in Figure 32(a) are complemented. These cubes are called *Modified Cubes* (they are the same as the Conflict cubes). After modification, the resultant Karnaugh map is called the *Modified Karnaugh Map* (*Modified Function* f_{ed}) as shown in Figure 32(b). Now the column multiplicity of the Modified Karnaugh Map is two.

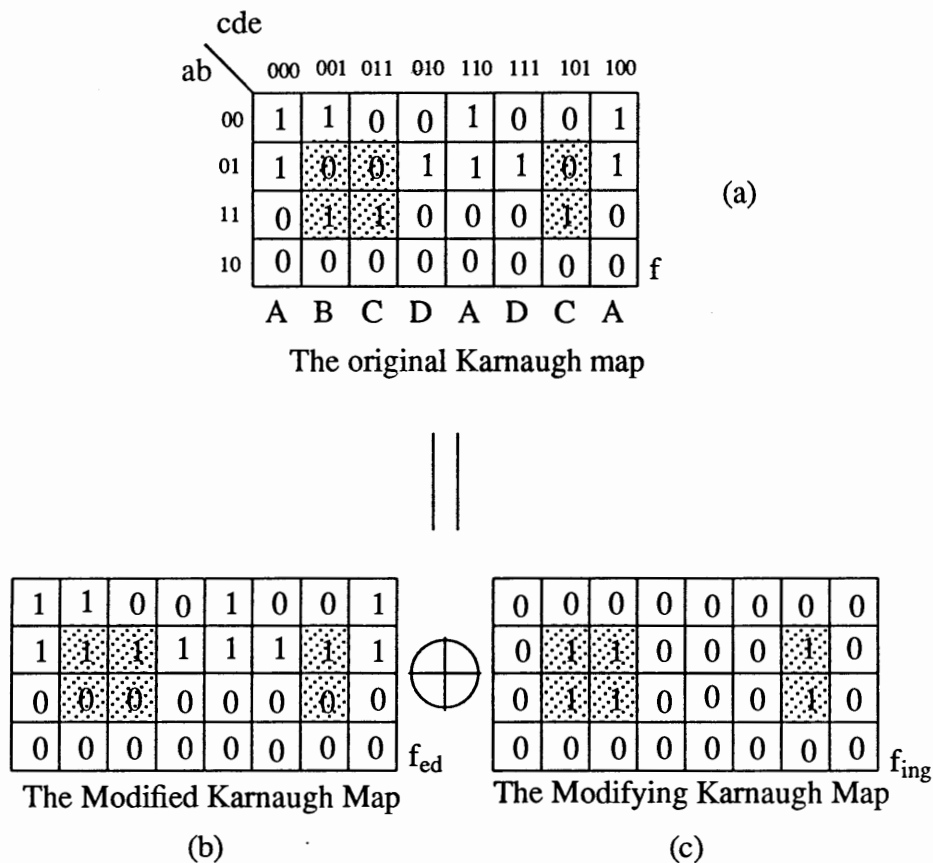


Figure 32. Local transformation.

Next, because the output value of the Modified Cubes have been complemented, the compensation must be made. To achieve this, another Karnaugh map is created with the positions corresponding to the Modified Cubes in the original Karnaugh map set to

1's and the others set to 0's. This is shown in Figure 32(c). It is called the *Modifying Karnaugh Map (Modifying Function f_{ing})*. Notice that in this case the column multiplicity of the Modifying Karnaugh Map is two as well. If there are DC cubes in the original Karnaugh map, we keep them unchanged in both the Modified and Modifying Karnaugh Maps, because they will make both of these maps as amenable as possible for further minimization.

Finally, the EXOR operation of the function described by the Modified Karnaugh Map and the function described by the Modifying Karnaugh Map results in the function described by the original Karnaugh map. That is:

$$f = f_{ed} \oplus f_{ing}$$

The presented method changes a nondecomposable function (in sense of the column multiplicity less than or equal to two) into two decomposable functions (with the column multiplicities of both equal to two). The method is called local transformation, but this local transformation is based on the global view of the entire function to make both the Modified and Modifying Karnaugh Maps more simple.

The application of local transformation to the general implementation of FPGA mapping is shown in Figure 33. If a function is nondecomposable, it will be transformed into two functions, a Modified and a Modifying function. If the Modified or Modifying function is nondecomposable (in a more restricted condition), the local transformation should be applied again.

Next, we use the function f in Figure 32(a) again as an example to show the detailed procedure of local transformation. In Figure 32(a) there are only four different columns as shown in Figure 34. We denote these four columns by letter A , B , C and D ,

and call them (letter) column *A*, column *B*, column *C* and column *D*.

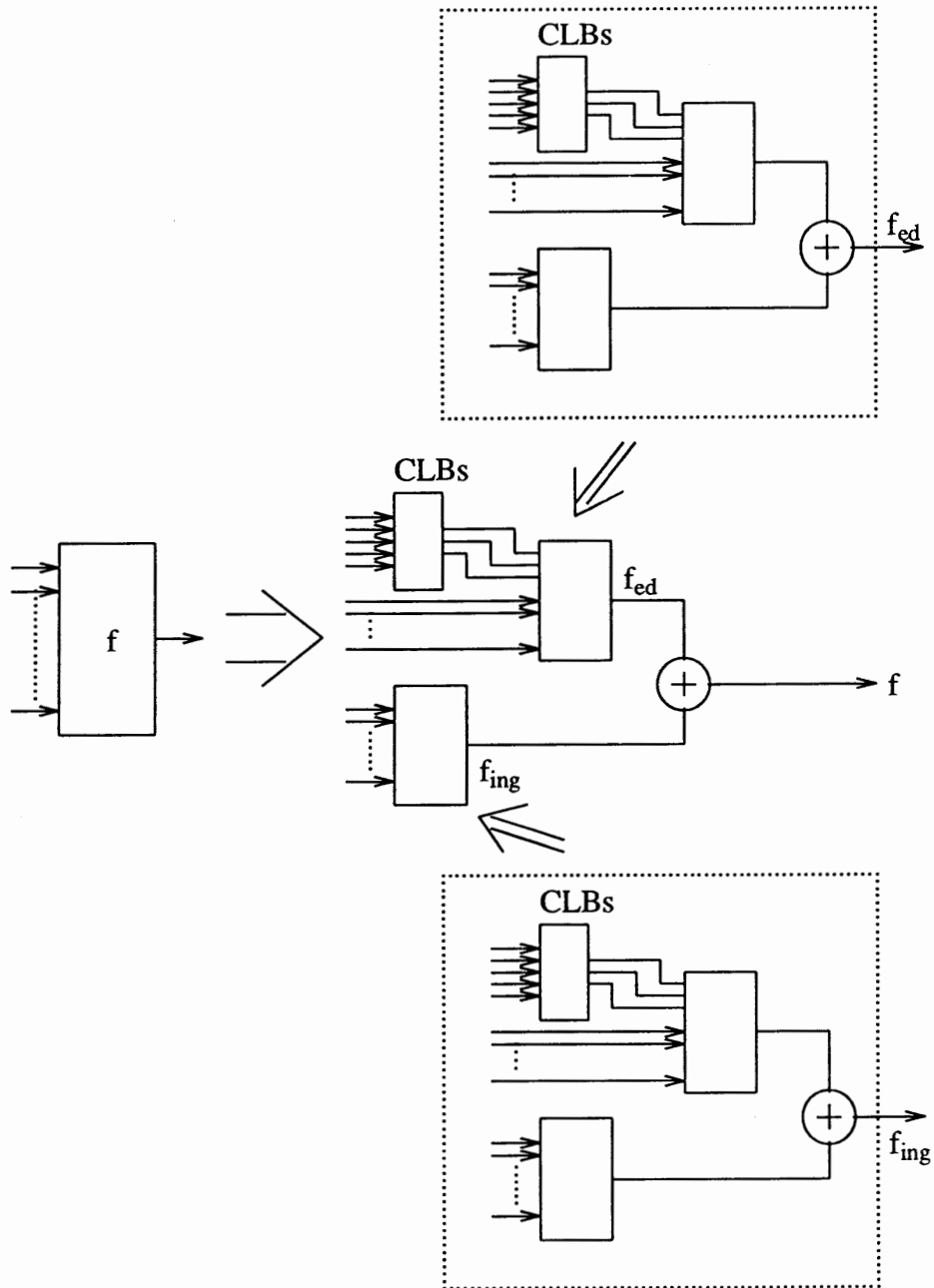


Figure 33. Application of local transformation to FPGA mapping.

Column *A* (letter column) includes three real columns: columns 000, 110 and 100 in the original Karnaugh map. Column *B* includes one column, column 001. Column *C* includes two columns: columns 011 and 101. Column *D* includes two columns: columns 010 and 111. "Number" in Figure 34 is the number of real columns that a letter column includes.

		ab			
		00	1	1	0 0
		01	1	0	0 1
		11	0	1	1 0
		10	0	0	0 0
Column		A	B	C	D
Number		3	1	2	2

Figure 34. Four different columns.

(1) Create the Modification Factor Table as shown in Figure 35 and initiate all its cells to zero. The value in each cell is a weighted Modification Factor of the column pair corresponding to the row and column labels of the table. Cell $A \rightarrow B$ stores the Modification Factor of complementing the output value of all Conflict Cubes in the column *A* in order to make columns *A* and *B* identical. Cell $B \rightarrow A$ stores the Modification Factor of complementing the output value of all Conflict Cubes in the column *B* in order to make columns *B* and *A* identical. Later on, we will sort the Modification Factor Table. In order to keep the correct correspondence between the value and the column pair it represents, we attach two more storage units to each cell to store the two columns that the Modification Factor corresponds to. So the Modification Factor Table is, in fact, a 3-tuple list. For example, the Modification Factor Table in Figure 35 is a list: $\{(A, B, 12), (A, C, 24), (A, D, 15), (B, A, 4), (B, C, 5), (B, D, 8), (C, A,$

16), (C, B, 10), (C, D, 8), (D, A, 10), (D, B, 16), (D, C, 8)}. The Modification Factors are calculated in the following way:

If the Conflict Cube is a minterm, increase the value of the corresponding cell in the table by amount of the number of input variables. If the Conflict Cube is composed of two minterms, increase the value of the corresponding cell in the table by amount of the number of input variables minus one. This is because a larger cube can simplify to a greater extent both the Modified and Modifying Karnaugh Maps. If the Conflict Cube is composed of four minterms, increase the value of the corresponding cell in the table by amount of the number of input variables minus two, and so forth. That is:

Modification Factor

$$= \sum (\text{number of input variables} - \text{number of '-'s in the Conflict Cube})$$

The summation is over all Conflict Cubes in the modified column. "-"s in the equation is the dashes in the Conflict Cube. For example, the cube 10--1 has two dashes.





	A	B	C	D
A		A→B 12	A→C 24	A→D 15
B	B→A 4		B→C 5	B→D 8
C	C→A 16	C→B 10		C→D 8
D	D→A 10	D→B 16	D→C 8	

Figure 35. Modification Factor Table.

Let's fill the Modification Factor Table in Figure 35 now. The Conflict Cube between column A and B is the cube -1. If we change column A to make it identical to column B , we need to complement the output value of the Conflict Cube (cube -1 in column A). That is to change the output value of minterm 01 of column A in Figure 34 from 1 to 0 and the output value of minterm 11 from 0 to 1. Column A will have the vector $[1, 0, 1, 0]$ after complementation of the Conflict Cube. This vector is the same as that of the column B . Cube -1 has one - (dash), the Modification Factor would be the number of input variables minus 1, that is four. Further more, column A includes three columns, columns 000, 110 and 100. All these three columns need to be modified. So the Modification Factor must be multiplied by three. Therefore the final Modification Factor is twelve as shown in cell $A \rightarrow B$ of the Modification Factor Table in Figure 35. Another way to make column A and B identical is to complement the output value of the Conflict Cube, cube -1 in column B . That is to change the output value of minterm 01 of column B in Figure 34 from 0 to 1 and the output value of minterm 11 from 1 to 0, so both columns have the same vector $[1, 1, 0, 0]$ after the modification. Column B includes only one column which is column 001, the Modification Factor is four as shown in Figure 35. Using the same reasoning, we can complete the Modification Factor Table.

(2) Sort the Modification Factor Table in increase order by their Modification Factors. Modify the columns which have smaller Modification Factors until the required column multiplicity is reached. Cell $B \rightarrow A$ is selected because it has the smallest value of four. Complement the output value of Conflict Cube, cube -1 in column 001 of the Karnaugh map in Figure 32(a), The vector of column 001 is changed from $[1, 0, 1, 0]$ to $[1, 1, 0, 0]$. At the same time, set the output value of cube -1001 of the Modifying Karnaugh in Figure 32(c) to 1. After modifying column B , the column multiplicity of the Modified Karnaugh Map is reduced to three. Our aim is to reduce it to two, so another modification will be carried out. The next smaller Modification Factor is five in cell $B \rightarrow C$. But we

cannot change column *B* to column *C* because we have changed column *B* to column *A*. The next smaller value is eight in cell *C*→*D*. Complement the output value of Conflict Cubes, cube -1 in both columns 011 and 101 of the Karnaugh map in Figure 32(a), The vectors of the columns 011 and 101 are changed from [0, 0, 1, 0] to [0, 1, 0, 0]. At the same time, set the output value of cubes -1011 and -1101 of the Modifying Karnaugh in Figure 32(c) to 1. After this modification, the column multiplicity of the Modified Karnaugh Map is reduced to two.

(3) We have transformed the function *f* with a column multiplicity of four into two functions: *f_{ed}* and *f_{ing}*, both of them have a column multiplicity of two. The relation between them is:

$$f = f_{ed} \oplus f_{ing}$$

The pseudo-code for local transformation is shown in Figure 36.

```

local_transformation( )
{
    create Modification Factor Table;
    sort Modification Factor Table in increasing order;
    modify column at the beginning of the queue;
    change Modified Karnaugh Map;
    fill Modifying Karnaugh Map;

    while ( column_multiplicity > required_column_multiplicity )
    {
        modify column at the next position
        change Modified Karnaugh Map;
        fill Modifying Karnaugh Map;
    }
}

```

Figure 36. Pseudo-code of local transformation.

CHAPTER VII

PROGRAM *TRADE* AND ITS EVALUATIONS

The techniques presented in the previous sections have been incorporated into a program named *TRADE* (*TRAnsformation and DEcomposition*) which reads in the input file in *Espresso* (*.type fr*) format and outputs in *Blif* format with the input variables of each node less than or equal to five. Cube calculus is used in *TRADE* for all operations.

VII.1. PROCEDURE OF *TRADE*

The basic steps of *TRADE* are as follows:

- (1) Read in the input file written in *Espresso* ".type fr" format.
- (2) Select an output. Perform partition analysis (the number of bond set variables is fixed to five) to obtain the "best" partitions and Additional Partitions. *Additional Partitions* are the partitions whose bond sets consist of the variables that are the input variables of some CLBs in the CLB Pool, and these variables are also in the range of the input variables of the current decomposition. The *CLB Pool* is a list of all CLBs previously generated by the program. For example, if the input variables of the current decomposition are $a, b, c, d, e, f, g, h, i$, and in the CLB Pool there are three CLBs with the input variables of each: $\{a, b, c, d, e\}$, $\{c, d, g, h, i\}$ and $\{a, h, k, l, m\}$ respectively. Then the variable sets $\{a, b, c, d, e\}$ and $\{c, d, g, h, i\}$ can be used as the additional partitions, while $\{a, h, k, l, m\}$ can not be used because there are no variables k, l and m in the current decomposition.

(3) Execute decompositions using the "best" and Additional Partitions. Encode the bond set and try to use as many CLBs from the CLB Pool as possible. The way to efficiently reuse CLBs will be discussed later in this chapter. Graph coloring technique is applied at this stage to get a quasi-optimum don't care assignment. Select a partition which results in the smallest Cover Ratio. The *Cover Ratio* is a ratio of the number of newly created CLBs over the difference of the input variables before and after decomposition, that is:

$$\text{Cover Ratio} = \frac{\text{number_of_newly_created CLBs}}{\text{inputs_before_decomposition} - \text{inputs_after_decomposition}}$$

For example, in Figure 37, if CLB x and y exist in the CLB Pool, CLB z is a newly created one, then Cover Ratio = $1/(8 - 6) = 0.5$.

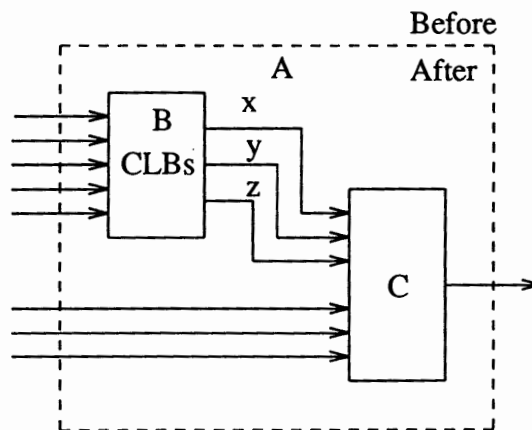


Figure 37. Before and after decomposition.

If the function is nondecomposable, perform the local transformation to make it decomposable.

(4) Repeat steps 2 to 3 for the blocks left (block C in Figure 37) until all decomposed blocks are with five or less inputs.

(5) Repeat steps 2 to 4 for all Modifying Functions which were created by local transformations.

(6) Repeat steps 2 to 5 for all outputs.

(7) Merge all possible nodes into the FG mode CLBs. The way to merge the nodes is discussed in the next section.

The pseudo-code for *TRADE* program is shown in Figure 38.

```

TRADE( )
{
    read_input_file( );                /* read in input file */

    for ( i = 0; i < number_of_primary_outputs; i++ )    /* loop for all
    {                                                         output functions */
        do
        {
            variable_partitioning( );        /* find best partition */
            create_incompatibility_graph( );    /* create incompatibility
            graph */
            graph_coloring( );                /* quasi-optimum don't
            care assignment */
            if ( decomposable != true )
                local_transformation( );        /* make decomposable */

            bond_set_encoding( );            /* encode bond set */
            CLB_reusing( );                  /* use CLBs in CLB
            Pool */
        } while ( function(s) from local transformation == true )
            /* if there is function from local
            transformation, repeat */
    }

    CLB_merging( );                /* merge nodes into FG mode CLBs */
    output_result( );              /* output results */
}

```

Figure 38. Pseudo-code of *TRADE* program.

VII.2. CLB MERGING

We have explained the FG mode CLB of Xilinx architecture in Chapter II. If two nodes satisfy the condition that each has no more than four input variables, and the total number of input variables in these two nodes is no more than five, these two nodes can be combined into a FG mode CLB. We use the following procedure to merge the nodes:

Collect all nodes with no more than four input variables, put them in a queue and sort them in decreasing order by the number of their inputs. Pick the node at the beginning of the queue, call it the *Master Node*, then pick another one next to it. Test if these two nodes can be combined into one FG mode CLB. If they can be combined, combine and remove them from the queue. If not, pick another node in the next position, and perform the combining test again. If, until the end of the queue, no node can be combined with the Master Node, remove the Master Node from the queue. Pick the node at the beginning of the queue as a new Master Node and repeat above operations until the queue is empty.

Next example is used to show the detailed merging procedure. There are six nodes in the queue.

Node Number	1	2	3	4	5	6
Inputs	(abcd)	(abce)	(abcd)	(abef)	(efij)	(befh)

Pick node 1 as the Master Node. Test to see if it can be combined with the next node which is node 2. They can be combined because the number of inputs of each node is four, $\{a, b, c, d\}$ and $\{a, b, c, e\}$, and the number of total variables out of these two nodes is five $\{a, b, c, d, e\}$. Combine them and give them a name, CLB A, and remove them from the queue. Now there are four nodes left in the queue.

Node Number	3	4	5	6
Inputs	(abcd)	(abef)	(efij)	(befh)

Pick node 3 as the Master Node. Test to see if it can be combined with the next node which is node 4. It cannot be combined because the number of inputs out of these two nodes is six $\{a, b, c, d, e, f\}$. It should be no more than five if the two are combined. Test again to see if it can be combined with node 5. The answer is no once more. Do the test with node 6. It fails again. So, no node can be combined with node 3. Give node 3 a name, CLB *B*, and remove it from the queue. The queue turns out to be:

Node Number	4	5	6
Inputs	(abef)	(efij)	(befh)

Pick node 4 as the Master Node. Test to see if it can be combined with node 5. The test fails. Test it with node 6. This time success. Combine these two nodes into a new node, CLB *C*, and remove them from the queue. Now, only node 5 is left in the queue. Give it a name, CLB *D*, and remove it from the queue. The queue is empty. The final merging result is as follows:

CLB	A	B	C	D
Node Number	1,2	3	4,6	5

Nodes 1 and 2 are combined into a FG mode CLB, CLB *A*. Nodes 4 and 6 are combined into another FG mode CLB, CLB *C*. Nodes 3 and 5 remain as they were. The pseudo-code for CLB merging is shown in Figure 39.

```

CLB_merging( )
{
    while ( queue != empty )
    {
        m_n = Master Node;
        n_n = next node;

        do
        {
            if ( merge(m_n, n_n) == true )
            {
                combine m_n and n_n into a FG mode CLB;
                remove m_n and n_n;
                break;
            } else {
                n_n = next node;
            }
        } while ( n_n != last node )
        else {
            remove m_n;
        }
    }
}

```

Figure 39. Pseudo-code of CLB merging.

VII.3. CLB REUSING

In the previous section, we have presented the encoding algorithm for the bond set, but we didn't mention the possibility that some CLBs in the CLB Pool might be reused. In *TRADE*, each previously generated CLB is recorded in a 32-bit long word, called a *CLB frame*. Because there can be up to five inputs to each CLB, thirty two bits are required to store all possible combinations of the inputs ($2^5 = 32$). For example, the logic of CLB *h*:

ment the pattern $\boxed{0011}$ (call it x , this is the pattern for coding as shown in Figure 40(c), not the internal storage frame). If we code y as $\boxed{1010}$ (y is a newly created CLB) as shown in Figure 40(c), we have realized the coding of xy by using one CLB in the CLB Pool and a newly created one. The y is coded in such a way that no two xy Codes are identical. Next example is used to show the detailed procedure of CLB Reusing algorithm.

Suppose that we have a function with seven input variables $\{a, b, c, d, e, f, g\}$ as shown in Figure 41.

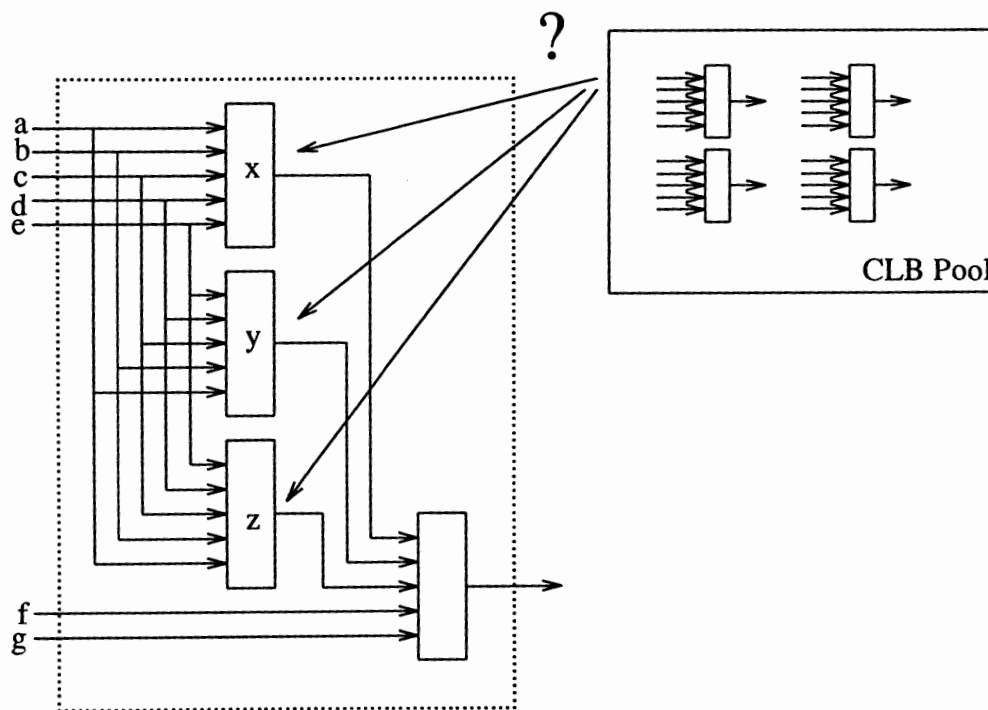


Figure 41. CLB reusing example.

After graph coloring, six colors are obtained as shown in Figure 42. In Figure 42, columns 0, 1, 3, 4, 7, 9, 12, 14, 15 and 17 belong to color 0, columns 18, 19, 22, 23, 25, 26 and 27 belong to color 1, and so forth. A careful reader will find that columns 20 and

29 are missing. They are DC columns and can be put in any color position.

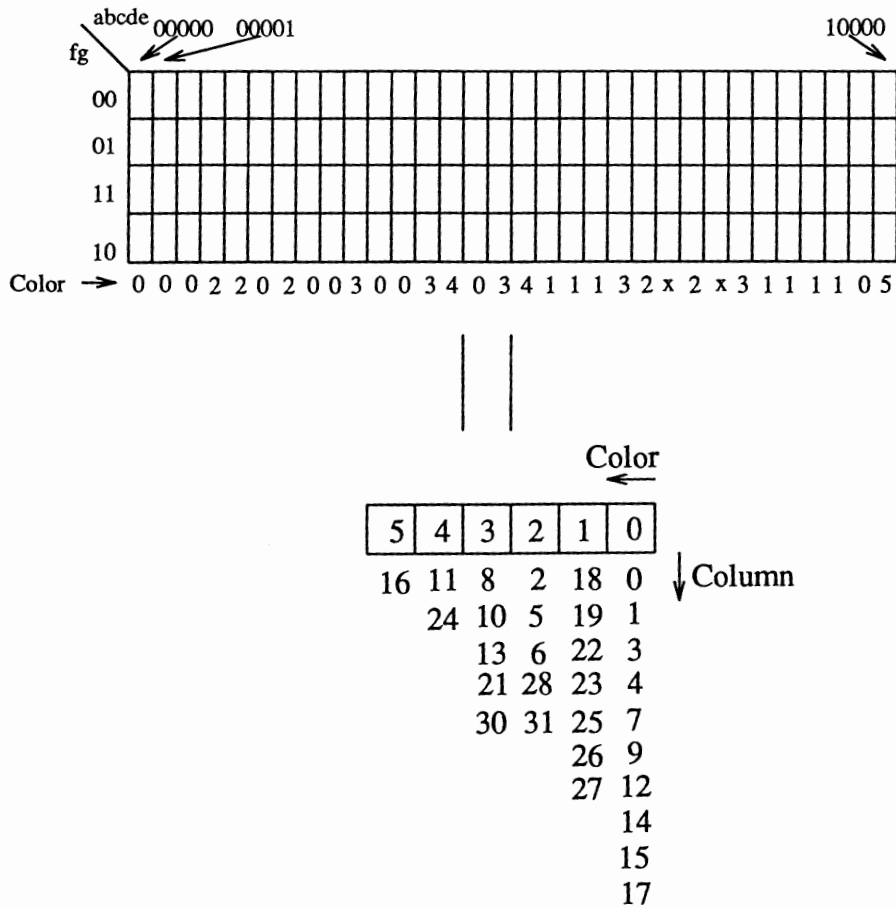


Figure 42. After graph coloring.

Because there are six colors (column multiplicity is six as well), we need three CLBs ($3 > \log_2 6 = 2.585$) to encode the bond set. Suppose that in the CLB Pool there are three CLBs which have the same input variables as that in the current bond set, they are CLBs w , x and y .

0001000010101001001001000000001011	w
01000111011101110111011101110011011	x
1000111101100110000000000000001100100	y

We try to use as many CLBs in the CLB Pool as possible. If the CLBs in the CLB Pool can fully cover the columns in some colors, they might be reused. The term "*fully cover the columns in some colors*" means that in the CLB frame all positions corresponding to the columns included in some colors must be exactly 1, while all the other positions must be 0. We test if the CLBs w , x and y can fully cover the columns in some colors in Figure 42. It is found that the CLB x can fully cover the colors 0, 1 and 3, because in the CLB x frame all positions correspond to the columns included in the color 0, 1 and 3 are 1, all the others are 0. CLB x can fully cover the colors 0, 1 and 3, the bits corresponding to the colors 0, 1 and 3 in the CLB x pattern are 1, the others are 0. CLB x has the pattern $\boxed{001011}$ as shown in Figure 43. The same test results in that CLB y can fully cover the colors 1 and 2, and it has the pattern $\boxed{000110}$ as shown in Figure 43. While CLB w can cover the columns 0, 1 and 3 in the color 0, but it cannot cover the columns 4, 7, 9, 12, 14, 15 and 17 in the color 0, this is not a full cover. Therefore CLBs x and y can be reused, but CLB w can not be reused. We add a new CLB (CLB z) and code it in such a way that each color has a different Code bit. The procedure for coding the newly added CLB is as follows:

Start from the color position 0 in Figure 43. CLB y and CLB x give the bits 0 and 1 (Code 01. Code is read vertically, not horizontally), respectively. Put a 0 at the color position 0 of CLB z as shown in Figure 43. In the color position 1, the corresponding bits from CLB y and CLB x are 1 and 1 (Code 11) respectively. Because Code 11 is different from Code 01 in the color position 0, put a 0 again at the color position 1 of CLB z . The same way, put a 0 at the color position 2 of CLB z . In the color position 3, the Code is 01, it is the same as that in the color position 0. Increase the Code put in the color position 0 of CLB z (that is 0) by 1, this leads to a 1. So put a 1 at the color position 3 of CLB z . Perform the same operation for the color positions 4 and 5. The final coding of CLB z is $\boxed{101000}$ as shown in

Figure 43.

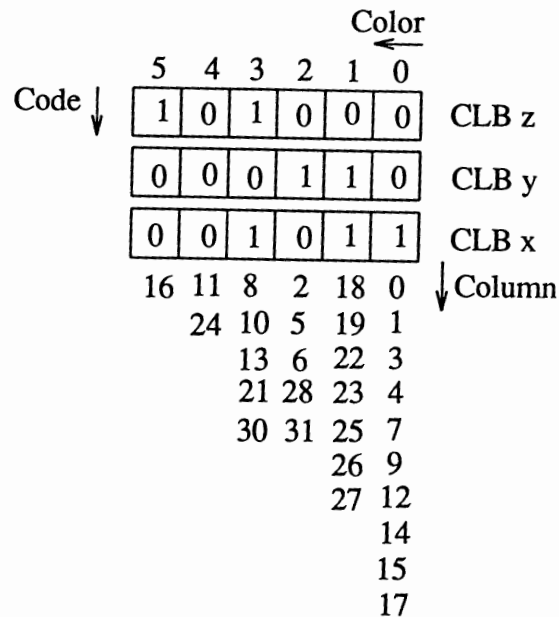


Figure 43. The final coding.

The newly generated CLB z has the internal storage frame as:

010000000001000001001001001000000000 z

It is formed according to the pattern 101000 of CLB z. The color position 3 is 1 in the pattern of CLB z. Columns 8, 10, 12, 21 and 30 correspond to this color, so in the CLB z frame, the positions 8, 10, 12, 21 and 30 are filled with 1. The same operation is performed for the color position 5, and the internal storage frame of CLB z is obtained. By proper coding, we have reused two CLBs in the CLB Pool. The pseudo-code for CLB reusing is shown in Figure 44.


```

CLB_reusing( )
{
    collect reusable CLBs;
    fully cover testing;

    for ( i = 0; i < total_color; i++ )
    {
        if ( reused_CLB_code_repeat(i) == true )
            code(i)++;

        new_CLB_code(i) = code(i);
    }
}

```

Figure 44. Pseudo-code of CLB reusing.

VII.4. EVALUATION OF THE RESULTS

We ran *TRADE* on a networked SUN 4/670MP Workstation. The results are listed in Table II. All results are verified by "verify" command of *MIS-II* system. The procedure of verifying is:

```

TRADE  MCNC_file_name m n  \* execute TRADE program
                                MCNC_file_name: The name of the example file
                                from the MCNC benchmark;
                                m: The maximum number of inputs to each CLB as
                                shown in Figure 45;
                                n: The maximum number of bond set outputs as
                                shown in Figure 45.
                                The output will be put in a file named wad.out. *\

misII                                     \* enter misII program *\
read_pla  MCNC_file_name                \* read in the MCNC benchmark example file *\
verify wad.out                          \* verify *\

```

The results listed under *MIS-PGA(phase 1)* are from [9], The results listed under *MIS-PGA(new)* are from [2]. Both of them were run on a DEC5500. There is no delay

information provided in [2]. The examples *root*, *bench*, *fout* and *test1* are taken from the Espresso package, and all of them are incompletely specified functions except *root*. *t-00* is obtained by changing all DC outputs in *test1* to OFF outputs, therefore, it is a completely specified function. *t-10* is obtained by randomly changing 10 percent of the OFF outputs in *t-00* to DC outputs. The same way, *t-20* to *t-90* is obtained by randomly changing 20 to 90 percent of the OFF outputs in *t-00* to DC outputs, respectively. Because we were not able to access the *MIS_PGA(phase 1)* and *MIS_PGA(new)* programs, we couldn't make comparisons of incompletely specified functions.

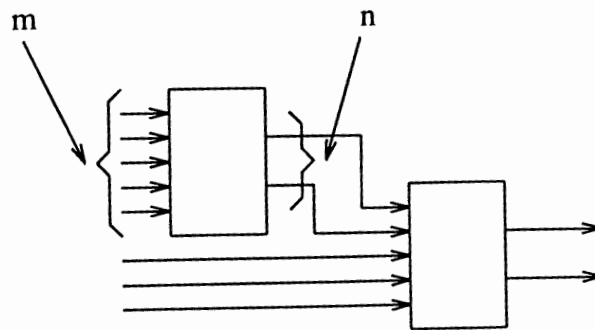


Figure 45. Verifying example.

In Table II, E/N is the name of the example. I/N is the number of input variables. O/N is the number of output functions. CLBs is the number of CLBs in the final mapped circuit. Time is the running time of the program which is measured by *time* command of UNIX system. The unit of the Time is second. Lev is the longest path (number of CLBs) that a signal must go from the primary input to the primary output in the final mapped circuit. From Table II, we observe that if the DC outputs are maintained, the number of CLBs can be greatly decreased. However, even for the completely specified functions, our program found better results than *MIS-PGA(phase 1)* and *MIS-PGA(new)* with respect to both the delay and area minimization.

TABLE II
COMPARISONS AMONG *TRADE*, *MIS-PGA(PHASE I)* AND *MIS-PGA(NEW)*

E/N	I/N	O/N	TRADE			MIS-PGA(phase I)			MIS-PGA(new)	
			CLBs	Lev	Time	CLBs	Lev	Time	CLBs	Time
<i>alu2</i>	10	8	22	3	12.2	122	6	42.6	109	773.8
<i>9sym</i>	9	1	6	3	4.9	7	3	15.2	7	339.7
<i>9symml</i>	9	1	6	3	4.7	7	3	9.9	7	127.2
<i>rd73</i>	7	3	5	2	3.7	8	2	4.4	6	24.0
<i>rd84</i>	8	4	8	3	11.6	13	3	9.8	10	73.7
<i>f51m</i>	8	8	9	3	2.3	23	4	5.9	17	14.4
<i>5xpl</i>	7	10	11	2	4.3	21	2	3.5	18	22.4
<i>z4ml</i>	7	4	4	2	2.0	10	2	2.1	5	5.0
<i>sao2</i>	10	4	27	3	13.8	45	5	9.5	28	41.9
<i>bw*</i>	5	28	27	1	0.3	28	1	8.3	28	17.3
<i>misex1</i>	8	7	14	2	3.4	17	2	1.7	11	2.7
<i>clip</i>	9	5	29	4	12.1	54	4	3.7	28	58.4
<i>b9</i>	16	5	29	4	28.7	47	3	2.3	39	27.6
<i>misex2</i>	25	18	31	4	17.0	37	3	1.4	28	3.4
<i>duke2</i>	22	29	159	6	370.7	164	6	16.4	110	203.7
<i>root</i>	8	5	21	3	9.8					
<i>bench*</i>	6	8	16	2	1.0					
<i>four*</i>	6	10	26	2	4.3					
<i>test1*</i>	8	10	66	3	21.1					
<i>t-00</i>	8	10	166	5	81.6					
<i>t-10*</i>	8	10	152	5	64.6					
<i>t-30*</i>	8	10	125	5	50.6					
<i>t-50*</i>	8	10	83	5	32.8					
<i>t-70*</i>	8	10	76	4	18.1					
<i>t-90*</i>	8	10	46	3	10.2					

* Incompletely specified function.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

VIII.1. CONCLUSIONS

A new general approach to the decomposition of incompletely specified functions and its application to FPGA mapping [29] [30] have been presented. Variable Partitioning, Graph Coloring, Bond Set Encoding, Local Transformation and CLB Reusing are the outstanding features of this approach. One of the main advantages of this approach is that it is intended for incompletely specified functions, thus giving for such kind of functions much better results than the existing methods.

Compared with the existing FPGA mapping approach, our method is totally new. We developed a fast graph coloring method for the don't care assignment, so the program can accept incompletely specified functions and perform a quasi-optimum assignment to the unspecified part of the function. We developed a high quality heuristic method to choose the "best" partitions, avoiding the thorough test of all possible decomposition charts which is impractical when there are many input variables. We introduced the local transformation concept, which can transform nondecomposable functions into decomposable ones, making it possible to apply decomposition method to FPGA mapping. Finally, the Cube calculus is used entirely in the *TRADE* program, the operation is global and very fast.

VIII.2. FUTURE WORK

The program has been successfully verified and benchmarked on several MCNC examples and some incompletely specified functions. There are still several opportunities to further improve both its speed and quality of the generated solutions. Currently we work on two-dimensional Karnaugh maps. A possible extension would be to develop the algorithms to operate on three-dimensional or multi-dimensional Karnaugh maps, to incorporate the FGM mode CLBs of Xilinx 3000 series and take advantage of the Xilinx 4000 series.

REFERENCES

1. Xilinx, Inc., *Xilinx Programmable Gate Array Data Book*, 1992.
2. Rajeev Murgai, Narendra Shenoy, Robert K. Brayton, and Alberto Shangiovanni-Vincentelli, "Improved Logic Synthesis Algorithms for Table Look Up Architectures," *ICCAD 1991*, pp. 564-567, Santa Clara, CA, Nov. 1991.
3. Rajeev Murgai, Yoshihito Nishizaki, Narendra Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli, "Logic Synthesis for Programmable Gate Arrays," *Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 620-625, Orlando, FL, June 1990.
4. D. Filo, J. C. Yang, F. Mailhot, and G. D. Micheli, "Technology Mapping for a Two-Output RAM-based Field-Programmable Gate Array," *European Design Automation Conf.*, pp. 534-538, February 1991.
5. Robert Francis, Jonathan Rose, and Zvonko Vranesic, "Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs," *Proc. 28th ACM/IEEE Design Automation Conf.*, pp. 227-233, San Francisco, CA, June 1991.
6. Robert J. Francis, Jonathan Rose, and Kevin Chung, "Chortle: A Technology Mapping Program for Lookup Table-Based Field Programmable Gate Array," *Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 613-619, 1990.
7. Kevin Karplus, "Xmap: A Technology Mapper for Table-lookup Field-Programmable Gate Arrays," *Proc. 28th ACM/IEEE Design Automation Conf.*, pp. 240-243, San Francisco, CA, June 1991.
8. Nam-Sung Woo, "A Heuristic Method for FPGA Technology Mapping Based on the Edge Visibility," *Proc. 28th ACM/IEEE Design Automation Conf.*, pp. 248-251, San Francisco, CA, June 1991.
9. Rajeev Murgai, Narendra Shenoy, Robert K. Brayton, and Alberto Shangiovanni-Vincentelli, "Performance Directed Synthesis for Table Look Up Programmable Gate Arrays," *ICCAD 1991*, pp. 572-575, Santa Clara, CA, Nov. 1991.
10. R. J. Francis, J. Rose, and Z. Vranesic, "Technology Mapping for Delay Optimization of Lookup Table-Based FPGAs," *MCNC Logic Synthesis Workshop*, 1991.
11. R. J. Francis, J. Rose, and Z. Vranesic, "Technology Mapping of Lookup Table-Based FPGAs for Performance," *Proc. ICCAD*, pp. 568-571, Nov. 1991.

12. Jason Cong, Andrew Kahng, Peter Trajmar, and Kuang-Chien Chen, "Graph Based FPGA Technology Mapping for Delay Optimization," *Proc. First Int'l ACM/SIGDA Workshop on Field Programmable Gate Arrays*, pp. 77-82, Berkeley, CA, February, 1992.
13. S. Lee Hight, "Complex Disjunctive Decomposition of Incompletely Specified Boolean Functions," *IEEE Trans. on Computers*, vol. c-22, no. 1, pp. 103-110, Jan. 1973.
14. V. Yun-shen and Archie C. Mckellar, "An Algorithm for the Disjunctive Decomposition of Switching Functions," *IEEE Tran. on Computer*, vol. c-19, no. 3, pp. 239-248, March 1970.
15. Devadas Varma and E. A. Trachtenberg, "Design Automation Tools for Efficient Implementation of Logic Functions by Decomposition," *IEEE Trans. CAD*, vol. 8, no. 8, pp. 901-916, August 1989.
16. J. Vasudevamurthy and J. Rajski, "A Method for Concurrent Decomposition and Factorization of Boolean Expressions," *Proc. IEEE ICCAD*, vol. 7, no. 12, pp. 1290-1300, December 1988.
17. R. L. Ashenhurst, "The Decomposition of Switching Functions," *Proc. Int'l Symp. Theory of Switching Function*, pp. 74-116, 1959.
18. H. Allen Curtis, "A Generalized Tree Circuit," *J. ACM*, vol. 8, pp. 484-496, 1961.
19. Marek A. Perkowski and James E. Brown, "A Unified Approach to Designs Implemented with Multiplexers and To the Decomposition of Boolean Functions," *Proc. ASEE Annual Conf.*, pp. 1610-1618, 1988.
20. H. Allen Curtis, "Generalized Tree Circuit-The Basic Building Block of an Extended Decomposition Theory," *J. ACM*, vol. 10, pp. 562-581, 1963.
21. J. P. Roth and R. M. Karp, "Minimization over Boolean Graphs," *IBM J. of Research and Development*, vol. 6, no. 2, pp. 227-238, April, 1962.
22. Loc Bao Nguyen, Marek A. Perkowski, and Nahum B. Goldstein, "PALMINI - Fast Boolean Minimizer for Personal Computers," *Proc. 24th ACM/IEEE Design Automation Conf.*, pp. 615-621, 1987.
23. Saeyang Yang and Maciej J. Ciesielski, "A Generalized PLA Decomposition with Programmable Encoders," *Proc. Int'l Workshop on Logic Synthesis, MCNC*, pp. 1-13, 1989.
24. Saeyang Yang and Maciej J. Ciesielski, "Optimum and Suboptimum Algorithms for Input Encoding and Its Relationship to Logic Minimization," *IEEE Trans. CAD*, vol. 10, no. 1, pp. 9-12, Jan. 1991.

25. Nicos Christofides, *Graph Theory*, Academic Press, New York, 1975.
26. M. Gondran and M. Minoux, *Graphs and Algorithms*, John Wiley & Sons, New York, 1984.
27. Ronald C. Read, *Graph Theory and Computing*, Academic Press, New York, 1972.
28. Frank Harary and John S. Maylee, "Graph and Applications," *Proc. First Colorado Symp. on Graph Theory*, John Wiley & Sons, New York, 1985.
29. Wei Wan and Marek A. Perkowski, "A New Approach to the Decomposition of Incompletely Specified Functions Based on Graph-coloring and Local Transformations and Its Application to FPGA Mapping," *Proc. European Design Automation Conf.*, Hamburg, Germany, Sept. 1992.
30. Wei Wan and Marek A. Perkowski, "TRADE: A Lookup Table FPGA Mapper Based on a Generalized Boolean Decomposition," *Submitted to ICCAD*, Santa Clara, CA, Nov. 1992.