1998

The Design of Cube Calculus Machine Using Sram-Based Fpga Reconfigurable Hardware Dec's Perle-1 Board

Qihong Chen
Portland State University

THESIS APPROVAL

The abstract and thesis of Michael A. Burns for the Master of Science in Electrical and Computer Engineering were presented October 30, 1997, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS: [redacted]

Marek A. Perkowski, Chair

[redacted]

Douglas V. Hall

[redacted]

Laszlo Csanky
Representative of the Office of
Graduate Studies

DEPARTMENT APPROVAL: [redacted]

Lee Casperson, Chair
Department of Electrical Engineering

* *

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by [redacted] on Jan 20, 1998

ABSTRACT

An abstract of the thesis of Michael A. Burns for the Master of Science in Electrical and Computer Engineering presented October 30, 1997.

Title: New Approaches to Column Compatibility Checking and Column-Based Input/Output Encoding for Curtis decompositions of completely or incompletely specified switching functions

Presented in this thesis are new approaches to column compatibility checking and column-based input/output encoding for Curtis decompositions of switching functions. These approaches can be used in Curtis-type functional decomposition programs for applications in several scientific disciplines. Examples of applications are: minimization of combinational and sequential logic, mapping of logic functions to programmable logic devices such as CPLDs, MPGAs, and FPGAs, data encryption, data compression, pattern recognition, and image refinement. Presently, Curtis-type functional decomposition programs are used primarily for experimental purposes due to performance, quality, and compatibility issues. However, in the past few years a renewal of interest in the area of functional decomposition has resulted in significant improvements in performance and quality of multi-level decomposition programs.

The goal of this thesis is to introduce algorithms that can significantly improve

the performance and quality of Curtis-type decomposition programs. In doing so, it is hoped that a Curtis-type decomposition program, complete with efficient, high quality algorithms for decomposition, will be a feasible tool for use in one or more practical applications.

Various testing and analyses were performed in order to evaluate the potential of algorithms presented in this thesis for use in a high quality Curtis-type decomposition program. Testing was done using a binary input, binary output Curtis-type decomposition program *MULTIS/GUD*. This program was implemented here at Portland State University by the Portland Oregon Logic Optimization Group.

NEW APPROACHES TO COLUMN COMPATIBILITY CHECKING AND COLUMN-BASED INPUT/OUTPUT ENCODING FOR CURTIS DECOMPOSITIONS OF COMPLETELY OR INCOMPLETELY SPECIFIED SWITCHING FUNCTIONS

by

MICHAEL A. BURNS

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

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1997

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank the people which have helped in many ways and who have steadfastly supported me during my thesis work. First and foremost I would like to thank Dr. Marek Perkowski, my advisor, for all of his guidance, encouragement, and inspiration. I would like to thank Dr. Douglas Hall for his feedback on my thesis. I would also like to thank each member of the Portland Oregon Logic Optimization group(POLO) for their many questions, answers, and feedback related to my thesis research. And I would like to thank Shirley, Laura, and Ellen for the help I received from them.

Last but not least, I would like to thank my parents for their unending support and encouragement throughout my thesis project.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The new approaches to column compatibility checking and column encoding presented in this thesis were designed to be part of a Curtis-style functional decomposition program. Because these two approaches are the highlights of this thesis, they are given the greatest coverage. However, before introducing these new approaches some basic information about functional decomposition should be mentioned.

Functional decomposition, first introduced in 1854 by Boole[6], has since been researched extensively for its potential applications to circuit minimization as well as a diverse variety of other interesting applications. One of the fascinating aspects of research in functional decomposition techniques is this variety of potential applications which cover a broad spectrum of scientific disciplines. The two primary areas of application in functional decomposition are in machine learning and circuit design.

In the machine learning area, there has been an increasing amount of research performed in order to determine the effectiveness of various decomposition methods for identification and classification of cancer cells and other harmful organisms,

identification of distant objects such as foreign aircraft, ships, tanks, etc., genetic code deciphering, pattern matching, automatic knowledge acquisition, theory formation and the development of various learning paradigms [12][23][35][48][64].

In circuit design, there are applications for functional decomposition in the minimization of combinatorial logic(switching functions) and state machines as well as applications in the mapping of logic to programmable logic devices such as PLAs, PALs, GALs, CPLDs, MPGAs, and FPGAs [15][18][33][36][50].

Though interesting as these applications are, specific details relating to each application will not be covered as they are beyond the scope of this thesis. However, interested readers may find out more about specific applications by consulting the corresponding references listed.

The following is a brief description of what is meant by functional decomposition so readers unfamiliar with functional decomposition may have a better idea of how it might be used in each application. In the most general sense, functional decomposition is the process of breaking down an initial problem description into a set of smaller sub-problem descriptions(according to some set of rules) which, composed back together are functionally equivalent to the initial description of the problem. For example, if an initial circuit description, having 100 inputs and 50 outputs, was to be implemented in a logic array where all gates in the array had 2 inputs and 1 output, a functional decomposition program could be used to break down the initial circuit description into a set of smaller sub-circuits(gates each with

2 inputs and 1 output). The resulting circuit description could then be mapped to the target device(provided, of course, that the device had the capacity to handle the number and type of gates obtained by the decomposition program). In Curtis-style decompositions, there are several steps(or phases) of execution. A few of the main steps, in the decomposition process are: input variable partitioning, column compatibility checking, column minimization, and column encoding.

One of the new approaches presented in this thesis addresses one of the main steps in the decomposition process referred to as column compatibility checking. The primary purpose of column compatibility checking is to build a compatibility or incompatibility graph which is used in the following step(column minimization) of a Curtis-style decomposition program. The nodes of the graph correspond to columns of the Karnaugh Map and the edges between the nodes indicate whether the nodes are compatible or incompatible. In the compatibility graph, edges between nodes indicate that they are compatible. In the incompatibility graph, edges between nodes indicate that they are incompatible.

In previous research of partition based methods, the column compatibility checking in Curtis-style decompositions was performed via an approach involving checking compatibility of columns in a pairwise fashion(one pair at a time for all pairs of columns). This approach is referred to as the "pair compatibility approach". An example of this method is described in Section 3.3. Additional examples of this approach approach can be found in a paper by Luba[31]. Presented

in this thesis is a new approach to column compatibility checking which can result in tremendous savings in execution time by simultaneously checking the compatibility of groups of columns at a time, rather than pairs of columns. This new approach is referred to as the "group compatibility approach". The approximate complexity of these two approaches are $O(n^2)$ and $O(n)$ respectively. Perhaps the greatest strength of this new approach is the ability to create, in a feasible way, large graphs which would otherwise be infeasible with the pair compatibility approach. Thereby, the ability to search areas of the solution space previously not possible using Curtis-style decompositions is greatly improved. It should be emphasized that the primary contribution of the new approach is to create the same data, but in a way which requires much less execution time.

Another significant advantage of the group compatibility approach is that it can also be used in a modified graph coloring approach for column minimization in order to greatly improve the time required to check for Ashenhurst(single output) decompositions when the number of nodes in the graph is large. In cases where the number of nodes in the graph is small, there is little or no gain by using this new approach. However, when the number of nodes in the graph is small, there is only a little need for fast column minimization techniques, because the execution time for this step is small in terms of the overall execution time of the decomposition.

The second new approach introduced in this thesis is the encoding of columns in a Curtis-style decomposition. The minimum necessary requirement of encoding

in a Curtis-style decomposition is to assign newly created intermediate variables so that an equivalence relation is maintained between the parent function and the decomposed sub-functions. The choice of encodings for these intermediate variables becomes increasingly important as the number of bits required for encoding increases. Proposed in this thesis is a new encoding approach which is intended to greatly improve decomposability of functions when the number of variables in the bound set is larger than the number of variables in the free set and when the function is at least 25% unspecified. While there is little or no advantage to this approach when functions are highly specified, this new approach can significantly improve decomposability of functions when they are highly unspecified. One may ask - "do such unspecified functions exist in real life ?" One answer to this question is yes, many functions in machine learning are very highly unspecified. Another answer to the same question is that any function can be made partially unspecified by simply performing a non-disjoint decomposition(i.e. adding at least one shared variable to both the bound set and the free set). The basis of this new approach is to increase the number of don't cares introduced into the predecessor function by encoding certain columns with the codes of multiple output classes(or symbols). For example, a certain column that is compatible with two of three output classes 00 and 01, may be given the codes 00, 01, or the combined code 0-. These don't cares can significantly reduce the minimum column multiplicity in the next level of decomposition by allowing a freedom to choose what values to assign, once it

has been determined what values will work best. Though this is not frequently possible with most columns, even a few extra don't cares can make the difference between a sub-function which is easy to decompose and one which is very difficult to decompose. A function which is more difficult to decompose may take a very long time to decompose and/or may result in a higher Decomposed Function Cardinality(DFC). DFC is one of the metrics used to provide a measure of the quality of a decomposition(where lower DFC is desired). In reference to the decomposition of a function representing a circuit, a high DFC indicates a complex circuit while a low DFC indicates a simpler circuit.

The format of this thesis is as follows: In Chapter 2 reference is made to the work of a few of the many researchers that have contributed significantly to the area of functional decomposition. Also, in Chapter 2, a brief overview of the basics of Curtis-type functional decomposition is given as well as some basic definitions and terminology that will be helpful to understand the following chapters. Chapters 3, 4, and 5 are the primary focal points of this thesis. These chapters introduce the three main algorithms in this thesis. Chapter 3 introduces a new approach to the column compatibility checking problem(**GCA** approach), followed by a comparison of this new approach with the classical approach. Presented in Chapter 4 is another significant algorithm which is an extension of the **GCA** approach presented in Chapters 3. This extension of the **GCA** approach is for improving efficiency in checking for Ashenhurst type decompositions. In Chapter 5, a new

approach to encoding(**DC_ENC**) is presented in detail(no results are presented for this algorithm as it has not been implemented yet). Introduced in Chapter 6 is the Curtis style functional decomposition program *MULTIS* which was designed here at Portland State University by the Portland Oregon Logic Optimization Group. Chapter 7 presents various experimental results for the program *MULTIS*. Finally, in Chapter 8 conclusions and future work are discussed.

CHAPTER 2

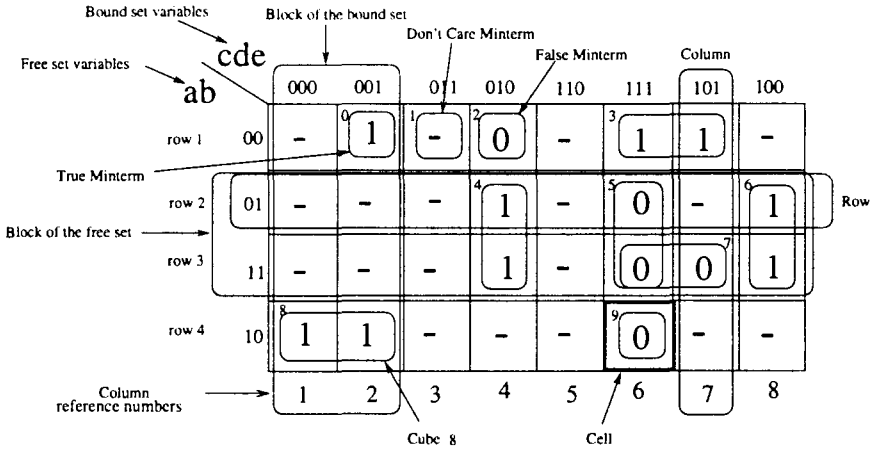
PREVIOUS APPROACHES TO FUNCTIONAL DECOMPOSITION

2.1 Definitions, Notations, and Terminology

Definition 2.1.1 *A* **Karnaugh Map** *is a rectangular array of cells which represents a truth table. There are* 2^n *cells in the map, where* n *is the number of input variables. The headings of the columns are the input variables corresponding to the bound set. The headings of the rows are the input variables corresponding to the free set. The values in each cell corresponds to the output value of the function for the corresponding set of input variables. For an example of a Karnaugh map, see Figure 2.1. The truth table for the function represented by the Karnaugh map is shown in Table 2.1. Note that in Table 2.1, cube number 1 has an unspecified output value. Typically, cubes are not shown in Truth Tables when all their output values are unspecified. However, it is shown in the table and in the Karnaugh map to illustrate what is referred to as a* $Don'tCareMinterm$.

Definition 2.1.2 *A* **Cell of a Karnaugh map** *is an individual square of the Karnaugh map corresponding to a product of input variables.*

Definition 2.1.3 Minterms *are product terms of function* F *for which all variables in the product term have specified values(i.e., 0 or 1 for binary functions).*



Karnaugh Map

Figure 2.1: Illustration of relevant terms used to reference the different parts of a Karnaugh Map

Definition 2.1.4 Don't Care Outputs *are output variables which are unspecified for a given product of input variables. These output variables are commonly shown as a dash "-" in a Truth table or Karnaugh map.*

Definition 2.1.5 Don't Care Inputs *are input variables which are unspecified for a given product of input variables. These input variables are commonly shown as a dash "-" in a truth table.*

Definition 2.1.6 Cares *refer to products of input variables for which at least one output variable is specified.*

Definition 2.1.7 Cubes *correspond to the products of arbitrary literals.*

Definition 2.1.8 Free set variables *are the subset of the input variables which correspond to the rows of the Karnaugh map.*

cube	a	b	c	d	e	F
0	0	0	0	0	1	1
1	0	0	0	1	1	-
2	0	0	0	1	0	0
3	0	0	1	-	1	1
4	-	1	0	1	0	1
5	-	1	1	1	1	0
6	-	1	1	0	0	1
7	1	1	1	-	1	0
8	1	0	0	0	-	1
9	1	0	1	1	1	0

Table 2.1: Truth table corresponding to the Karnaugh map in Figure 2.1

Definition 2.1.9 Bound set variables *are the subset of the input variables which correspond to the columns of the Karnaugh map.*

Definition 2.1.10 Output set variables *are the variables which specify the output values for each cube or minterm in a function.*

Definition 2.1.11 Partition Π *on a set* S *is a collection of disjoint subsets whose set union is* S*. The disjoint subsets are called the blocks of* Π*.*

Definition 2.1.12 Rough partition Π *on a set* S *is a collection of nondisjoint subsets whose set union is* S*. The nondisjoint subsets are called the blocks of* Π*.*

Partition Blocks: There are different types of partition blocks(i.e., blocks of the free set, bound set, output set, cover set, etc...). Also, the type of elements within these blocks can be different. For example, a block of the cover set can be composed of cubes or columns.

Product of partitions denoted by $X \cdot Y$, results in a new partition where each block in the new partition is formed from the intersection of the each of the blocks in X with each of the blocks in Y.

Intersection of partition blocks denoted by $X_i \cap Y_j$, results in a partition block having all elements which are common to both X_i and Y_j.

Union of partition blocks, denoted by $X_i \cup Y_j$, results in a partition block having all elements which are in either X_i or Y_j.

Definition 2.1.13 *Let B be a subset of the set of inputs X. An input partition generated by set B is denoted as:*

$$P(B) = \prod_{x \in B} P(x) \tag{2.1}$$

where $\prod$ denotes the product of partitions. $P(x)$ is a partition for variable x.

P(B) = Rough partition on the bound set variables, where **B** is the set of variables for the bound set. Individual partition blocks are composed of cubes.

P(A) = Rough partition on the free set variables, where **A** is the set of variables for the free set. Individual partition blocks are composed of cubes.

P(F) = Rough partition on the output set variables, where **F** is the output set of variables. Individual partition blocks are composed of cubes.

Definition 2.1.14 Blocks of the bound set *are the subsets of the rough partition P(B). These blocks of cubes represent sets of one or more columns which*

have at least one cube in common with a specific column X. No column in the same block as column X contains any cubes which are not contained in column X. Stated formally: Let X denote the set of cubes which are contained in column X and let Y be some other set of cubes which are contained in column Y. Then, any column Y may be contained in the same block of the bound set as column X iff $Y \subseteq X$.

Definition 2.1.15 Blocks of the free set *are the subsets of the rough partition P(A). These blocks of cubes represent sets of one or more rows which have at least one cube in common with a specific row X. No row in the same block as row X contains any cubes which are not contained in row X. Stated formally: Let X denote the set of cubes which are contained in row X and let Y be some other set of cubes which are contained in row Y. Then, any row Y may be contained in the same block of the bound set as row X iff $Y \subseteq X$.*

Definition 2.1.16 Blocks of the output set *are the subsets of the rough partition P(F).*

Definition 2.1.17 Blocks of cubes *are subsets of cubes within a rough partition.*

Definition 2.1.18 *A* **Compatible Class (CC)** *is a set of elements which are mutually compatible. A CC of columns is a set of columns which are mutually compatible.*

Definition 2.1.19 *A* **Maximum CC (MCC)** *is a CC that cannot be covered by any other CC(i.e. a MCC is a CC that is not contained in any other CC).*

Definition 2.1.20 Cofactor: *A submap of the Karnaugh map corresponding to some combination of input variables is referred to as a* **cofactor**. *If this is a combination of free variables, it will also be called a row - referring to a Kmap. If this is a combination of bound variables, it will also be called a column.*

Definition 2.1.21 Compatible Cubes: *Two cubes are said to be compatible if they belong to the same block of the partition on the free set being considered and the same two cubes belong to the same block in the output partition. Two cubes are said to be incompatible if they belong to the same block of the partition on the free set being considered and the same two cubes do not belong to the same block in the output partition.*

Definition 2.1.22 Compatible Columns: *Two columns of a Karnaugh Map are said to be compatible columns if every pair of outputs in corresponding cells (i.e., outputs corresponding to the same combination of input variables in the free set) are compatible. In a binary single-output function, two outputs are compatible as long as they are not complements of each other.*

Definition 2.1.23 *A* **Cover Set** *is a set of CCs such that the union of all CCs in the set is equal to the set of all cubes(or columns) in the function.*

Definition 2.1.24 Column Multiplicity *is a number representing the number of CCs in a cover set.*

2.2 Fundamentals Of Curtis Decomposition

The purpose of the following subsections is to introduce the fundamentals of Curtis decomposition which are necessary for understanding the main algorithms presented in Chapters 3, 4, and 5. In Section 2.2.1, a brief introduction to Curtis decomposition is presented. In Section 2.2.2, partition calculus for Curtis decomposition is presented.

2.2.1 Introduction

Shown in Figure 2.2g is a block diagram representation of a Curtis decomposition of the function F shown in Figure 2.2f. The sub-function G is referred to as the predecessor sub-function and the sub-function H is referred to as the successor sub-function. In a Curtis decomposition, the number of outputs from the sub-function G is required to be less than the number of inputs to the sub-function G. $H(A, G(B, C))$ represents the decomposed function such that $F = H(A, G(B, C))$. The sets A and B are the *free set variables* and *bound set variables*, respectively where $A \cup B = X$, and $A \cap B = \emptyset$. The set C is some subset of A and is referred to as the *shared set of variables.* When set C is an empty set, the decomposition is called a disjoint decomposition. If set C is not an empty set, the decomposition

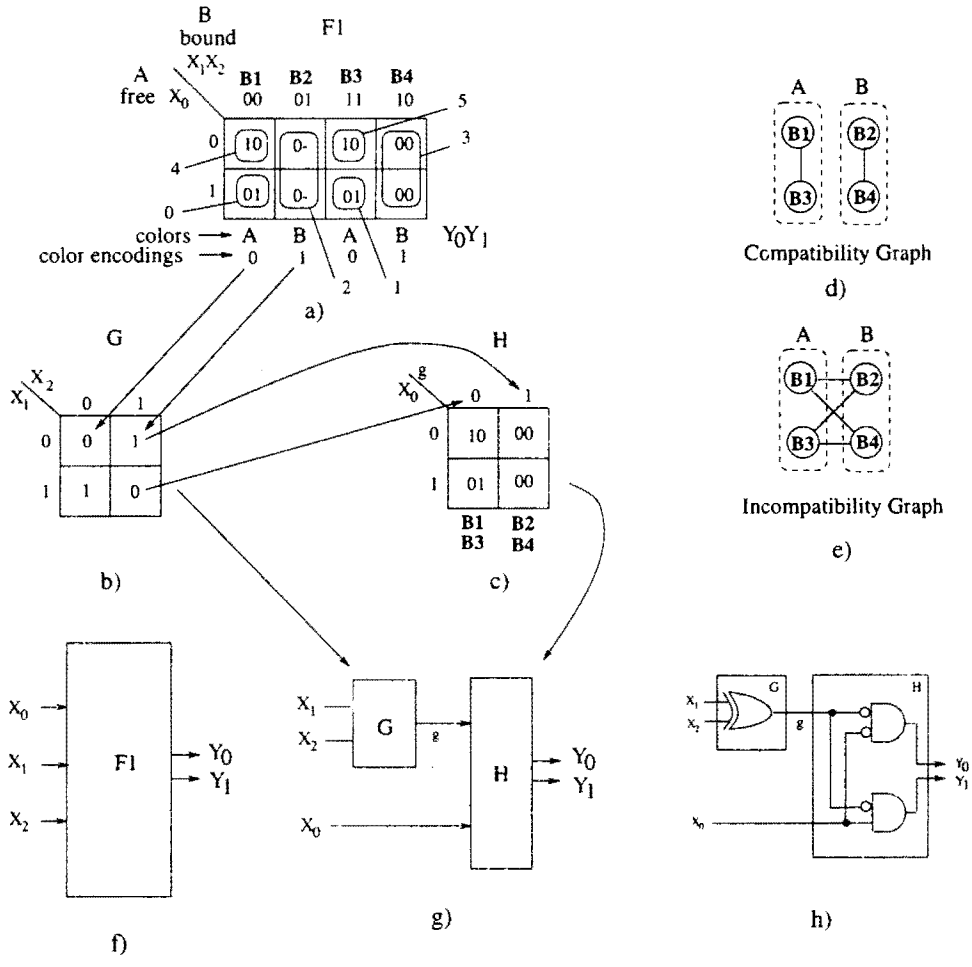


Figure 2.2: Curtis Decomposition of function $F1$ corresponding to Table 2.2

is called a nondisjoint decomposition.

The following is a brief explanation of the decomposition process in a Curtis style decomposition:

Shown in Figure 2.2a is the Karnaugh map of Function $F1$ with the given free

set and bound set shown. Cube numbers are shown for each cube represented in the Karnaugh map. The primary objective, in a Curtis style decomposition, is to break up a function F into two smaller sub-functions G and H. Block diagrams of these sub-functions are shown in Figure 2.2g. The decomposition of one function into two smaller functions is considered one *loop* in the decomposition process. In a complete decomposition of functions with many input variables, many *loops* in the decomposition process may be executed before the sub-functions created are of acceptable size(i.e., less than or equal to a user specified size).

The function $F1$ is broken up into two smaller sub-functions by finding reduced sets(CCs or $MCCs$) of compatible columns from the columns found in function $F1$. Ideally, it is desired to find the minimum or near minimum number of CCs as the cover set. For now, assume the goal is to find the minimum number of CCs. For this simple function, it is easy to see that column B_1 is compatible with B_3 and column B_2 is compatible with B_4. There are no other combinations of columns which are compatible and therefore there are only two column types(CCs). These are shown in Figure 2.2c with the labels of columns that are compatible with each column type shown below each column. This is the successor sub-function H sometimes simply referred to as the H block for brevity.

The compatibility relationship between these columns is represented in a compatibility graph as in Figure 2.2d. Here, the nodes correspond to the columns and the edges between the nodes indicate that the columns are compatible. The groups

that are enclosed by the dotted lines correspond to the two column types(or CCs of columns) found in Figure 2.2c. Shown for completeness is the incompatibility graph in Figure 2.2e which is simply the complement of the compatibility graph. In the incompatibility graph, nodes represent columns as in the compatibility graph, but the edges between the nodes indicate that columns are incompatible.

The inputs to the H block are the free set variables, X_0 in this case, and the encoded outputs from the G block, variable g in this case. The outputs of the H block are the outputs Y_0Y_1 from function $F1$. The encoded outputs of the G block are the encodings assigned to the column types in the original function. For simplicity, random codes are assigned to the column types in the original function. These encodings are shown below each of the columns shown in Figure 2.2a and in the cells of the Karnaugh map in Figure 2.2b. The inputs to the G block are the bound set variables which correspond to each of the columns in the original function(Function $F1$). Once the encoding of CCs in the cover set has been completed, then the current *loop* in the decomposition process is finished. Shown in Figure 2.2h is a circuit representing the logic in the decomposed function.

2.2.2 Partition Calculus Formalism For Decomposition

In the following subsections, an example decomposition is illustrated in detail using partition calculus. The function $F1$, used in the following subsections, is the same as used previously in Section 2.2.1. The next example is illustrated using par-

tition calculus because the main algorithms presented in this thesis are partition-based. Example 2.2.1 is split up into three parts. Section 2.2.2.1 illustrates the formation of the free set, bound set, and output set partitions. Section 2.2.2.2 illustrates the formation of the cover set Π_G. Section 2.2.2.3 illustrates the encoding of classes in the cover set.

2.2.2.1 Forming Partitions For The Input and Output Variables

Example 2.2.1

	X_0	X_1	X_2	Y_0	Y_1
0	1	0	0	0	1
1	1	1	1	0	1
2	-	0	1	0	-
3	-	1	0	0	0
4	0	0	0	1	0
5	0	1	1	1	0

Table 2.2: Truth table for function $F1$

Given is the truth table(Table 2.2) corresponding to the Karnaugh map in Figure 2.2a. The numbers of rows(cubes) in the table are shown for each cube of the Kmap.

In partition-based Curtis decompositions, partitions are formed for the bound set, the free set, and the output set of variables. In order to obtain the partitions for the bound set, the free set, and the output set, partitions are first formed for each of the input variables and output variables. The blocks of each partition are

formed by grouping together all cubes which have the same input values for the variable under consideration. Don't cares are considered to be the same value as both the on-set and the off-set. Semicolons are used to separate individual blocks of each partition. The following are the rough partitions for the input variables describing the function represented in the table.

$$P(X_0) = (X_0 = 0; X_0 = 1) = (2,3,4,5;\ 0,1,2,3);$$

$$P(X_1) = (X_1 = 0; X_1 = 1) = (0,2,4;\ 1,3,5);$$

$$P(X_2) = (X_2 = 0; X_2 = 1) = (0,3,4;\ 1,2,5);$$

And for output variables:

$$P(Y_0) = (Y_0 = 0; Y_0 = 1) = (0,1,2,3;\ 4,5);$$

$$P(Y_1) = (Y_1 = 0; Y_1 = 1) = (2,3,4,5;\ 0,1,2);$$

For a given bound set B = $\{X_1, X_2\}$ and free set A = $\{X_0\}$, the partitions of the bound set, free set, and output set of variables are as follows:

$$P(A) = P(X_0) = (X_0 = 0; X_0 = 1) = (2,3,4,5;\ 0,1,2,3);$$

$$P(B) = P(X_1X_2) = (X_1X_2 = 00;\ X_1X_2 = 01;\ X_1X_2 = 10;\ X_1X_2 = 11)$$

$$= (0,4;\ 2;\ 1,5;\ 3);$$

$$P(F) = P(Y_0Y_1) = (Y_0Y_1 = 00; Y_0Y_1 = 01; Y_0Y_1 = 10; Y_0Y_1 = 11)$$
$$= (2,3;\ 0,1,2;\ 4,5;\ \emptyset);$$

2.2.2.2 Forming The Cover Set Π_G

Theorem 2.2.1 *Functions G and H represent a serial decomposition of function F, i.e., $F = H(A, G(B, C))$ if there exists a partition $\Pi_G \supseteq P(B \cup C)$ such that $P(A) \cdot \Pi_G \subseteq P(F)$, where all the partitions are over the set of cubes and the number of two-valued output variables of component Π_G is equal to $g = \lceil \log_2 L(\Pi_G) \rceil$, here $L(\Pi)$ denotes the number of blocks of partition Π, and $\lceil x \rceil$ denotes the smallest integer equal to or larger than x.*

Here $\lceil \log_2 L(\Pi_G) \rceil$ gives us the number of output signals from function G. For Curtis type decompositions, there is an additional requirement that the number of outputs of G must be less than the number of inputs of G.

To find a cover set Π_G, it is necessary to find a set of CCs which covers all the blocks in $P(B \cup C)$ that also satisfies Theorem 2.2.1 (i.e., $P(A) \cdot \Pi_G \subseteq P(F)$). To solve this problem, consider a subset of primary inputs, $B \cup C$, and the q-block partition $P(B \cup C) = (B_1, B_2, \ldots, B_q)$ generated by this subset. Then use a relation of compatibility of partition blocks to form CCs.

Definition 2.2.1 Compatibility relation: *Two blocks B_i and $B_j \in P(B \cup C)$ are* **compatible** *iff merging blocks B_i and B_j into a single block satisfies*

$P(A) \cdot (B_i \cup B_j) \subseteq P(F)$.

For example, from Table 2.2:

$$P(A) = P(X_0) = (2,3,4,5;\ 0,1,2,3);$$

$$P(B) = P(X_1X_2) = (0,4;\ 2;\ 1,5;\ 3) = (B_1, B_2, B_3, B_4);$$

Merging B_1 and B_2 together results in:

$$(B_1 \cup B_2) = (0,2,4);$$

$$P(A) \cdot (B_1 \cup B_2) = (2,4;\ 0,2) \not\subseteq P(F) \ = \ (2,3;\ 0,1,2;\ 4,5;\ \emptyset);$$

therefore B_1 and B_2 are incompatible, denoted as $B_1 \not\sim B_2$.

In the same way, the compatibility relation can be used to check other pairs of blocks in $P(B)$. The result is: $B_1 \not\sim B_2$, $B_1 \sim B_3$, $B_1 \not\sim B_4$, $B_2 \not\sim B_3$, and $B_2 \sim B_4$.

From the set of pair-wise compatible blocks, form compatible classes. Thus $MCC1 = \{B_1, B_3\}$, $MCC2 = \{B_2, B_4\}$. Note that in general, these classes do not need to be MCCs. However, in this simple case, the CCs are also MCCs. From these $MCCs$, the minimal cover set can be found:

$$\Pi_G = \{\{B_1, B_3\}, \{B_2, B_4\}\} = (0,1,4,5;\ 2,3).$$

Now check to see if the conditions in Theorem 2.2.1 are satisfied.

Because

$$P(B) < \Pi_G \tag{2.2}$$

and

$$P(A) \cdot \Pi_G = (4,5;\ 0,1;\ 2,3;\ 2,3) < P(F) = (2,3;\ 0,1,2;\ 4,5;\ \emptyset); \tag{2.3}$$

then this is a feasible decomposition with bound set $B = \{X_1, X_2\}$ and free set $A = \{X_0\}$. Therefore, $F = H(X_0, G(X_1, X_2))$. Notice also that Π_G corresponds to a single output function G because there are two blocks in Π_G(i.e., $\lceil \log_2 L(\Pi_G) \rceil = \lceil \log_2 2 \rceil = 1$).

2.2.2.3 Encoding of Compatible Classes

The process of encoding compatible classes in the cover set allows a given function to be split up into two smaller subfunctions which are equivalent to the original function. Put simply, encoding in Curtis style decomposition is the process of assigning codes to columns such that there is a mapping of columns from the original function to a reduced set of columns which form the successor subfunction H. The mapping of columns from the original function to the successor subfunction is done

via the output codes assigned to the predecessor subfunction G. Encoding of compatible classes can be very important when the number of bits in the encodings is greater than one. However, when there is only one bit required for encoding, then encoding becomes trivial. In this example, encoding is trivial because only one bit is required to encode the classes in Π_G. Therefore, class $MCC1$(color A) is arbitrarily assigned to binary code "0" and $MCC2$(color B) is assigned to binary code "1". Next, all columns are given the code assigned to the MCC they belong to. The codes assigned to columns are shown in Figure 2.2c.

Next it is very easy to find functions g, Y_0, and Y_1 from the Karnaugh maps G and H shown in Figure 2.2b and Figure 2.2c. Hence,

$$g = X_1 \ominus X_2, \quad Y_0 = \bar{g}\bar{X}_0, \quad Y_1 = \bar{g}X_0$$

The circuit represented by these functions is shown in Figure 2.2h. The following is a definition of a metric(DFC) which is used to evaluate the quality of a decomposition.

Definition 2.2.2 Decomposition Function Cardinality (DFC) *is sometimes used as a measure to evaluate the quality or effectiveness of a decomposition (where a lower DFC is desired). More specifically, it is an integer value which is equal to*

$$DFC = \sum_{i=1}^{n} 2^{I_i} \times O_i$$

where I_i and Oi are the number of inputs and outputs to each logic block i, and n is the total number of blocks.

Stated simply, the total DFC of a function is equal to the sum of the $DFCs$ of the individual logic blocks in the function. It should be noted that the blocks referred to in the definition for DFC are logic blocks corresponding to a block diagram(i.e., not partition blocks). To accurately assess the quality of decompositions, one needs to know specific requirements of the problem such as power, timing, delay, area, etc. Better metrics for assessing the quality of decompositions have been proposed [54]. However, for simplicity and didactic purposes, DFC as defined here is used as the primary metric for the quality of decompositions.

We can compare the DFC of the original function $F1$ to the DFC of the decomposed set of logic blocks G and H by applying the formula to the logic blocks shown in Figure 2.2f and Figure 2.2g.

$$DFC_{F1} = \sum_{i=1}^{1} 2^{I_1} \times O_1 = 2^4 \times 2 = 32.$$

$$DFC_{G+H} = \sum_{i=1}^{2} 2^{I_i} \times O_i = (2^2 \times 1) + (2^2 \times 2) = 4 + 8 = 12.$$

Therefore, in this example, the DFC of the decomposed function is much lower than that of the original function. More often than not, the DFC of a decom-

posed function will be lower than the undecomposed function. More meaningful applications of DFC are in comparisons of alternative decompositions of the same function, where the decomposition resulting in lower DFC are considered to be better quality decompositions.

2.3 Basic Functional Decomposition Types

In the following subsections, the primary decomposition types are introduced along with mention of some of the researchers who have been credited with significant contributions to each of the decomposition types.

2.3.1 Serial Decomposition

Basically, a serial decomposition is a decomposition of function F into two sub-functions, a predecessor sub-function G, and a successor sub-function H. The outputs of the sub-function G are inputs to the sub-function H. In Figure 2.3, the basic forms of serial decompositions for disjoint and nondisjoint cases are shown. These forms apply for each of the serial decomposition approaches mentioned in the following sections.

2.3.1.1 Ashenhurst-Curtis Approach

The approach by Curtis is a serial type decomposition approach with the predecessor sub-function having $b + c$ inputs and g outputs where g is not more than

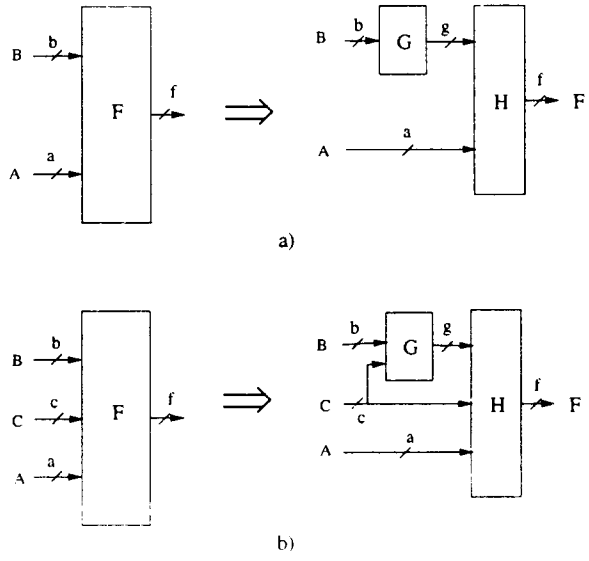


Figure 2.3: Ashenhurst-Curtis type Decompositions: a) Disjoint Decomposition b) Non-Disjoint Decomposition.

$(b+c-1)$. The successor sub-function has $a+c+g$ inputs and f outputs where f is the same number of outputs as the original function before it is decomposed. For disjoint decompositions, the shared set of input variables C is empty and the decomposition appears as in Figure 2.3a. For nondisjoint decompositions, the shared set of input variables C is common to the set of input variables to both G and H sub-functions. An example of this type of serial decomposition is shown in Figure 2.3b.

An Ashenhurst decomposition is a special case of the Curtis type decomposition. In the Ashenhurst decomposition, there exists exactly one output from the predecessor sub-function, provided of course that a decomposition exists. A decomposition is said to exist for an Ashenhurst decomposition if the column multiplicity is equal to two. This requires only one bit(output) to encode the two

column types.

Ashenhurst and Curtis are two of the early researchers which contributed significantly to the research in the area of functional decomposition. Ashenhurst introduced the single output serial decomposition in 1959[3]. This was followed soon after by the introduction of the more general Curtis style decomposition[13]. These researchers are responsible for much of the basic formalisms for functional decomposition and many of the proofs for the existence theorems.

2.3.1.2 Roth-Karp Approach

The approach by Roth and Karp is another example of the Ashenhurst-Curtis style decomposition. Their approach differs very little from the basics of the Ashenhurst-Curtis decomposition. However, one important distinction is that the approach by Roth and Karp restricts decomposed sub-functions to a set of pre-characterized sub-functions and simple gates. Also, they use cube calculus to represent functions instead of Karnaugh maps. More emphasis is given to algorithm development that would be practically efficient. Those interested may read more about the Roth-Karp approach in the following papers [25][26].

2.3.1.3 Lai-Pedram-Vrudhula Approach

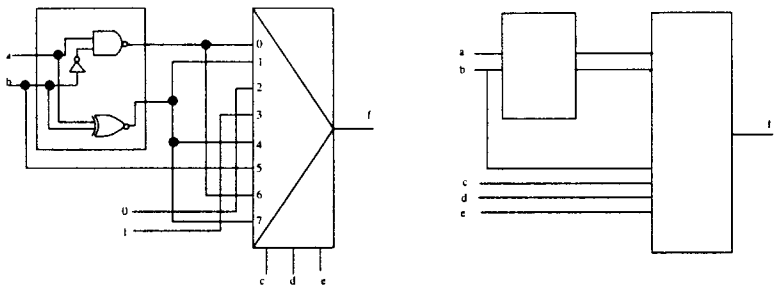
The Lai-Pedram-Vrudhula approach[34] is basically an Ashenhurst-Curtis approach applied to a BDD data structure. Their results presented at DAC '93

revealed the most efficient approach so far to Ashenhurst-Curtis decompositions for very large functions. The program was able to quickly decompose functions which were previously not possible by other Ashenhurst-Curtis decomposers due to excessive running times. Though the Lai-Pedram-Vrudhula approach is very efficient at decomposing very large functions, it may result in poor quality solutions in terms of number of gates or logic blocks. It would be interesting to compare the quality of their solutions with other approaches. Unfortunately, no results were given on small functions and no results for number of logic blocks or gates were given on large functions. Therefore, it is not easy to compare the quality of their solutions with the solutions of others.

In the Lai-Pedram-Vrudhula approach, completely specified functions are represented using a single BDD while incompletely specified functions are represented using an ON-BDD and an OFF-BDD. Partitioning of variables to bound and free sets is done by ordering variables in the BDD(s) and then checking the column multiplicity using a min-cut method. The cut-set yielding the minimum cut(i.e., minimum column multiplicity) is used to divide the input variables into free and bound sets. The BDD is then split into two BDDs, one BDD from the top half above the cut and the other from below the cut. After a random encoding scheme is used to encode the predecessor sub-function, these new BDDs represent the decomposed function as a combination of a predecessor sub-function and a successor sub-function.

2.3.1.4 Luba Approach

The serial decomposition approach by Luba [30][31][33] is another example of an Ashenhurst-Curtis style decomposition. Unlike the BDD-based approach by Lai-Pedram, Luba uses a partition-based representation with corresponding partition-based operations. The basic steps involved in Luba's serial decomposition approach are summarized below. The algorithm first uses heuristic criterion to partition the input variables to bound and free sets A and B, where the set B is the set of input variables to the sub-function G and the set A is the set of input variables to the sub-function H. Next, for an assumed set of shared variables C, it calculates the maximum compatible classes for the blocks of partition $P(B \cup C)$ and a minimal cover of compatible classes. If the column multiplicity(number of blocks in that cover) is determined to be acceptable by the program user, then the next phase in the decomposition process is executed. If the column multiplicity is not determined to be acceptable by the user, then the program re-partitions the input variables to new bound and free sets A and B. In the first run of the decomposer, the program checks the existence of a disjoint decomposition by assuming $C = 0$. If such a decomposition does not exist, then they add additional input variables to the set C until the decomposition existence criterion is satisfied.



a) PUB Decomposition

b) Block Diagram Representation

Figure 2.4: Example of PUB Decomposition

2.3.1.5 PUB Approach

The approach of Perkowski-Uong-Brown(PUB) [38][43], in contrast to previous approaches, is not a variant of the Ashenhurst-Curtis style decomposition. Unlike the other serial type approaches, the PUB approach uses a conceptual multiplexor-based decomposition scheme. The only similarity between the PUB approach and other serial type approaches is the use of partitioning of variables to free and bound sets. In Curtis decomposition the multiplicity index is for cofactors of bound variables, whereas in PUB decomposition the multiplicity index is for cofactors of free variables. This approach has potential for significant savings of area in cases where many inputs to the multiplexors share common sub-functions. Observe that the sharing of sub-function outputs in the multiplexor scheme shown in Figure 2.4. Savings can be quite significant when applied to functions of many variables.

2.3.2 Parallel Decomposition

In simple terms, a parallel decomposition involves splitting up a multi-output function into two or more functions each having a subset of the output variables of the original function. More about specific parallel decompositions is mentioned in the following sections.

It is important to note that most of the literature published in the area of functional decomposition has concentrated on the serial type as opposed to parallel type decompositions. This is perhaps because most decompositions can be performed exclusively using a serial type decomposer. However, the reverse is not true. Therefore, the serial decomposition is considered more important in the overall decomposition scheme. However, parallel decompositions can be of great importance when partitioning subsets of outputs to be later decomposed together serially. By selecting subsets of output variables which are partially symmetric, successive serial decompositions on the subsets of outputs may result in simpler circuit descriptions. The simpler circuit descriptions result as a consequence of sharing part of the overall logic between one or more functions.

2.3.2.1 N_TO_ONE Approach

The most basic of parallel decomposition approaches is to split an n-output function into n single output functions. This type of decomposition is straight forward and therefore requires no further explanation. The general form of this

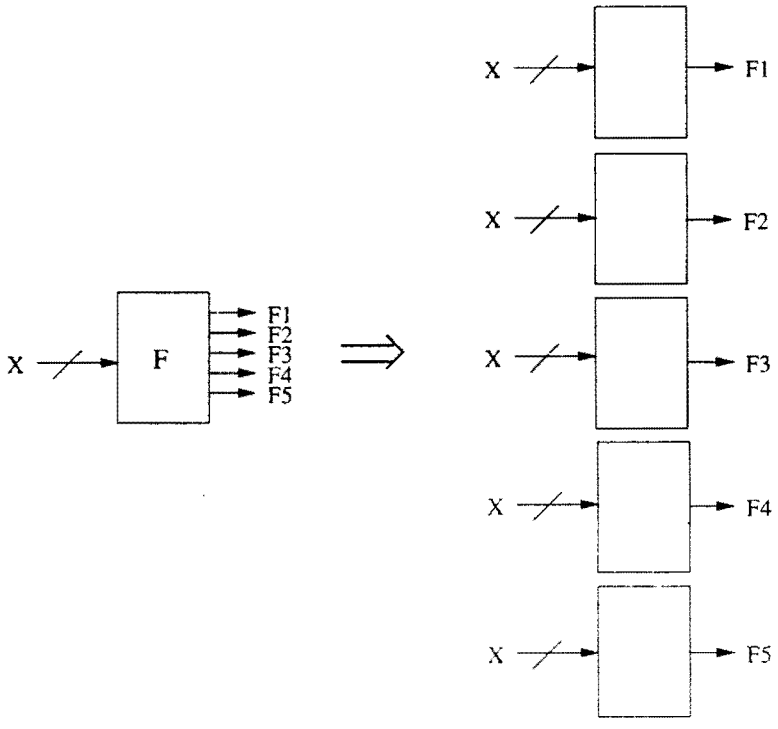


Figure 2.5: N_TO_ONE Parallel Decomposition

type of decomposition is shown in Figure 2.5.

2.3.2.2 Luba Approach

This approach splits up a given n-output function into 2 sub-functions, one sub-function having r outputs and the other sub-function having $n - r$ outputs. Each of the parallel sub-functions may have different sets of input variables. For example, given an original function which has input variables 0 thru 8 and output variables F1, F2, F3, F4, and F5. Then parallel sub-function P1 might have input variables 0,2,3,4,5,8 with output variables F1 and F2. Similarly, sub-function P2 might have input variables 0,1,2,3,4,6,7 with output variables F3, F4, and F5. This

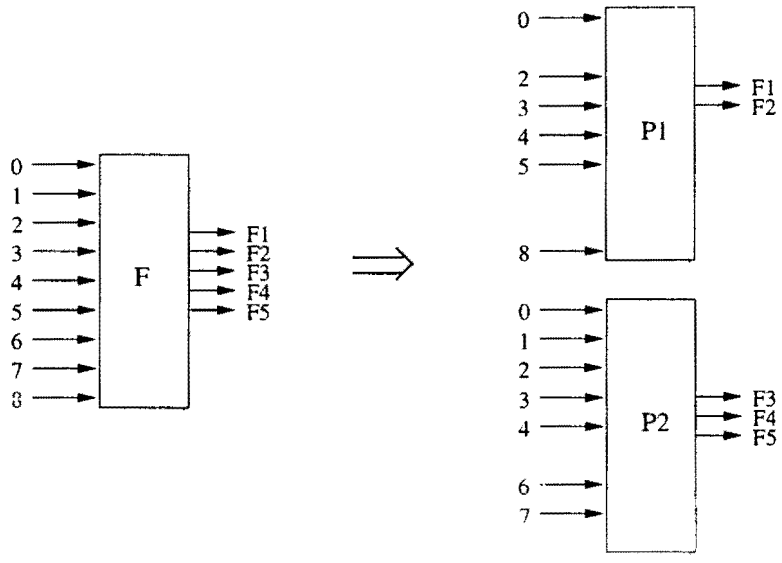


Figure 2.6: Luba Parallel Decomposition

is shown in Figure 2.6. Note that not all input variables are required for each of the parallel sub-functions. This means that certain outputs are only dependent on a subset of the full set of input variables. The subset of input variables that a function is dependent on is referred to as the support set for that function. The basis of Luba's parallel approach[33] is to partition functions into two sets such that those functions which are grouped together have as similar support sets as possible.

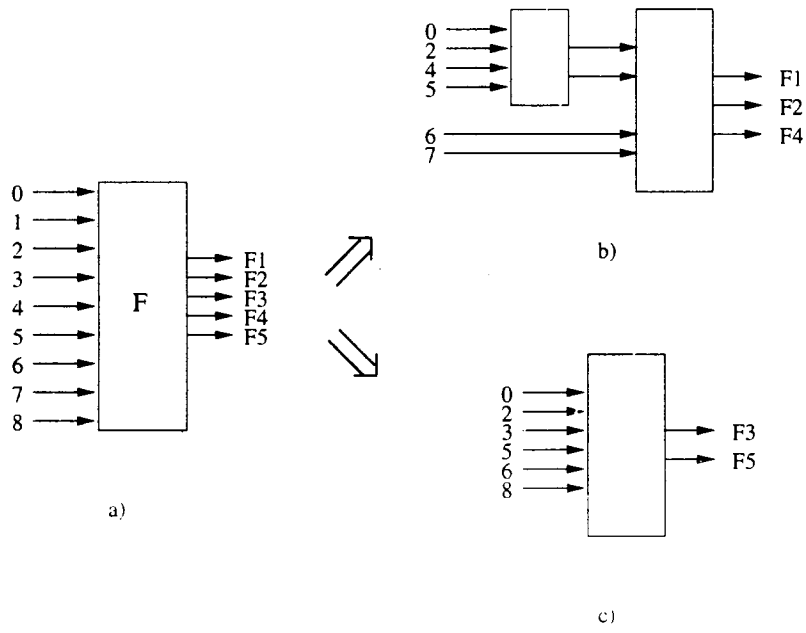


Figure 2.7: Perkowski Serial-Parallel Decomposition

2.3.3 Serial-Parallel Decomposition

2.3.3.1 Perkowski Approach

The result of this type of approach on a hypothetical function F is shown in Figure 2.7. The basic idea of this approach is the following: An incompatibility graph is created for a multi-output function(F1,F2,F3,F4,F5), with edges labeled by output function names. A subset of labels, F3 and F5, are removed to decrease the multiplicity index of the graph. Based on the new incompatibility graph for labels F1, F2, and F4, the decomposition from Figure 2.7a is found. The result of the decomposition for labels F1, F2, and F4, is shown in Figure 2.7b. Now the vacuous input variables are removed from functions F3 and F5 which leads to the

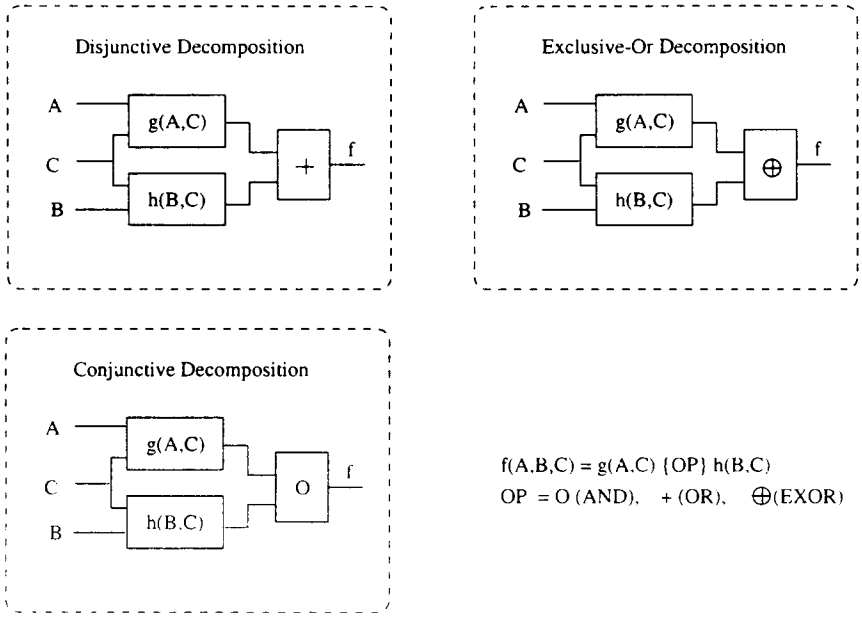


Figure 2.8: Strong Disjunctive, Conjunctive and Linear Decompositions.

block in Figure 2.7c.

Though this approach may be the most complicated, it may find better combinations of functions to be decomposed by serial decompositions thereby resulting in better overall decompositions. Moreover, this approach combines the parallel and serial decompositions. Parallel decomposition is not assumed here, it results from general analysis of function partitioning based on graph coloring[55].

2.3.4 Gate Decomposition

2.3.4.1 Steinbach-Bochmann Approach

The Steinbach-Bochmann decomposition approach is not an Ashenhurst-Curtis type of decomposition. It is a distinctly different type of decomposition which de-

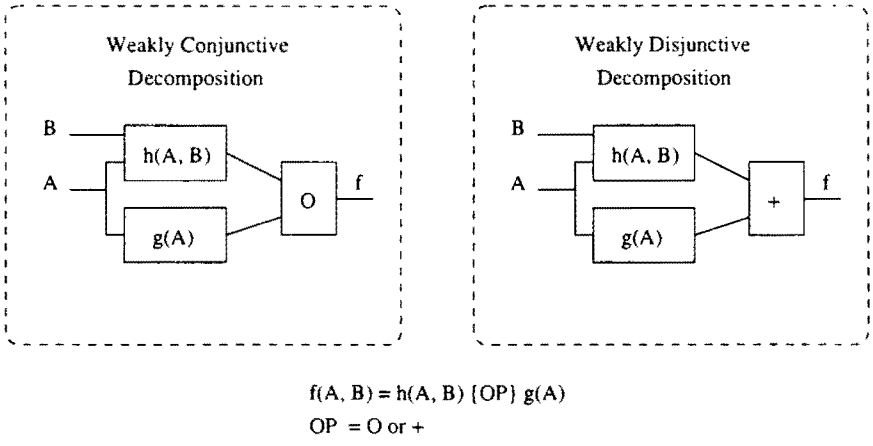


Figure 2.9: Conjunctive and Disjunctive Weak Decompositions.

composes functions into a set of 2-input sub-functions(or gates). Similar to the PUB decomposition, the Steinbach-Bochmann decomposition borrows the concept of variable partitioning to free and bound sets from Ashenhurst-Curtis. Only four types of gates are allowed(AND, OR, EXOR and NOT). Unlike the Ashenhurst-Curtis type of decomposers which decompose functions into two sub-functions(1 predecessor and 1 successor) for each loop in the decomposition process, the Steinbach-Bochmann decomposer decomposes functions into three sub-functions for each loop in the decomposition process(2 predecessors and 1 successor). In their approach, the successor sub-function is always one of the gate types mentioned above. Another important distinction is that their approach does not require column compatibility checking, column minimization or encoding like the Ashenhurst-Curtis decomposers. However, their approach requires partitioning to a free set, a bound set, and a shared set as does the Ashenhurst-Curtis approaches.

Readers interested in details of the gate type decomposition can refer to papers by Steinbach-Bochmann [62].

2.4 Conclusions On Previous Work

Presented in previous sections were several different approaches to functional decomposition. Which approach performs better in the greatest number of performance categories is not known due to the variety of applications. Examples of categories are: lowest power consumption, fewest levels, fewest rows, minimum delay, minimum number of 2-input gates, minimum DFC, minimum area, minimum number of logic blocks which can be mapped to specific programmable devices, minimum execution times, and the ability to decompose functions of many input and output variables(100 inputs and 100 outputs) without running out of memory.

Ideally, it is desired to have a single decomposition program which performs better in the greatest number of categories and is easily modified to enhance performance in one or more specific categories of interest. Though it has not been proven, it has been speculated that a Curtis type decomposer can be used to obtain any form of decomposed function found using other decomposition approaches. If this was known to be true, then this would be very valuable information because then it would be theoretically possible to find decomposition solutions, with a Curtis type decomposer, at least as optimal as other approaches for many of the performance categories. A formal proof that a Curtis type decomposer has the

potential to obtain better results than other decomposition approaches would be very time consuming to formulate, regardless whether it is true or not. However, results already obtained show that in some instances, even a very basic implementation of a Curtis type decomposer MULTIS/GUD[52] was able to perform better than a program implemented using the Steinbach-Bochmann approach.

In Chapter 7, results are presented which compare the Steinbach-Bochmann approach with MISII and Ashenhurst-Curtis programs GUD, GUD_MV, and TRADE. Programs GUD and TRADE were implemented by the POLO(Portland Oregon Logic Optimization) group at at Portland State University. Program GUD_MV was implemented by Stanislaw Grygiel at Portland State University. Unlike the other decomposition programs compared, the program GUD_MV is a decomposition program which outputs results in a multi-valued format. consistently, the multi-valued program GUD_MV performed better than the other decomposers compared in terms of DFC. The Steinbach-Bochmann approach performed better than the binary decomposers compared(second only to the multi-valued program GUD_MV). However, excluding the results for the multi-valued program GUD_MV, the program MULTIS/GUD was able to perform the best in a few instances in terms of DFC. It should be noted that the results for program GUD were obtained using pseudo random partitioning and encoding schemes. Also, the general strategy of GUD was very simple and was not developed extensively due to the large amount of time spent programming and debugging the basic framework of the program and

conversions of input/output formats of other decomposers for the main testbed program MULTIS. With better quality decomposition strategy, partitioning, and encoding schemes, results for program GUD would probably result in much better solutions, perhaps even better than the other approaches compared. It should be noted however, that until better quality schemes are implemented, only speculation can be made about possible improvements in results from program GUD. Future work is concerned with the following issues:

- General Decomposition Strategy
- Variable Partitioning To Bound and Free Sets
- Column Compatibility Checking
- Column Based Input/Output Encoding

While algorithms for general decomposition strategies and variable partitioning are of great importance to the quality of decompositions, they are not the primary topics of this thesis and therefore are not covered in any detail. The primary topics of this thesis address the issues of column compatibility checking and column based input/output encoding for Ashenhurst-Curtis type decompositions. One of the algorithms presented in this thesis was designed to speed up a part of the decomposition process(column compatibility checking) which accounts for a significant portion of the overall program execution time. Results obtained from the

implemented algorithm show that it does speed up column compatibility checking in the decomposition process and by a significant margin. Hence, this new algorithm allows the program to obtain solutions to decompositions more quickly. This algorithm is presented in Chapter 3. The other main algorithm presented in this thesis is a column based input/output encoding approach. This algorithm was not implemented because of time spent on other parts of the decomposition program(MULTIS/GUD) and on additional research. However, examples done by hand indicate significant potential in simplifying sub-functions in the decomposition process by assigning multiple codes to columns. This algorithm is presented in Chapter 5.

CHAPTER 3

COLUMN COMPATIBILITY CHECKING IN CURTIS-STYLE DECOMPOSITIONS

3.1 Introduction

Column compatibility checking is the process of constructing a compatibility or incompatibility graph to be used in a Curtis-style decomposition. The nodes in the graph represent columns(or groups of columns) and edges represent the compatibility relationship between the columns. The graph, once constructed, is used in the next major step in the decomposition process(column minimization).

This chapter presents a new approach which can significantly decrease the time required for column compatibility checking over classical approaches in Curtis-style decompositions in cases where large graphs are constructed. Reducing execution time is the primary contribution of the new approach presented in this chapter(i.e., the same data is obtained, but much faster). Large graphs may be created for certain technologies such as in PLA partitioning for CPLDs or FPGAs. Large graphs may be used in other applications as well. Few applications for Curtis-style decompositions absolutely require large graphs to be constructed. However, some of the highest quality decompositions may require large graphs to be constructed. Typically, the greater the number of variables in the bound set, the more nodes

there will be in a compatibility or incompatibility graph. Unfortunately, little is published about the use of large bound sets in Curtis-style decompositions. This is most likely due to the increased computation time required for partitioning, column compatibility checking, column minimization and encoding when bound sets are large.

In addition to saving time when the bound sets are large, this new approach can be used to search a significant part of the search space on large functions previously not feasible using previous approaches due to the large computational requirements. Yet another advantage of the new approach is that it can be integrated into a modified graph coloring algorithm to speed up column minimization as well.

Previously, column compatibility checking was done in a pairwise fashion referred to here as the Pair Compatibility Approach(**PCA**). Classical examples of this type of approach are found in papers by Luba[30][31]. The basic idea behind the **PCA** approach is to check the compatibility relationship of each column with every other column(one pair at a time).

The new approach presented in this chapter is referred to as the Group Compatibility Approach(**GCA**). The basic idea behind the **GCA** approach is to check the compatibility relationship between pairs of groups of columns instead of checking the compatibility between each pair of columns(one pair at a time).

Note that columns and the blocks of the bound set are usually referred to in this

chapter simply as columns. Similarly, rows and blocks of the free set are usually referred to in this chapter simply as the rows. This is done for two main reasons. One reason is that blocks of the bound set and blocks of the free set are treated the same way in the algorithms as with rows and columns. The second reason is to provide an aid to the graphical explanation of the approaches(i.e., it is much simpler to visualize rows and columns in a Karnaugh map than it is to visualize blocks of the free set and blocks of the bound set.

The format of this chapter is as follows: In Section 3.3, the algorithm for the classical approach(**PCA**) to the column compatibility checking problem is introduced. In Section 3.4, the algorithms for the new approach(**GCA**) to the column compatibility checking problem are introduced. Also presented in Sections 3.3 and 3.4, are two examples illustrating the differences in the **PCA** approach and the **GCA** approach. In Section 3.5, an analysis is presented which compares the **PCA** approach vs. the **GCA** approach. In Section 3.6 a comparisons of results are given. Finally, in Section 3.7 concluding remarks are presented.

3.2 Definitions, Notations, and Terminology

The following are relevant definitions, terminology, and notations used in the algorithms presented in this chapter. Certain definitions and notations may require additional explanation in order to fully understand what they represent and what they are used for. Additional explanations are presented in relevant sections

in this chapter.

Definition 3.2.1 Classes of cubes *are groups of cubes that have compatible output values.*

Definition 3.2.2 Classes of columns *are groups of columns which have compatible outputs within some subset of rows of the Karnaugh map. However, unless otherwise specified, this does not imply that the all columns in a class are compatible with each other within all rows of the Karnaugh map.*

Definition 3.2.3 Incompatible classes *are classes which are incompatible with some other class.*

Definition 3.2.4 Incompatible classes of cubes *are classes in which some or all of the cubes in one class are incompatible with some or all of the cubes in some other class.*

Definition 3.2.5 Incompatible classes of columns *are classes in which all of the columns in one class are incompatible with all of the columns in some other class.*

Definition 3.2.6 Pairs of incompatible classes of cubes *are two specific classes of cubes in which some or all of the cubes in one class are incompatible with some*

or all of the cubes in the other class. Unless otherwise specified, the pairs of incompatible classes of cubes are rough partitions of cubes which are elements of the same row but not of the same output class.

Definition 3.2.7 Pairs of incompatible classes of columns *are two specific classes of columns in which all of the columns in one class are incompatible with all of the columns in the other class.*

Definition 3.2.8 Output classes *are the individual partition blocks within the output partition. The cubes which belong to each output class have compatible output values.*

$\mathbf{A_i}$ = i_{th} partition block in $P(A)$. In simpler terms, A_i is a set of cubes corresponding to a row(or rows) in the Karnaugh map.

$\mathbf{B_i}$ = i_{th} partition block in $P(B)$. In simpler terms, B_i is a set of cubes corresponding to a column(or columns) in the Karnaugh map.

$\mathbf{F_i}$ = i_{th} partition block in $P(F)$. In simpler terms, F_i is a set of cubes which have compatible output values.

$\mathbf{B_{ij}}$ denotes an *edge* between columns B_i and B_j when referring to a compatibility graph or an incompatibility graph.

$\mathbf{AB_{ijk}} = AB_{ijk} = (B_i \cup B_j) \cap A_k =$ The set of all cubes that are elements of block B_i or B_j that are also elements of the same block of the free set A_k.

$\mathbf{IC_{ij}}$ denotes the class of cubes from row A_i that are elements of the same output

class F_j. Cubes are elements of the same output class F_j if they have compatible output values. For single output binary functions, one of the classes will have output value 0 while the other will have output value 1.

IC is the set of pairs of incompatible classes of cubes of the form (IC_{ij}, IC_{ik}). All classes IC_{ij} are incompatible with all classes IC_{ik} for all pairs in row i, and for all rows i.

$\mathbf{IR_{ir}}$ is the class of cubes that are incompatible with repeated cube r in row i.

IR is the set of pairs of incompatible classes of cubes of the form $(r,\ IR_{ir})$.

IB is the set of pairs of incompatible classes of columns of the form (IB_{ij}, IB_{ik}). All classes IB_{ij} are incompatible with all classes IB_{ik} for all pairs.

$\mathbf{IB_{ij}}$ denotes the set of columns(B_n) which have at least one cube in common with the class of cubes IC_{ij}. Expressed in simpler terms, IB_{ij} denotes a set of columns which have the same output values for a particular cofactor(or row) of the Karnaugh map.

Definition 3.2.9 Repeated cubes, *as defined here, are care cubes which are elements of more than one output class.*

Example:

```
1  0111 00
2  0001 01
3  0110 0-
4  1010 10
5  0010 11
```

There are four output classes corresponding to output vectors 00, 01, 10, and 11. From these output vectors, cubes may be classified as follows:

P(F) = {(1,3);(2,3);(4);(5)}.

Since cube number 3 is an element of more than one output class, it is referred to as a repeated cube.

$\mathbf{SR_i}$ denotes the set of repeated cubes that are found in row i of the Karnaugh map.

$\mathbf{R_{ir}}$ is the set of cubes which belong to at least one class IC_{ij} which cube r also belongs to. (R_{ir} represents the set of all "care" cubes which are compatible with cube r in row i).

3.3 Classical Approach To Column Compatibility Checking: PCA

3.3.1 Application To Single Output Functions

The following is a brief explanation of the **PCA** approach to column compatibility checking. The goal of this algorithm is to obtain either an incompatibility graph or a compatibility graph. Recall that a compatibility graph is simply the complement of the incompatibility graph. Figure 3.1a shows the Karnaugh map used to illustrate the **PCA** algorithm. In general, this algorithm checks the compatibility of each pair of columns and if compatible, then assigns an edge in the compatibility graph between the two nodes corresponding to the two columns. The method used for checking the compatibility of two columns is straight forward and

requires little explanation.

To determine if two columns are compatible, the output values of the columns are checked to see if they are compatible for each and every combination of the free set variables(i.e., each row). The order that each pair of columns are checked for compatibility is arbitrary. Begin by arbitrarily selecting the two columns highlighted in Figure 3.1a (columns B_1 and B_6). Figure 3.1b shows the output partition $P(F)$ with cubes classified according to their output values. In Figure 3.1c-f are shown the compatibility checks within each row necessary to determine if the two columns are compatible. Columns B_1 and B_6 are compatible within rows 1 thru 3, but are incompatible in row 4. Therefore, column B_1 is incompatible with column B_6. The remaining pairs of columns shown in Figure 3.2a are checked in the same manner. This results in the compatibility graph shown in Figure 3.2b with edges between nodes in the graph indicating that two columns are compatible.

3.3.1.1 Algorithm PCA For Single Output Functions

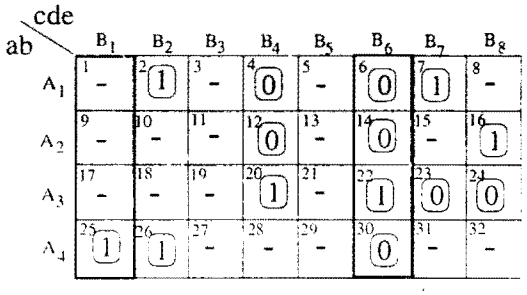
This algorithm is based on the pair compatibility approach by Luba[30].

Algorithm parameters defined:

a = Number of blocks in the free partition P(A).

b = Number of blocks in the bound partition P(B).

f = Number of blocks in the output partition P(F).

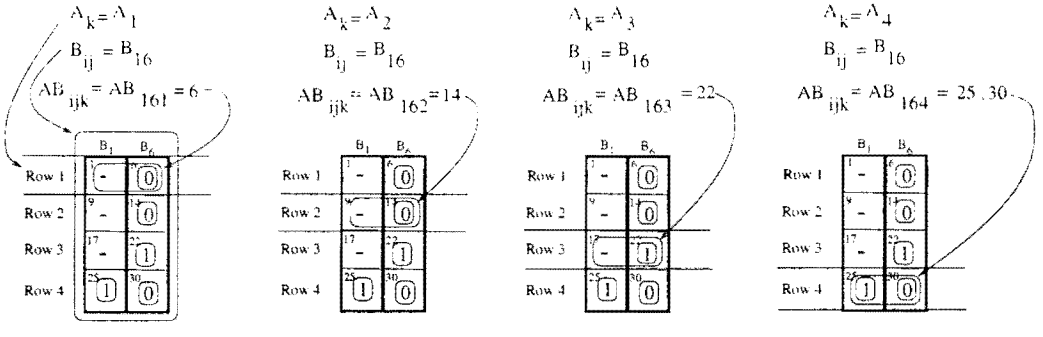


a) Karnaugh Map showing two columns selected to illustrate Pair Compatibility Checking. Columns are incompatible if, within any row, either [1 1] or [1 0] is found. Otherwise they are compatible.

$P(F) = \{F_1 : F_2\}$

$= \{4,6,12,14,23,24,30 : 2,7,16,20,22,25,26\}$

b) All cubes with the same output values are located in the same partition block. For two columns to be incompatible implies that not all cubes in AB_{ijk} are contained in one of the output partition blocks.



c) Check if B_1 & B_6 are compatible in row 1.

d) Similarly, we check row 2.

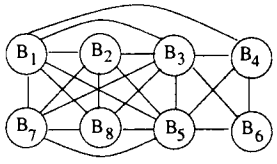
e) Similarly, we check row 3.

f) Checking row 4 we find $B_1 \not\sim B_6$.

Figure 3.1: Diagram illustrating **PCA** approach to Column Compatibility Checking for Single Output Functions: # 1

B_1 B_2	B_1 B_3	B_1 B_4	B_1 B_5	B_1 B_6	B_1 B_7	B_1 B_8
B_2 B_3	B_2 B_4	B_2 B_5	B_2 B_6	B_2 B_7	B_2 B_8	
B_3 B_4	B_3 B_5	B_3 B_6	B_3 B_7	B_3 B_8		
B_4 B_5	B_4 B_6	B_4 B_7	B_4 B_8			
B_5 B_6	B_5 B_7	B_5 B_8				
B_6 B_7	B_6 B_8					
B_7 B_8						

a) Build the compatibility graph by adding edges between each pair of nodes(columns) that are compatible. Listed above are the pairs of columns to be checked. Shown circled are the pairs of columns that are incompatible.



b) Compatibility Graph

Figure 3.2: Diagram illustrating **PCA** approach to Column Compatibility Checking for Single Output Functions: # 2

Algorithm 3.3.1

```
Begin

// For all pairs of blocks(columns) i and j, check if they are compatible blocks.

for (i=1; i<b ; i++)

   {

   for (j=(i+1); j≤b ; j++)

      {

      Return_Value = CHECK_PAIR_COMPATIBLE(i, j);

      if (Return_Value = True)

         Record block i as compatible with block j.

      else

         Record block i as incompatible with block j.
```

```
        }
    }
end.

Function CHECK_PAIR_COMPATIBILITY(i, j);

CHECK_PAIR_COMPATIBILITY(i, j);
    {
    // Combine the cubes contained in two blocks(or columns) to be checked
    // for compatibility...
    B_ij = B_i ∪ B_j;
    for (k=1; k≤a ; k++)
        {
        // For each row k do the following:
        // Initialize variable "Compatible" for each new row to check.
        Compatible = False;
        // Find the set of cubes within row k which are elements of
        // either block B_i or B_j.
        AB_ijk = B_ij ∩ A_k;
        // Check if all cubes in AB_ijk are contained in one of the output classes.
        // If all cubes in a set(or class) are elements of the same output class,
```

```
    // then they are mutually compatible. If so, then columns i and j are
    // compatible in row k.
    for (l=1; l≤f ; l++)
        {
        if (AB_ijk ⊆ F_l)
            {
            // B_i and B_j have compatible outputs for row k.
            Compatible = True;
            Goto check_next_row;
            }
        }
    if (Compatible = False)
        Return False;
    check_next_row:
    }
Return True;
}
```

ab \ cde	000	001	010	011	111	110	101	100
00	1 -	2 1	3 -	4 0	5 -	6 0	7 1	8 -
01	9 -	10 -	11 -	12 0	13 -	14 0	15 -	16 1
11	17 -	18 -	19 -	20 1	21 -	22 1	23 0	24 0
10	25 1	26 1	27 -	28 -	29 -	30 0	31 -	32 -
Column reference numbers →	1	2	3	4	5	6	7	8

Figure 3.3: Karnaugh Map for function $F2$.

3.3.1.2 Illustration Of The PCA Approach On Function F2

Here function **F2** is decomposed using the **PCA** approach. The same function is again decomposed using the **GCA** approach in Section 3.4.1.2. In each case, the example decomposition problem is completed using the same column minimization method(set covering) in order to illustrate how the results of column compatibility checking are used in a Curtis-style decomposition. The set covering method is not covered in detail as it is not one of the central topics of this thesis.

Problem Description For Function F2

Given is the function described by the Karnaugh map in Figure 3.1 and repeated again in Figure 3.3, with the bound and free sets {c,d,e} and {a,b}, respectively. The following are the rough partitions for the bound set, free set, and output set

respectively. Commas separate minterm numbers(or cube numbers) within each rough partition and semicolons separate partition blocks. Don't cares are not enumerated in the partitions(i.e., they are not used in the partition operations) and $\emptyset$(empty set) indicates no specified values in a particular partition. Construct the compatibility graph and perform column minimization to obtain a cover set Π_G.

$$
\begin{aligned}
P(B) &= (25;\ 2,26;\ \emptyset;\ 4,12,20;\ \emptyset;\ 6,14,22,30;\ 7,23;\ 16,24) \\
&= (B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8); && (3.1) \\
P(A) &= (2,4,6,7;\ 12,14,16;\ 20,22,23,24;\ 25,26,30) \\
&= (A_1, A_2, A_3, A_4); && (3.2) \\
P(F) &= (4,6,12,14,23,24,30;\ 2,7,16,20,22,25,26) \\
&= (F_1, F_2); && (3.3) \\
& && (3.4)
\end{aligned}
$$

Decomposition Of Function F2

There are four steps shown in this example. Step one illustrates the **PCA** approach. The remaining steps illustrate the column minimization phase of the decomposition process. The steps for the decomposition are as follows:

Step 1: This step illustrates the **PCA** approach.

Execution of PCA Algorithm.

Find all compatible pairs of columns which represent the compatibility graph. Compatible pairs of columns are those which have the same output values in corresponding positions of every row. Any one of the following pairs of cells classifies as having the same values: (1,1),(1,X),(0,0),(0,X),(X,X). For simplicity, let B_{ij} denote the compatible pair of columns B_i and B_j, also denoted by $(B_i,\ B_j)$.

The condition for compatibility between each pair of columns can be expressed as follows:

If

$$\forall k \forall m \quad A_k \cap (B_i \cup B_j) \subseteq F_m \tag{3.5}$$

Then B_i and B_j are compatible, denoted as $B_i \sim B_j$. Using this condition for compatibility find the set of all pairs of compatible columns.

Begin by arbitrarily selecting two columns(B_1 and B_2) to check if they are compatible. Merging B_1 and B_2 together, produces:

$$B_{12} = (B_1 \cup B_2) = ((25) \cup (2,26)) = (2,25,26); \tag{3.6}$$

Now the check of the condition for compatibility is performed:

Row 1) $A_1 \cap (B_1 \cup B_2) \subseteq F_1$?

$(2,4,6,7) \cap (2,25,26) = (2) \subseteq (4,6,12,14,23,24,30)$?

Not satisfied!

$A_1 \cap (B_1 \cup B_2) \subseteq F_2$?

$(2,4,6,7) \cap (2,25,26) = (2) \subseteq (2,7,16,20,22,25,26)$?

Satisfied! Therefore B_1 is compatible with B_2 in row 1.

Row 2) $A_2 \cap (B_1 \cup B_2) \subseteq F_1$?

$(12,14,16) \cap (2,25,26) = (\emptyset) \subseteq (4,6,12,14,23,24,30)$?

Satisfied! Therefore B_1 is compatible with B_2 in row 2.

Row 3) $A_3 \cap (B_1 \cup B_2) \subseteq F_1$?

$(20,22,23,24) \cap (2,25,26) = (\emptyset) \subseteq (4,6,12,14,23,24,30)$?

Satisfied! Therefore B_1 is compatible with B_2 in row 3.

Row 4) $A_4 \cap (B_1 \cup B_2) \subseteq F_1$?

$(25,26,30) \cap (2,25,26) = (25,26) \subseteq (4,6,12,14,23,24,30)$?

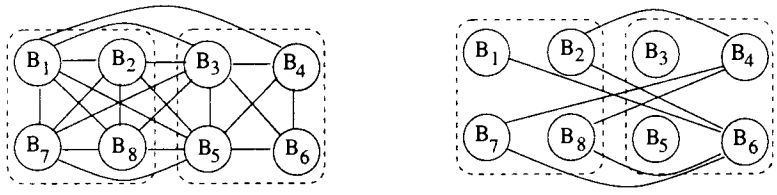
Not satisfied!

$A_4 \cap (B_1 \cup B_2) \subseteq F_2$?

$(25,26,30) \cap (2,25,26) = (25,26) \subseteq (2,7,16,20,22,25,26)$?

Satisfied! Therefore B_1 is compatible with B_2 in row 4.

Shown circled with dashed lines are the classes chosen for the final cover set.



a) Compatibility Graph

b) Incompatibility Graph

Figure 3.4: Compatibility and Incompatibility graphs for Function $F2$

Therefore since the compatibility relation above was satisfied for each of the four rows(A_i), then $B_1 \sim B_2$.

In the same way the compatibility relation is checked for all other pairs of blocks in $P(B)$.

This results in the set C_B of pairwise compatible blocks:

$$C_B = (B_{12}, B_{13}, B_{14}, B_{15}, B_{17}, B_{18}, B_{23},$$
$$B_{25}, B_{27}, B_{28}, B_{34}, B_{35}, B_{36}, B_{37}, B_{38}, B_{45}, B_{46}, B_{56}, B_{57}, B_{58}, B_{78}) \quad (3.7)$$

Based on set C_B, the compatibility graph from Figure 3.4 is created. It has nodes corresponding to the columns in the Karnaugh map and the edges between them indicating that the columns are compatible(edges correspond to elements of C_B).

The next stage of the decomposition process(column minimization) is not part of the algorithm for column compatibility checking. However, it is shown

here to illustrate how the results from column compatibility checking are used in the next stage of the decomposition process. This next stage is shown in steps 2 thru 4.

Step 2: Incrementally construct CCs from the set of compatible pairs of columns found in step 1 until each column is present in at least one CC. For each iteration, add the next column to be considered to each CC provided that the new column is compatible with all other columns in the class. If a new column is not compatible with any of the existing classes, then add a new class containing all previously assigned columns that are mutually compatible with each other and with the new column. For simplicity, the block index number is used to denote each block (i.e., block B_i is simply shown as index i).

$Iteration1 = (1)$

Next, combine column 2 with the CCs of the previous incremental step.

$Iteration2 = (1,2)$

Next, combine column 3 with the CCs of the previous incremental step.

$Iteration3 = (1,2,3)$

Next, combine column 4 with the CCs of the previous incremental step.

It is found that column 4 is not compatible with column 2 and therefore a new CC must be introduced which includes column 4 and not column 2.

$Iteration4$ = { (1,2,3), (1,3,4) }

Similarly, complete the CCs until all columns are included in at least one CC.

$Iteration5$ = { (1,2,3,5), (1,3,4,5) }

$Iteration6$ = { (1,2,3,5), (1,3,4,5), (3,4,5,6) }

$Iteration7$ = { (1,2,3,5,7), (1,3,4,5), (3,4,5,6) }

$Iteration8$ = { (1,2,3,5,7,8), (1,3,4,5), (3,4,5,6) }

Step 3: Find the minimum number of CCs which will completely cover all of the columns. The CCs in the last row of the previous step ($Iteration8$) form the CCs for this function on the given bound set.

$$CC1 = (1,2,3,5,7,8) = (B_1, B_2, B_3, B_5, B_7, B_8)$$

$$CC2 = (1,3,4,5) = (B_1, B_3, B_4, B_5)$$

$$CC3 = (3,4,5,6) = (B_3, B_4, B_5, B_6)$$

From these CCs, the minimal cover set Π_G can be formed:

$$\Pi_G = ((B_1, B_2, B_3, B_5, B_7, B_8); \quad (B_3, B_4, B_5, B_6)) \tag{3.8}$$

Step 4: This step in the decomposition process is optional. Whether or not this step should be executed depends on the type of encoding method used(i.e., some

encoding methods require disjoint cover sets). However, this step is executed here to illustrate the difference between disjoint and non-disjoint cover sets. To form the disjoint cover set Π_G, remove redundant blocks from all CCs of the minimal cover found in the previous step (i.e., $((B_1, B_2, B_7, B_8); (B_3, B_4, B_5, B_6))$. This results in the disjoint cover set Π_G:

where $\Pi_G = ((B_1, B_2, B_7, B_8); \ (B_3, B_4, B_5, B_6))$ This completes the illustration of this example.

3.3.2 Application To Multiple Output Functions

3.3.2.1 Algorithm PCA For Multiple Output Functions

Unlike the GCA approach which has a separate algorithm for single output functions and multiple output functions, the PCA approach has one algorithm for both single output functions and multiple output functions. This algorithm was presented in Section 3.3.1.1 and therefore will not be repeated here.

3.3.2.2 Illustration Of The PCA Approach On Function F3

Problem Description For Function F3

Table 3.1 describes the next example, function $F3$. The first column is the enumeration of cubes. The input variables are denoted X_1 thru X_4 and the output variables are Y_1 and Y_2. The multiple valued map corresponding to Table 3.1

Cube	X_1	X_2	X_3	X_4	Y_1	Y_2
1	0	0	2	0	1	1
2	3	0	-	1	0	0
3	3	1	0	-	-	0
4	2	1	3	0	0	1
5	-	1	1	1	1	-
6	1	0	3	0	0	-
7	2	-	3	1	0	1
8	3	1	1	0	1	0
9	1	1	-	1	-	0
10	3	0	2	0	1	-
11	1	1	3	1	0	0

Table 3.1: Table for Example Function $F3$

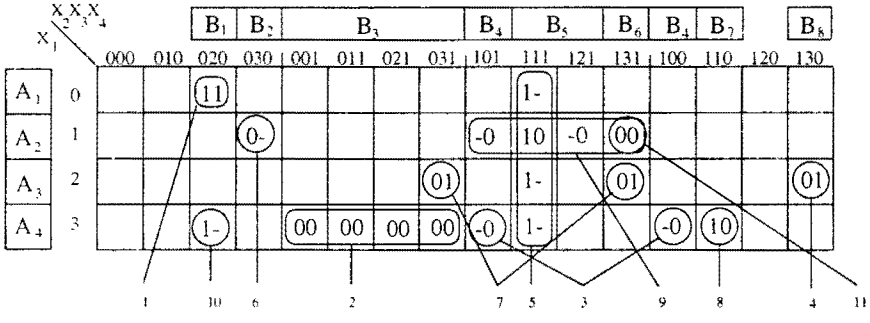


Figure 3.5: Multi-valued Map for Function $F3$

is shown in Figure 3.5. Find the decomposition $H(A, G(B))$ given the specified bound and free sets. For this example, variable X_1 was chosen for the free set and variables X_2, X_3, and X_4 were chosen for the bound set variables. The following are the corresponding bound, free, and output partitions.

$$P(A) = P(X_1) = (1,5;\ 5,6,9,11;\ 4,5,7;\ 2,3,5,8,10)$$

$$= (A_1, ..., A_4)$$

$$P(B) = P(X_2) \cdot P(X_3) \cdot P(X_4) = (1,10;\ 6;\ 2,7;\ 3,9;\ 5,9;\ 7,9,11;\ 8;\ 4)$$

$$= (B_1, ..., B_8)$$

$$P(F) = (1,5,10;\ 2,3,6,9,11;\ 3,5,8,9,10;\ 4,6,7)$$

$$= (F_1, ..., F_4)$$

Thus:

$A_1 = (1,5)$; $A_2 = (5,6,9,11)$; $A_3 = (4,5,7)$; $A_4 = (2,3,5,8,10)$

$B_1 = (1,10)$; $B_2 = (6)$; $B_3 = (2,7)$; $B_4 = (3,9)$;

$B_5 = (5,9)$; $B_6 = (7,9,11)$; $B_7 = (8)$; $B_8 = (4)$

$F_1 = (1,5,10)$; $F_2 = (2,3,6,9,11)$; $F_3 = (3,5,8,9,10)$; $F_4 = (4,6,7)$

Decomposition Of Function F3

Step 1: This step illustrates the use of the **PCA** approach to column compatibility checking.

Execution of PCA Algorithm.

Find all compatible pairs of columns. Begin by arbitrarily selecting two

columns(B_3 and B_5) to check if they are compatible. Merging B_3 and B_5 together, the following set of cubes are produced:

$$B_3 \cup B_5 = B_{35} = (2, 5, 7, 9); \tag{3.9}$$

Now the check of the condition for compatibility is performed:

Row 1) $A_1 \cap (B_3 \cup B_5) \subseteq F_1$?

$(1, 5) \cap (2, 5, 7, 9) = (5) \subseteq (1, 5, 10)$?

Satisfied! Therefore B_3 is compatible with B_5 in row 1.

Row 2) $A_2 \cap (B_3 \cup B_5) \subseteq F_1$?

$(5, 6, 9, 11) \cap (2, 5, 7, 9) = (5, 9) \subseteq (1, 5, 10)$?

Not satisfied!

$\mathrm{A}_2 \cap (B_3 \cup B_5) \subseteq F_2$?

$(5, 6, 9, 11) \cap (2, 5, 7, 9) = (5, 9) \subseteq (2, 3, 6, 9, 11)$?

Not satisfied!

$\mathrm{A}_2 \cap (B_3 \cup B_5) \subseteq F_3$?

$(5, 6, 9, 11) \cap (2, 5, 7, 9) = (5, 9) \subseteq (3, 5, 8, 9, 10)$?

Satisfied! Therefore B_3 is compatible with B_5 in row 2.

Row 3) $A_3 \cap (B_3 \cup B_5) \subseteq F_1$?

$(4,5,7) \cap (2,5,7,9) = (5,7) \subseteq (1,5,10)$?

Not satisfied!

$A_3 \cap (B_3 \cup B_5) \subseteq F_2$?

$(4,5,7) \cap (2,5,7,9) = (5,7) \subseteq (2,3,6,9,11)$?

Not satisfied!

$A_3 \cap (B_3 \cup B_5) \subseteq F_3$?

$(4,5,7) \cap (2,5,7,9) = (5,7) \subseteq (3,5,8,9,10)$?

Not satisfied!

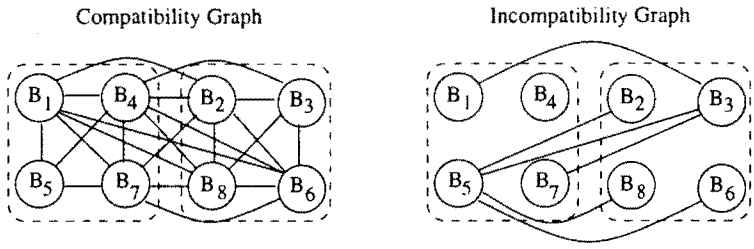
$A_3 \cap (B_3 \cup B_5) \subseteq F_4$?

$(4,5,7) \cap (2,5,7,9) = (5,7) \subseteq (4,6,7)$?

Not satisfied!

Since the condition for compatibility was not satisfied for row $3(A_3)$, it is known that columns(blocks) 3 and 5 are not compatible. Checking for compatibility of row $4(A_4)$ is not necessary, since it has already been determined that these two blocks are incompatible.

Therefore since the compatibility expression above was not satisfied for all rows, then $B_4 \not\sim B_5$.



Shown circled with dashed lines are the classes chosen for the final cover set.

Figure 3.6: Compatibility and Incompatibility graphs for Function $F3$

In the same way check the compatibility relation on all other pairs of blocks in $P(B)$. This results in the set of pairwise compatible blocks C_B:

$$C_B = (B_{12}, B_{14}, B_{15}, B_{16}, B_{17}, B_{18}, B_{23}, B_{24}, B_{26}, B_{27}, B_{28}, B_{34}, B_{36}, B_{38}, B_{45}, B_{46}, B_{47}, B_{48}, B_{57}, B_{67}, B_{68}, B_{78}) \quad (3.10)$$

This set of compatible pairs forms the compatibility graph from Figure 3.6 with the nodes corresponding to the blocks(of the bound set) in the Karnaugh map and the edges between them indicating that the blocks are compatible. Also shown in this figure is the incompatibility graph which is the complement of the compatibility graph.

The next stage of the decomposition process(column minimization) is not part of the algorithm for column compatibility checking. However, it is shown here to illustrate how the results from column compatibility checking are used in the next stage of the decomposition process. This next stage is shown in

steps 2 thru 4.

Step 2: Find the largest CCs from the set of compatible pairs found in step 1. For simplicity, only the block numbers are used(i.e block B_1 is denoted by index 1).

$Iteration1 = (1)$

Next, combine column 2 with the CCs of the previous incremental step.

$Iteration2 = (1,2)$

Next, combine column 3 with the CCs of the previous incremental step.

It is found that column 3 is not compatible with column 1 and therefore a new CC must be introduced which includes column 3 and not column 1.

$Iteration3 = (1,2),(2,3)$

Next, combine column 4 with the CCs of the previous incremental step.

$Iteration4 = (1,2,4),(2,3,4)$ Similarly, complete the CCs until all columns are included in at least one CC.

$Iteration5 = (1,2,4),(1,4,5),(2,3,4)$

$Iteration6 = (1,2,4,6),(1,4,5),(2,3,4,6)$

$Iteration7 = (1,2,4,6,7),(1,4,5,7),(2,3,4,6)$

$Iteration8 = (1,2,4,6,7,8),(1,4,5,7),(2,3,4,6,8)$

Step 3: Find the set of CCs from the last iteration which together will cover P(B) completely. A set of CCs is said to cover P(B) completely if each block in P(B) is an element of at least one CC. The CCs from *Iteration*8 form the CCs for this function on the given bound set.

$$CC1 = (B_1, B_2, B_4, B_6, B_7, B_8)$$

$$CC2 = (B_1, B_4, B_5, B_7)$$

$$CC3 = (B_2, B_3, B_4, B_6, B_8)$$

Therefore the one of the possible minimal covers is:

$$((B_1, B_4, B_5, B_7); \quad (B_2, B_3, B_4, B_6, B_8))$$

Step 4: This step is optional. To form Π_G, remove redundant blocks from all CCs of the minimal cover $((B_1, B_4, B_5, B_7); \quad (B_2, B_3, B_6, B_8))$ found in the previous step. This results in the partition Π_G which satisfies the requirement $P(A) \cdot \Pi_G \subseteq P(F)$.

$$\Pi_G = (1,4,5,7;\ 2,3,6,8)$$

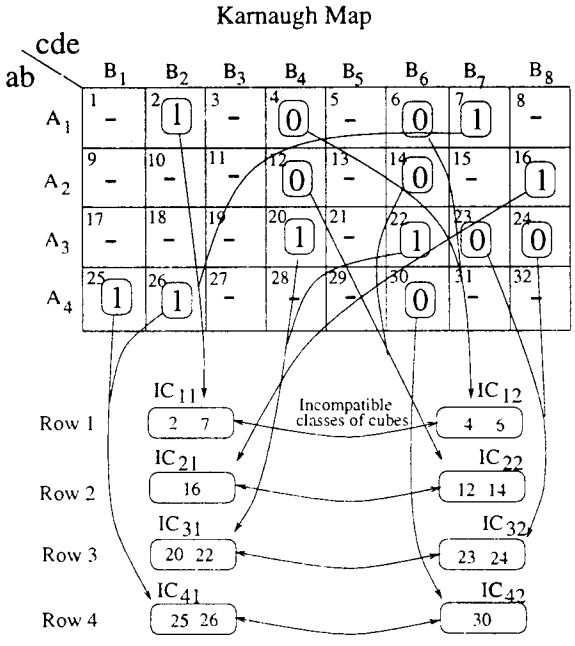
Since only two classes were required to form the cover set, then the resulting function is $F = H(x_1, G(x_2, x_3, x_4))$ where G is a single output function. This completes the illustration of this example.

3.4 New Approach To Column Compatibility Checking: GCA

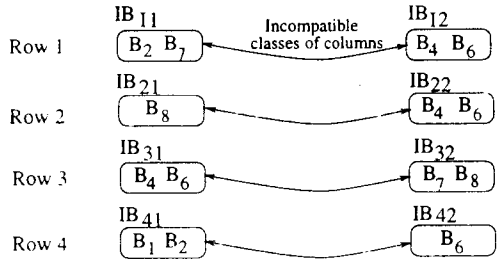
The new algorithms presented in the following sections can greatly reduce the number of calculations required to create the compatibility or incompatibility graph when there is a large number of blocks in the bound set.

The basic idea of this algorithm is to find pairs of incompatible classes of minterms for each row and then replace these incompatible classes of minterms with incompatible classes of columns(blocks of $P(B)$) which contain those minterms. These incompatible classes of columns are used to form the set of pairs of incompatible columns. Each pair of incompatible columns are represented by an edge in the incompatibility graph.

There are two **GCA** algorithms, one for single output functions (Algorithm 3.4.1) and one for multiple output functions (Algorithm 3.4.2). These algorithms share many of the same steps. However, unlike Algorithm 3.4.2, Algorithm 3.4.1 is not sufficient to handle multiple output functions(i.e. some columns may be incorrectly classified as compatible or incompatible). For this reason, Algorithm 3.4.2 has additional steps so that all columns are correctly classified for multiple output functions. An explanation of this difference is deferred until after the reader has been introduced to the algorithm for single output functions. A detailed explanation is given in Section 3.4.2 as to why the extra steps are necessary.

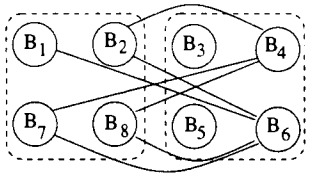


a) Karnaugh map showing cubes in each row classified according to their output values.

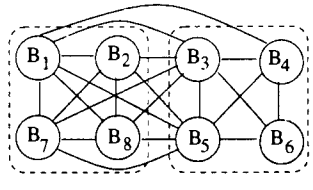


b) For each class of cubes, form a class of columns by including all columns which contain at least one cube from the class of cubes. Note that each column from one class(within each row) is incompatible with every column in the other class. For example: For row 1, B_2 is incompatible with B_4 & B_6. Also, B_7 is incompatible with B_4 & B_6.

Shown circled with dashed lines are the classes chosen for the final cover set.



c) Incompatibility Graph



d) Compatibility Graph

Figure 3.7: Diagram illustrating **GCA** approach to Column Compatibility Checking for Single Output Functions

3.4.1 Application To Single Output Functions

The following is a brief explanation of the **GCA** approach to column compatibility checking for single output functions:

The desired result of this algorithm is either an incompatibility graph or a compatibility graph. Figure 3.7a shows the result of the first step in the algorithm. Here the cubes which are elements of each row are separated into classes based on the output value of the cubes. For single output functions, there are two output classes(0 and 1). For row 1, find two cubes which are elements of each output class(i.e., cubes 2 and 7 both have output value 1 and cubes 4 and 6 both have output value 0). Similarly, separate cubes in other rows into classes based on their output values. From these classes of cubes within each row, the observation can be made that columns which contain cubes in one class are incompatible with the columns that contain cubes in the opposite class.

Therefore, with this observation in mind, perform the next step in the algorithm by forming a class of columns for each class of cubes. This is accomplished by including all columns which have at least one cube in common with a specific class of cubes to a new class of columns. The result of this step is shown in Figure 3.7b. As shown in Figure 3.7b, for classes within each row, all columns in one class are mutually incompatible with all columns in the opposite class. From these pairs of incompatible classes of columns, construct either an incompatibility graph or a

compatibility graph.

For simplicity, first construct the incompatibility graph by adding an edge between nodes(columns) in the graph which are incompatible. This is done for each row as follows: for each column in one class, add an edge between the node that corresponds to that column, to every node corresponding to the columns in the other class. This results in the incompatibility graph shown in Figure 3.7c). The complement of the incompatibility graph is shown in Figure 3.7d. That concludes the general description of the GCA approach for single output functions.

3.4.1.1 Algorithm GCA For Single Output Functions

Algorithm parameters defined:

GRAPH = Set of pairs of incompatible columns representing the incompatibility graph or alternatively, the set of pairs of compatible columns representing the compatibility graph.

i = number of rows(or blocks in the free partition).

j = number of output classes(or blocks in the output partition).

n = number of columns(or blocks in the bound partition).

Algorithm 3.4.1

Begin

a) $IC = FORM\_SET\_OF\_PAIRS\_IC(A, F, i, j);$

// This function forms pairs of classes of cubes for each row in the Karnaugh

// map. Cubes in each class are elements of the same row and output class.

// Within a pair of classes, all cubes in one class are incompatible with all

// cubes in the other class(with the exception of repeated cubes).

e) $IB = FORM\_SET\_OF\_PAIRS\_IB(IC, B, i, n);$

// This function forms pairs of classes of columns such that, within a pair of

// classes, all columns in one class are incompatible with all columns in the

// other class.

f) $GRAPH = FORM\_GRAPH\_FROM\_IB(IB, i, B, n);$

// This function forms the incompatibility(or compatibility) graph.

end.

Detailed explanation of functions used in Algorithm 3.4.1.

Note that only three out of six function calls in the multiple output Algorithm 3.4.2 are executed by the single output algorithm. Therefore, the parts of this algorithm which differ from the multiple output algorithm, are indicated in the corresponding parts below.

a) Part a of **GCA** Algorithms 3.4.1 and 3.4.2 basically involves obtaining the

quotient of partitions($P(A)|P(F)$) as previously done by Luba[31]. However, the quotient of partitions is used here for column compatibility checking, whereas Luba used quotient partitions to partition variables to free and bound sets.

Function $FORM\_SET\_OF\_PAIRS\_IC(A, F, i, j)$;

This function forms the set of pairs of incompatible classes of cubes IC. The pairs of incompatible classes of cubes are of the form $(IC_{ij}, IC_{ik}) \in IC$. To find each pair (IC_{ij}, IC_{ik}), classify cubes which are elements of row A_i according to the output classes they belong to.

$$\forall i \forall j : IC_{ij} \;=\; A_i \cap F_j$$

The classes IC_{ij} and IC_{ik} obtained from each row A_i are incompatible with one another and therefore constitute the set of pairs of incompatible classes of cubes referred to as IC above.

Function $FORM\_SET\_OF\_PAIRS\_IC(A, F, i, j)$ returns set IC.

b) Part b in the multiple output algorithm is not executed in the single output

algorithm.

c) Part c in the multiple output algorithm is not executed in the single output algorithm.

d) Part d in the multiple output algorithm is not executed in the single output algorithm.

e) Function $FORM\_SET\_OF\_PAIRS\_IB(IC, B, IC\_LENGTH, n)$;

This function forms the set of pairs of incompatible classes of columns IB. The pairs of incompatible classes of columns are of the form $(IB_{ij}, IB_{ik}) \in IB$. To find each class IB_{ij}, find all columns which contain at least one cube that is contained in class IC_{ij}, and put them together in class IB_{ij}. This is done as follows:

$$IB_{ij} = \{n \mid \forall i \forall j \;\; IC_{ij} \cap B_n \neq \emptyset \}$$

where n is simply an index for blocks of the bound set.

The pairs of classes IB_{ij} and IB_{ik} obtained from each row i are incompatible with each other and therefore constitute the set of pairs of incompatible classes of columns referred to above as IB.

Function $FORM\_SET\_OF\_PAIRS\_IB(IC, B, IC\_LENGTH, n)$ returns the set IB and IB_LENGTH.

f) Function $FORM\_GRAPH\_FROM\_IB(IB, IC\_LENGTH, B, n)$;

This step simply changes the way the compatibility/incompatibility information is stored. This step is not required for column minimization algorithms that are able to work directly with the incompatible classes of columns formed in the previous step. However, if the column minimization algorithm requires a graph with nodes representing individual blocks of the bound set, then the data stored in the previous step is converted from pairs of incompatible classes of columns to pairs of columns which are incompatible. To convert to pairs of columns which are incompatible, perform the following:

For each row i, assign every column in class IB_{ij} as pairwise incompatible with every column in class IB_{ik}.

The above conversion can create the data which corresponds to both the compatibility and incompatibility graphs simultaneously. This can be done by simply using a two dimensional array with a bit set indicating that

a pair is compatible, and if it is not set, then the pair is incompatible. Function $FORM\_GRAPH\_FROM\_IB(IB, IC\_LENGTH, B, n)$ returns a pointer($GRAPH$) to the data representing the desired graph.

3.4.1.2 Illustration Of The GCA Approach On Function F2

Problem Description For Function F2

Repeated here for completeness is the following information: Given is the function described by the Karnaugh map in Figure 3.7 and repeated again in Figure 3.8, with the bound and free sets {c,d,e} and {a,b}, respectively. The following are the rough partitions for the bound set, free set, and output set respectively. Commas separate minterm numbers(or cube numbers) within each rough partition and semicolons separate partition blocks. Don't cares are not enumerated in the partitions(i.e., they are not used in the partition operations) and $\emptyset$(empty set) indicates no specified values in a particular partition. Construct the compatibility graph and perform column minimization to obtain a cover set Π_G.

$$\begin{aligned} P(B) &= (25;\ 2,26;\ \emptyset;\ 4,12,20;\ \emptyset;\ 6,14,22,30;\ 7,23;\ 16,24) \\ &= (B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8); \qquad (3.11) \\ P(A) &= (2,4,6,7;\ 12,14,16;\ 20,22,23,24;\ 25,26,30) \\ &= (A_1, A_2, A_3, A_4); \qquad (3.12) \\ P(F) &= (4,6,12,14,23,24,30;\ 2,7,16,20,22,25,26) \end{aligned}$$

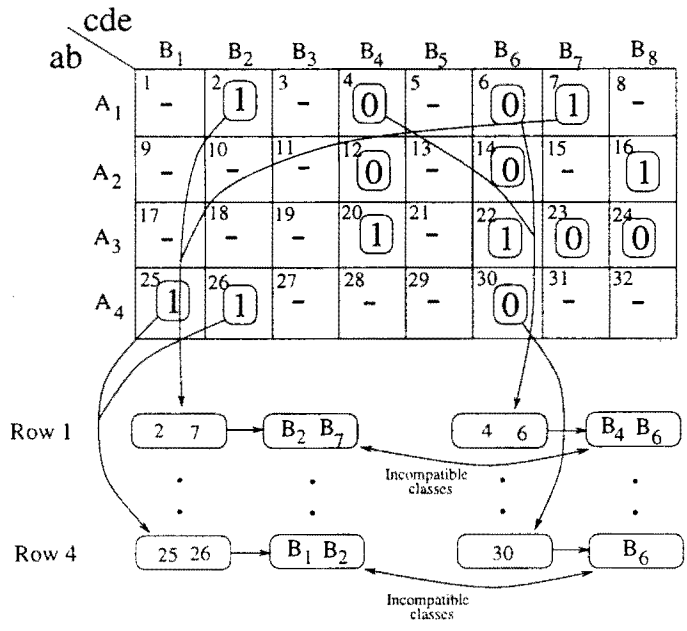


Figure 3.8: Karnaugh Map for function $F2$ showing incompatible classes of columns

$$= \quad (F_1, F_2); \tag{3.13}$$

Decomposition Of Function F2

Step 1: This step of the decomposition illustrates the **GCA** approach.

Execution of GCA Algorithm 3.4.1.

a) Generate set IC using function

$FORM\_SET\_OF\_PAIRS\_IC(A, F, i = 4, j = 2)$. This function performs the following calculations:

$$\forall i \forall j : IC_{ij} \;=\; A_i \cap F_j$$

The first class to be formed is IC_{11}. It is formed as follows:

$$IC_{11} = (A_1 \cap F_1) = ((2,4,6,7) \cap (4,6,12,14,23,24,30)) = (4,6).$$

Similarly, the remaining IC_{ij} are found. This results in the following classes of cubes.

$IC_{11} = (4,6)$, $IC_{12} = (2,7)$,

$IC_{21} = (12,14)$, $IC_{22} = (16)$,

$IC_{31} = (23,24)$, $IC_{32} = (20,22)$,

$IC_{41} = (30)$, $IC_{42} = (25,26)$,

The following is IC expressed as the set of pairs of incompatible classes of the form(IC_{i1}, IC_{i2}):

$IC =$

(((4,6), (2,7)),

((12,14), (16)),

((23,24), (20,22)),

((30), (25,26))).

b) Skip part b) since this is a single output function.

c) Skip part c) since this is a single output function.

d) Skip part d) since this is a single output function.

e) Generate IB using function

$FORM\_SET\_OF\_PAIRS\_IB(IC, B, i = 4, n = 8)$. IB is found accordingly:

$$IB_{ij} = \{n \mid \forall i \forall j \;\; IC_{ij} \cap B_n \neq \emptyset \}$$

For simplicity, only the column number is used to denote each column (i.e., B_i is shown as number i). The first class in IB to generate is IB_{11}. IB_{11} is found incrementally as follows(initially $IB_{11} = (\emptyset)$):

i) $IC_{11} \cap B_1 = (4, 6) \cap (25) = \emptyset$...

therefore IB_{11} remains unchanged,

ii) $IC_{11} \cap B_2 = (4, 6) \cap (2, 26) = \emptyset$...

therefore IB_{11} remains unchanged,

iii) $IC_{11} \cap B_3 = (4,6) \cap (\emptyset) = \emptyset$...

therefore IB_{11} remains unchanged,

iv) $IC_{11} \cap B_4 = (4,6) \cap (4,12,20) = (4)$...

therefore $IB_{11} = (4)$,

v) $IC_{11} \cap B_5 = (4,6) \cap (\emptyset) = \emptyset$...

therefore IB_{11} remains unchanged,

vi) $IC_{11} \cap B_6 = (4,6) \cap (6,14,22,30) = (6)$...

therefore $IB_{11} = (4,6)$,

vii) $IC_{11} \cap B_7 = (4,6) \cap (7,23) = \emptyset$...

therefore IB_{11} remains unchanged,

viii) $IC_{11} \cap B_8 = (4,6) \cap (16,24) = \emptyset$...

therefore the resulting class for $IB_{11} = (4,6)$.

Similarly, the remaining IB_{ij} classes are found. This results in the following classes of columns.

$IB_{11} = (4,6)$, $IB_{12} = (2,7)$,

$IB_{21} = (4,6)$, $IB_{22} = (8)$,

$IB_{31} = (7,8)$, $IB_{32} = (4,6)$,

$IB_{41} = (6)$, $IB_{42} = (1,2)$,

Therefore $IB =$

(((4,6), (2,7)),

((4,6), (8)),

((7,8), (4,6)),

((6), (1,2))).

Note: Each column in a pair of classes is incompatible with all columns in the opposite class. Each of these pairs represents the sets of columns which have a conflicting output in a particular row and therefore cannot be combined in the column minimization step.

f) Construct the desired graph using function $FORM\_GRAPH\_FROM\_IB(IB, i = 4, B, n = 8)$. This function performs the following calculations:

For each row index i, assign all columns in class IB_{i1} as pairwise incompatible with all columns in class IB_{i2}.

For row 1:

$(IB_{11} = (4,6)) \not\sim (IB_{12} = (2,7))$. Therefore columns 4 and 6 are incompatible with columns 2 and 7. This forms the incompatible pairs B_{24}, B_{26}, B_{47}, and B_{67}.

For row 2:

$(IB_{21} = (4,6)) \not\sim (IB_{22} = (8))$. Therefore columns 4 and 6 are incompatible with column 8. This forms the incompatible pairs B_{48} and B_{68}.

For row 3:

$(IB_{31} = (7,8)) \not\sim (IB_{32} = (4,6))$. Therefore columns 4 and 6 are incompatible with columns 7 and 8. This forms the incompatible pairs B_{47}, B_{48}, B_{67}, and B_{68}.

For row 4:

$(IB_{41} = (6)) \not\sim (IB_{42} = (1,2))$. Therefore column 6 is incompatible with columns 1 and 2. This forms the incompatible pairs B_{16} and B_{26}.

The sets of incompatible pairs for rows results in the set of pairwise incompatible columns I_B:

$$I_B = (B_{16}, B_{24}, B_{26}, B_{47}, B_{48}, B_{67}, B_{68})$$

The set IB of pairwise incompatible columns is used to form the incompatibility graph in Figure 3.7c. The set of pairwise compatible columns C_B can be obtained simply by removing set I_B from the set of all pairs of columns:

$$C_B = (B_{12}, B_{13}, B_{14}, B_{15}, B_{17}, B_{18}, B_{23}, B_{25}, B_{27}, B_{28}, B_{34}, B_{35}, B_{36}, B_{37}, B_{38}, B_{45}, B_{46}, B_{56}, B_{57}, B_{58}, B_{78})$$

This set of pairwise compatible columns forms the compatibility graph shown previously in Figure 3.7d. This completes the illustration of the GCA approach for this example.

Steps 2-4: The remaining steps(steps 2-4) in the decomposition of this function are carried out in the same exact way as was done in the previous example decomposition of Function $F2$ and are therefore not repeated here.

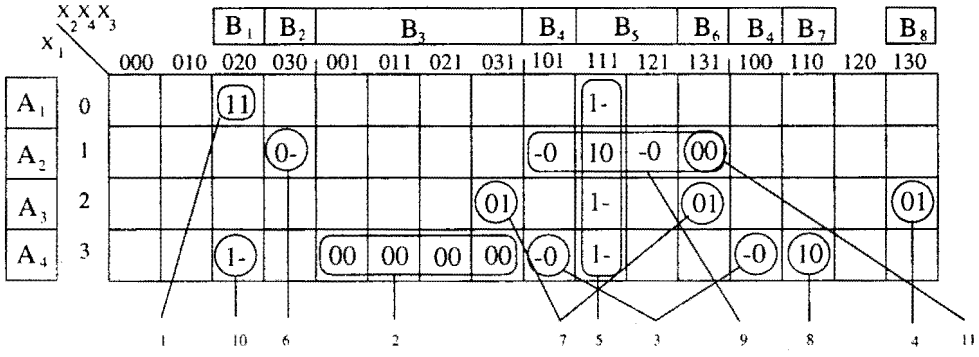
3.4.2 Application To Multiple Output Functions

The fundamental distinction between **GCA** algorithms for single output vs. multiple output functions is that cubes may be compatible with more than one output class(output partition block) in multiple output functions. These cubes are referred to as repeated cubes. In single output functions, disjoint subsets of cubes

are classified in each row according to output classes they belong to. When this is done, then pairs of classes of cubes are formed(IC_{ij}; IC_{ik}). In each of these pairs of classes, all cubes in one class are incompatible with all cubes in the other class. However, in multiple output functions, cubes classified in the same manner may result in pairs of classes of cubes which are not disjoint. When this occurs, then all cubes in one class are not incompatible with all cubes in the other class. To solve this problem, the set of pairs of classes of incompatible cubes are formed first for the set of repeated cubes only. This set is denoted IR. Then the repeated cubes are removed from all classes within set IC to form the new set IC of non-repeated cubes. The set IR is appended to IC and the remainder of the algorithm is identical to the single output algorithm. The following is a brief explanation of the GCA approach for multiple output functions:

Begin by classifying cubes within each row according to the output classes they are compatible with. The output classes are 00, 01, 10, and 11. The resulting classes formed are shown in Figure 3.9a.

From the classes IC_{ij}, the set of repeated cubes SR_i are found for each row i. In row 1, cube 5 is compatible with more than one class and is therefore classified as a repeated cube. Because there are no other repeated cubes in row 1, then SR_1 contains only one cube(cube 5). Similarly, the repeated cubes are found for the remaining rows. The resulting classes SR_i are shown in Figure 3.9b.



Output Classes →	11	00	10	01
	IC_{11}	IC_{12}	IC_{13}	IC_{14}
Row 1	1 5	Ø	5	Ø
	IC_{21}	IC_{22}	IC_{23}	IC_{24}
Row 2	5	6 9 11	5 9	6
	IC_{31}	IC_{32}	IC_{33}	IC_{34}
Row 3	5	Ø	5	4 7
	IC_{41}	IC_{42}	IC_{43}	IC_{44}
Row 4	5 10	2 3	3 5 8 10	Ø

a) Multi-valued map showing cubes in each row classified according to their output values.

b) Set of repeated cubes for each row. Repeated cubes are cubes that appear in more than one output class for a particular row.

SR_1 = (5)
SR_2 = (5,6,9)
SR_3 = (5)
SR_4 = (3,5,10)

Figure 3.9: Diagram illustrating **GCA** approach to Column Compatibility Checking for Multiple Output Functions: # 1

R_{ir}

row \ Repeated cube	3	5	6	9	10
1	-	R_{15}= {1}	-	-	-
2	-	R_{25}= {9}	R_{26}= {9,11}	R_{29}= {5,6,11}	-
3	-	R_{35}= {Ø}	-	-	-
4	R_{43}= {2,5,8,10}	R_{45}= {3,8,10}	-	-	R_{410}= {3,5,8}

a) Sets of cubes which are compatible with repeated cubes corresponding to each row. Dashes indicate that the repeated cube does not appear in that row of the Karnaugh map and therefore does not need to be considered for that row.

IR_{ir}

row \ Repeated cube	3	5	6	9	10
1	-	I_{15}= {Ø}	-	-	-
2	-	I_{25}= {6,11}	I_{26}= {5}	I_{29}= {Ø}	-
3	-	I_{35}= {4,7}	-	-	-
4	I_{43}= {Ø}	I_{45}= {2}	-	-	I_{410}= {2}

b) Sets of cubes which are incompatible with repeated cubes corresponding to each row. Dashes indicate that the repeated cube does not appear in that row of the Karnaugh map and therefore does not need to be considered for that row.

c) Set of pairs of incompatible classes of the form {r, I_{ir} }, obtained using the repeated cubes.

IR =

(5, I_{25}) = (5, {6,11}),
(5, I_{35}) = (5, {4,7}),
(5, I_{45}) = (5, {2}),
(6, I_{26}) = (6, {5}),
(10, I_{410}) = (10, {2}).

Figure 3.10: Diagram illustrating GCA approach to Column Compatibility Checking for Multiple Output Functions: # 2

Output Classes

	11	00	10	01
	IC_{11}	IC_{12}	IC_{13}	IC_{14}
Row 1	1	Ø	Ø	Ø
	IC_{21}	IC_{22}	IC_{23}	IC_{24}
Row 2	Ø	11	Ø	Ø
	IC_{31}	IC_{32}	IC_{33}	IC_{34}
Row 3	Ø	Ø	Ø	4 7
	IC_{41}	IC_{42}	IC_{43}	IC_{44}
Row 4	Ø	2	8	Ø

a) Set of classes IC_{ij} after repeated cubes have been removed.

IC = $(IC_{42};\ IC_{43}) = (\{2\}; \{8\})$.

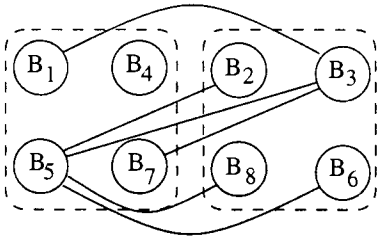
b) Set of pairs of incompatible classes of the form $\{I_{ij}, I_{ik}\}$, obtained using the non-repeated cubes. Pairs of incompatible classes which contain an empty set are not considered.

Figure 3.11: Diagram illustrating **GCA** approach to Column Compatibility Checking for Multiple Output Functions: # 3

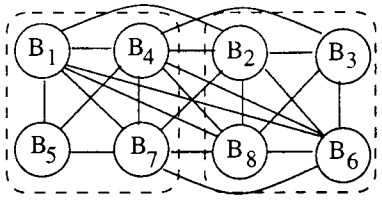
IC + IR =		IB =
$(IC_{42};\ IC_{43}) = (\{2\};\ \{8\})$,		$(IB_{11};\ IB_{12}) = (\{B_3\};\ \{B_7\})$,
$(5, I_{25}) = (5, \{6,11\})$,		$(IB_{21};\ IB_{22}) = (\{B_5\};\ \{B_2\ B_6\})$,
$(5, I_{35}) = (5, \{4,7\})$,	$\Longrightarrow$	$(IB_{31};\ IB_{32}) = (\{B_5\};\ \{B_3\ B_6\ B_8\})$,
$(5, I_{45}) = (5, \{2\})$,		$(IB_{41};\ IB_{42}) = (\{B_5\};\ \{B_3\})$,
$(6, I_{26}) = (6, \{5\})$,		$(IB_{51};\ IB_{52}) = (\{B_2\};\ \{B_5\})$,
$(10, I_{410}) = (10, \{2\})$.		$(IB_{61};\ IB_{62}) = (\{B_1\};\ \{B_3\})$.

a) IC + IR is the complete set of pairs of incompatible classes of cubes. IB is the set of pairs of incompatible classes of columns obtained from IC and IR.

Shown circled with dashed lines are the classes chosen for the final cover set.



b) Incompatibility Graph



c) Compatibility Graph

Figure 3.12: Diagram illustrating **GCA** approach to Column Compatibility Checking for Multiple Output Functions: # 4

Next, for each repeated cube r, determine the set of cubes which are compatible with that cube in each row that it is contained in. These sets are denoted R_{ir}. In row 1, there is only one repeated cube($SR_1 = 5$). Therefore, the set of cubes that are compatible with cube 5 in row 1 is $R_{ir} = R_{15} = 1$. The remaining compatible sets R_{ir} for each repeated cube are shown in Figure 3.10a.

Next, sets are formed from the cubes in each row that are incompatible with each repeated cube. This is done by simply adding all cubes to the incompatible set IR_{ir}, which are elements of row A_i, that are not elements of the compatible set R_{ir}. Results for all IR_{ir} are shown in Figure 3.10b.

From each repeated cube r and the corresponding class of cubes IR_{ir} that are incompatible with cube r, form a pair of incompatible classes of cubes of the form (r, IR_{ir}). The set of all such pairs is the set IR. The set IR is shown in Figure 3.10c.

Now that the pairs of incompatible classes of cubes are formed from the repeated cubes, the repeated cubes can be removed from all classes IC_{ij}, so that only the non-repeated cubes remain. The new set of classes IC_{ij} is shown in Figure 3.11a. From these classes, the set of pairs(IC_{ij}; IC_{ik}) $\in IC$ is formed. The set IC is shown in Figure 3.11b.

Next, the set IR is appended to set IC to form the complete set of pairs of incompatible classes of cubes. The complete set of pairs of incompatible classes of cubes($IC + IR$) is shown in Figure 3.12a.

From the complete set $IC + IR$, the set of pairs of incompatible classes of columns is formed(IB). This is done by adding all columns to a class such that each column in the class has at least one cube in common with the corresponding class of cubes in $IC + IR$. Set IB is shown in Figure 3.12a. Finally, create the incompatibility graph or the compatibility graph from the set IB.

Finally, to create the incompatibility graph, note that all columns within each class IB_{ij} are incompatible with all columns in IB_{ik}. For each pair of columns which are incompatible, an edge is added to the incompatibility graph. Shown in Figure 3.12b is the resulting incompatibility graph. Shown in Figure 3.12c is the corresponding compatibility graph.

3.4.2.1 Algorithm GCA For Multiple Output Functions

Additional algorithm parameters defined:

IC_LENGTH = number of pairs of classes in IC.

IR_LENGTH = number of pairs of classes in IR.

IB_LENGTH = number of pairs of classes in IB.

Algorithm 3.4.2

Note: Steps in this multiple output algorithm which are the same as the single output algorithm, Algorithm 3.4.1, are marked with an asterisk.

Begin

*a)** $IC = FORM\_SET\_OF\_PAIRS\_IC(A, F, i, j);$

// This function forms pairs of classes of cubes for each row in the Karnaugh

// map. Cubes in each class are elements of the same row and output class.

// Within a pair of classes, all cubes in one class are incompatible with all

// cubes in the other class(with the exception of repeated cubes).

b) $IR, IR\_LENGTH = FORM\_SET\_OF\_PAIRS\_IR(IC, A, i, j);$

// This function forms pairs of classes of cubes for each row in the Karnaugh

// map. In each pairs of classes produced, one element is a repeated cube and

// the other element in the pair is a set of cubes(repeated and/or non-repeated

// cubes) which are incompatible with the repeated cube element.

c) $IC, IC\_LENGTH = REMOVE\_REPEATED\_CUBES(IC, SR, i, j);$

// This function deletes the repeated cubes from the set IC produced in part a

// and returns the new set IC composed of non-repeated cubes.

d) $IC, IC\_LENGTH =$

APPEND_IR_TO_IC(IC, IR, IC_LENGTH, IR_LENGTH);

// This function appends two lists(or sets) together and returns the result.

*e)** $IB, IB\_LENGTH =$

FORM_SET_OF_PAIRS_IB(IC, B, IC_LENGTH, n);

// This function forms pairs of classes of columns such that, within a pair of

// classes, all columns in one class are incompatible with all columns in the

// other class.

f) GRAPH = FORM_GRAPH_FROM_IB(IB, IB_LENGTH, n);*

// This function forms the incompatibility(or compatibility) graph.

end.

Detailed explanation of functions used in Algorithm 3.4.2.

a) Function $FORM\_SET\_OF\_PAIRS\_IC(A, F, i, j)$;

Form the set of pairs of incompatible classes of cubes IC. The pairs of incompatible classes of cubes are of the form $(IC_{ij}, IC_{ik}) \in IC$. To find each pair (IC_{ij}, IC_{ik}), classify cubes which are elements of row A_i according to the output classes they belong to.

$$\forall i \forall j : IC_{ij} = A_i \cap F_j$$

The classes IC_{ij} and IC_{ik} obtained from each row A_i are incompatible with one another and therefore constitute the set of pairs of incompatible classes of cubes referred to as IC above.

The function returns set IC.

b) Function $FORM\_SET\_OF\_PAIRS\_IR(IC, A, i, j)$;

Find all pairs of incompatible classes of the form $((r),(IR_{ir}))$ where IR_{ir} denotes the class of cubes which are incompatible with the repeated cube r. This is accomplished as follows:

i) For each row i find all $r \in SR_i$.

SR_i is found as follows:

For all pairs of classes(IC_{ij}, IC_{ik}),

If $(IC_{ij} \cap IC_{ik} \neq \emptyset)$

then $SR_i = SR_i \cup (IC_{ij} \cap IC_{ik})$.

(SR_i is initially empty).

ii) Find all R_{ir} (R_{ir} is a set of cubes that are compatible with repeated cube r in row i). For each row i do the following:

If $r \in SR_i$, then $\forall j$: if $r \in IC_{ij}$, then $R_{ir} = R_{ir} \cup IC_{ij}$

(R_{ir} is initially empty).

iii) For each row i, find all IR_{ir}.

IR_{ir} denotes the class of cubes which are incompatible with repeated cube r in row i. For each repeated cube $r \in SR_i$ do the following:

$IR_{ir} = A_i \ominus R_{ir}$

The operator "$\ominus$" is used to imply the symmetrical difference between two sets. The set of all such pairs(r, IR_{ir}) is denoted IR.

This function returns set IR and IR_LENGTH.

c) Function $REMOVE\_REPEATED\_CUBES(IC, SR, i, j)$;

The purpose of this function is to remove repeated cubes from set IC. This

function is called to remove repeated cubes that are no longer needed(the repeated cubes were used previously by function $FORM\_SET\_OF\_PAIRS\_IR(IC, A, i, j)$). Not only are the repeated cubes no longer needed, they must be removed from set IC so that the remaining classification of columns using set IC is done correctly. The resulting set IC, will contain pairs of classes of cubes containing only the non-repeated cubes. Any pair of incompatible classes(IC_{ij}, IC_{ik}), that has one or both classes which are empty sets, are removed from the set of pairs of classes in IC. This reduces the number of pairs in IC.

This function returns set IC and IC_LENGTH.

d) Function $APPEND\_IR\_TO\_IC(IC, IR, IC\_LENGTH, IR\_LENGTH)$;

This function simply appends set IR to set IC. Figure 3.13 shows an illustration of sets IR and IC for an arbitrary function having the classes shown. In Figure 3.13a, the relationship between the indexes of IC and the rows of the Karnaugh map and the output classes are shown. Shown in Figure 3.13b and 3.13c are arbitrary sets IC and IR. Note that the sets IC and IR have the same basic format. Both sets contain pairs of incompatible classes of cubes. The primary difference is the labeling scheme. Pairs in

IC

	i \ j	OC #1	OC #2	OC #3
		1	2	3
row 1	1	IC_{11}	IC_{12}	IC_{13}
row 2	2	IC_{21}	IC_{22}	IC_{23}

OC = Output Class

a)

IC =

$(IC_{11}; IC_{12}),$
$(IC_{11}; IC_{13}),$
$(IC_{12}; IC_{13}),$
$(IC_{21}; IC_{22}),$
$(IC_{21}; IC_{23}),$
$(IC_{22}; IC_{23}).$

b)

IR =

$(1; I_1),$
$(3; I_3),$
$(7; I_7),$

$S_R = \{1,3,7\}$

S_R = Repeated cubes

c)

IC + IR =

$(IC_{11}; IC_{12}),$
$(IC_{11}; IC_{13}),$
$(IC_{12}; IC_{13}),$
$(IC_{21}; IC_{22}),$
$(IC_{21}; IC_{23}),$
$(IC_{22}; IC_{23}),$
$(1; I_1),$
$(3; I_3),$
$(7; I_7),$

d)

Figure 3.13: Diagram showing labeling of IC and IR and the combined set

IC are of the form (IC_{ij}, IC_{ik}) while pairs in IR are of the form (r, IR_{ir}). Therefore, combine sets IC and IR to obtain the complete set of incompatible classes of cubes. The new combined set IC appears in Figure 3.13d.

This function returns the new set IC and IC_LENGTH.

e) Function $FORM\_SET\_OF\_PAIRS\_IB(IC, B, IC\_LENGTH, n)$;

This function forms the set of pairs of incompatible classes of columns IB. The pairs of incompatible classes of columns are of the form $(IB_{ij}, IB_{ik}) \in IB$. To find each class IB_{ij}, find all columns which contain at least one cube that is contained in class IC_{ij}, and put them together in class IB_{ij}. This is done as follows:

$$IB_{ij} = \{n \mid \forall i \forall j \;\; IC_{ij} \cap B_n \neq \emptyset \}$$

The pairs of classes IB_{ij} and IB_{ik} obtained from each row i are incompatible with each other and therefore constitute the set of pairs of incompatible classes of columns, denoted above as IB.

This function returns the set IB and IB_LENGTH.

f) Function $FORM\_GRAPH\_FROM\_IB(IB, IC\_LENGTH, n)$;

This step simply changes the way the compatibility/incompatibility information is stored. This step is not required for column minimization algorithms that are able to work directly with the incompatible classes of columns formed in the previous step. However, if the column minimization algorithm requires a graph with nodes that represent individual blocks of the bound set, then the data stored in the previous step is converted from the pairs of incompatible classes of columns to pairs of columns which are incompatible(or pairs of columns which are compatible). To convert to the pairs of columns which are incompatible, perform the following:

For each row i, assign every column in class IB_{ij} as pairwise incompatible with every column in class IB_{ik}.

The above conversion can create the data which corresponds to both the compatibility and incompatibility graphs simultaneously. This is done by simply using a two dimensional array with a bit set indicating that a pair is compatible, and if it is not set, then the pair is incompatible. The function

returns a pointer($GRAPH$) to data representing the desired graph.

3.4.2.2 Illustration Of The GCA Approach On Function F3

Problem Description For Function F3

Repeated here for completeness is the problem description for function $F3$.

Cube	X_1	X_2	X_3	X_4	Y_1	Y_2
1	0	0	2	0	1	1
2	3	0	-	1	0	0
3	3	1	0	-	-	0
4	2	1	3	0	0	1
5	-	1	1	1	1	-
6	1	0	3	0	0	-
7	2	-	3	1	0	1
8	3	1	1	0	1	0
9	1	1	-	1	-	0
10	3	0	2	0	1	-
11	1	1	3	1	0	0

Table 3.2: Table for Example Function $F3$

Table 3.2 describes the next example, function $F3$. The first column is the enumeration of cubes. The input variables are denoted X_1 thru X_4 and the output variables are Y_1 and Y_2. The multiple valued map corresponding to Table 3.2 is shown in Figure 3.14. Find the decomposition $H(A, G(B))$ given the specified bound and free sets. For this example, variable X_1 was chosen for the free set and variables X_2, X_3, and X_4 were chosen for the bound set variables. The following are the corresponding bound, free, and output partitions.

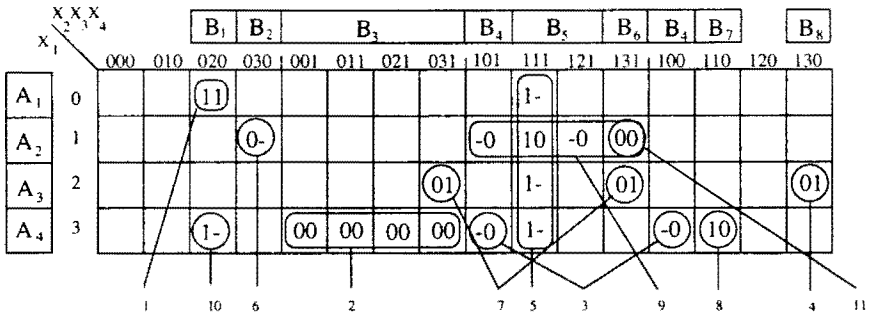


Figure 3.14: Multi-valued Map for Function $F3$

$$P(A) = P(X_1) = (1,5;\ 5,6,9,11;\ 4,5,7;\ 2,3,5,8,10)$$

$$= (A_1, ..., A_4)$$

$$P(B) = P(X_2) \cdot P(X_3) \cdot P(X_4) = (1,10;\ 6;\ 2,7;\ 3,9;\ 5,9;\ 7,9,11;\ 8;\ 4)$$

$$= (B_1, ..., B_8)$$

$$P(F) = (1,5,10;\ 2,3,6,9,11;\ 3,5,8,9,10;\ 4,6,7)$$

$$= (F_1, ..., F_4)$$

Thus:

$$A_1 = (1,5); \quad A_2 = (5,6,9,11); \quad A_3 = (4,5,7); \quad A_4 = (2,3,5,8,10)$$

$$B_1 = (1,10); \quad B_2 = (6); \quad B_3 = (2,7); \quad B_4 = (3,9);$$

$$B_5 = (5,9); \quad B_6 = (7,9,11); \quad B_7 = (8); \quad B_8 = (4)$$

$$F_1 = (1,5,10); \quad F_2 = (2,3,6,9,11); \quad F_3 = (3,5,8,9,10); \quad F_4 = (4,6,7)$$

Decomposition Of Function F3

Step 1: This step of the decomposition illustrates the **GCA** approach.

Execution of GCA Algorithm 3.4.2.

a) Generate set IC using function $FORM\_SET\_OF\_PAIRS\_IC(A, F, i = 4, j = 2)$. This function performs the following calculations:

$$\forall i \forall j : IC_{ij} = A_i \cap F_j$$

The first class to be formed is IC_{11}. It is formed as follows:

$$IC_{11} = (A_1 \cap F_1) = ((1,5) \cap (1,5,10)) = (1,5).$$

Similarly, the remaining IC_{ij} are found. This results in the following classes of cubes.

$$IC_{11} = (1,5), \quad IC_{12} = \emptyset, \quad IC_{13} = (5), \quad IC_{14} = \emptyset.$$

$$IC_{21} = (5), \quad IC_{22} = (6,9,11), \quad IC_{23} = (5,9), \quad IC_{24} = (6).$$

$$IC_{31} = (5), \quad IC_{32} = \emptyset, \quad IC_{33} = (5), \quad IC_{34} = (4,7).$$

$$IC_{41} = (5,10), \quad IC_{42} = (2,3), \quad IC_{43} = (3,5,8,10), \quad IC_{44} = \emptyset.$$

Function returns IC.

b) Generate set IR using function

$FORM\_SET\_OF\_PAIRS\_IR(IC, A, i = 4, j = 4)$. This function is executed in three parts as follows:

i) Find the set of repeated cubes SR_i for each row i. For all pairs of classes(IC_{ij}, IC_{ik}), check the following:

If $(IC_{ij} \cap IC_{ik} \neq \emptyset)$

then $SR_i = SR_i \cup (IC_{ij} \cap IC_{ik})$.

(SR_i is initially empty).

$SR_2 = (IC_{21} \cap IC_{22}) \cup (IC_{21} \cap IC_{23}) \cup (IC_{21} \cap IC_{24}) \cup (IC_{22} \cap$

$IC_{23}) \cup (IC_{22} \cap IC_{24}) \cup (IC_{23} \cap IC_{24})$.

$= ((5) \cap (6,9,11)) \cup ((5) \cap (5,9)) \cup ((5) \cap (6)) \cup ((6,9,11) \cap (5,9)) \cup$

$((6,9,11) \cap (6)) \cup ((5,9) \cap (6)) = (5,6,9)$.

Similarly, the remaining sets SR_i are found for each row i. This results in the following sets.

$SR_1 = (5)$,

$SR_2 = (5,6,9)$,

$SR_3 = (5)$,

$SR_4 = (3,5,10)$,

ii) Find all R_{ir}(set of cubes in row i compatible with cube r).

For each row i do the following:

If $r \in SR_i$, then $\forall j$: if $r \in IC_{ij}$, then $R_{ir} = R_{ir} \cup IC_{ij}$ (R_{ir} is initially empty).

For row 1, there is only one repeated cube(i.e., $SR_1 = (5)$). Therefore, begin by generating the set R_{15}(row 1 and cube 5). Ignore empty sets.

Since $r = 5$ is an element of IC_{11} and IC_{13}, then

$$R_{15} = IC_{11} \cup IC_{13} = (1,5) \cup (5) = (1,5).$$

Similarly, the remaining sets R_{ir} are found for each cube r. Repeated cube 3 is contained only in row 4. Therefore:

$R_{43} = (2,3,5,8,10)$.

Repeated cube 5 is contained in all rows. Therefore:

$R_{15} = (1,5)$,

$R_{25} = (5,9)$,

$R_{35} = (5)$,

$R_{45} = (3,5,8,10)$.

Repeated cube 6 is contained only in row 2. Therefore:

$R_{26} = (1,6,5)$.

Repeated cube 9 is contained only in row 2. Therefore:

$R_{29} = (5,6,9,11)$.

Repeated cube 10 is contained only in row 4. Therefore:

$R_{410} = (3, 5, 8, 10)$.

Hence, the following are all the sets of R_{ir}:

$R_{43} = (2, 3, 5, 8, 10)$,

$R_{15} = (1, 5)$,

$R_{25} = (5, 9)$,

$R_{35} = (5)$,

$R_{45} = (3, 5, 8, 10)$,

$R_{26} = (1, 5, 6)$,

$R_{29} = (5, 6, 9, 11)$,

$R_{410} = (3, 5, 8, 10)$.

iii) Find all IR_{ir} and form the set of pairs of classes $(r, IR_{ir}) \in IR$.

For each repeated cube $r \in SR_i$ do the following:

$IR_{ir} = A_i \oplus R_{ir}$.

Examples:

$IR_{15} = A_1 \oplus R_{15} = (1, 5) \oplus (1, 5) = \emptyset$.

$IR_{25} = A_2 \oplus R_{25} = (5, 6, 9, 11) \oplus (5, 9) = (6, 11)$.

Remaining IR_{ir} are determined in the same way. Results for all IR_{ir} are shown below.

$IR_{43} = (\emptyset)$,

$IR_{15} = (\emptyset)$,

$IR_{25} = (6, 11)$,

$IR_{35} = (4, 7)$,

$IR_{45} = (2)$,

$IR_{26} = (5)$,

$IR_{29} = (\emptyset)$,

$IR_{410} = (2)$.

From these classes IR_{ir}, the desired set IR is formed.

$IR =$

$(3, IR_{43}) = (3,(\emptyset))$,

$(5, IR_{15}) = (5,(\emptyset))$,

$(5, IR_{25}) = (5,(6,11))$,

$(5, IR_{35}) = (5,(4,7))$,

$(5, IR_{45}) = (5,(2))$,

$(6, IR_{26}) = (6,(5))$,

$(9, IR_{29}) = (9,(\emptyset))$,

$(10, IR_{410}) = (10,(2))$.

Pairs which contain an empty set are discarded. Also, the pairs (r, IR_{ir}) obtained using the same repeated cube r, are combined. For example, $(5, IR_{25})$, $(5, IR_{35})$, and $(5, IR_{45})$ can be combined to $(5, (IR_{25} \cup IR_{35} \cup IR_{45})) = (5, (2, 4, 6, 11))$. A new label is arbitrarily chosen for the new combined class(IR_{55}).

$IR =$

$(5, IR_{55}) = (5,(2,4,6,7,11))$,

$(6, IR_{26}) = (6,(5))$,

$(10, IR_{410}) = (10,(2))$.

Since there are 3 pairs in the set, assign $IR\_LENGTH = 3$.

Function returns IR and IR_LENGTH.

c) Generate new IC using function

$REMOVE\_REPEATED\_CUBES(IC, SR, i = 4, j = 4)$;

Remove repeated cubes from all classes $IC_{ij} \in IC$.

$\forall i \forall j \; IC_{ij} = IC_{ij} \oplus SR_i$.

Therefore, the following is the set of incompatible classes IC_{ij} produced in part a) after the repeated cubes (cubes 3,5,6,9, and 10) were removed.

$IC_{11} = (1) \quad IC_{12} = \emptyset \quad IC_{13} = \emptyset \quad IC_{14} = \emptyset$

$IC_{21} = \emptyset \quad IC_{22} = (11) \quad IC_{23} = \emptyset \quad IC_{24} = \emptyset$

$IC_{31} = \emptyset \quad IC_{32} = \emptyset \quad IC_{33} = \emptyset \quad IC_{34} = (4, 7)$

$IC_{41} = \emptyset \quad IC_{42} = (2) \quad IC_{43} = (8) \quad IC_{44} = \emptyset$

Pair all combinations of incompatible classes (IC_{ij}, IC_{ik}) within each row. Ignore pairs containing an empty set as one of its two classes. After pairing all the classes(IC_{ij}, IC_{ik}), which do not include $\emptyset$, only one pair((2),(8)) is found. Therefore, this results in the following set $IC =$:

$IC =$

$(IC_{42}, IC_{43}) = ((2),(8))$.

Function returns IC and IC_LENGTH.

d) Append set IR to IC using function

$APPEND\_IR\_TO\_IC(IC, IR, IC\_LENGTH = 1, IR\_LENGTH = 3)$;

$IC =$

$(IC_{42}, IC_{43}) = ((2),(8))$,

$(5, IR_{55}) = (5,(2,4,6,7,11))$,

$(6, IR_{26}) = (6,(5))$,

$(10, IR_{410}) = (10,(2))$.

Function returns combined set IC and new length IC_LENGTH.

e) Generate set IB using function

$FORM\_SET\_OF\_PAIRS\_IB(IC, B, IC\_LENGTH = 4, n = 8)$.

IB is found accordingly:

$$IB_{ij} = \{n \mid \forall i \forall j \;\; IC_{ij} \cap B_n \neq \emptyset \}$$

For simplicity, only the column number is used to denote each column (e.g. B_i is shown as index i). The first class in IB to generate is IB_{11}. IB_{11} is found incrementally as follows(initially $IB_{11} = (\emptyset)$):

i) $IC_{11} \cap B_1 = (2) \cap (1, 10) = \emptyset$...

therefore IB_{11} remains unchanged,

ii) $IC_{11} \cap B_2 = (2) \cap (6) = \emptyset$...

therefore IB_{11} remains unchanged,

iii) $IC_{11} \cap B_3 = (2) \cap (2, 7) = (2)$...

therefore $IB_{11} = (3)$,

iv) $IC_{11} \cap B_4 = (2) \cap (3, 9) = \emptyset$...

therefore IB_{11} remains unchanged,

v) $IC_{11} \cap B_5 = (2) \cap (5, 9) = \emptyset$...

therefore IB_{11} remains unchanged,

vi) $IC_{11} \cap B_6 = (2) \cap (7, 9, 11) = \emptyset$...

therefore IB_{11} remains unchanged,

vii) $IC_{11} \cap B_7 = (2) \cap (8) = \emptyset$...

therefore IB_{11} remains unchanged,

viii)$IC_{11} \cap B_8 = (2) \cap (4) = \emptyset$...

therefore the resulting class for $IB_{11} = (3)$.

Similarly, the remaining IB_{ij} are found. This results in the following incompatible classes of columns.

$IB =$

$(IB_{11}, IB_{12}) = ((3),(7))$,

$(IB_{21}, IB_{22}) = (5,(2,3,6,8))$,

$(IB_{31}, IB_{32}) = (2,(5))$,

$(IB_{41}, IB_{42}) = (1,(3))$.

Note that each block(or column) in a pair of classes is incompatible with all columns in the opposite class.

Function returns IB and IB_LENGTH.

f) Construct the desired graph using function $FORM\_GRAPH\_FROM\_IB(IB, IB\_LENGTH = 4, n = 8)$. This function performs the following calculations:

For each index i, assign all columns in class IB_{i1} as pairwise incom-

patible with all columns in class IB_{i2}.

For pair 1:

$(IB_{11} \not\sim IB_{12}) = ((3) \not\sim (7))$. Therefore column 3 is incompatible with column 7. This forms the incompatible pair B_{37}.

For pair 2:

$(IB_{21} \not\sim IB_{22}) = (5 \not\sim (2,3,6,8))$. Therefore column 5 is incompatible with columns 2,3,6, and 8. This forms the incompatible pairs $B_{25}, B_{35}, B_{56},$ *and* B_{58}.

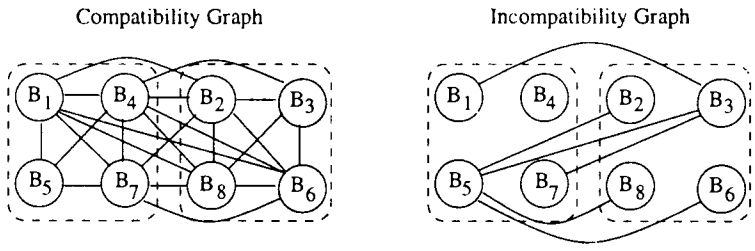
For pair 3:

$(IB_{31} \not\sim IB_{32}) = (2 \not\sim (5))$. Therefore column 2 is incompatible with column 5. This forms the incompatible pairs B_{25}.

For pair 4:

$(IB_{41} \not\sim IB_{42}) = (1 \not\sim (3))$. Therefore column 1 is incompatible with column 3. This forms the incompatible pairs B_{13}.

This results in the set of pairwise incompatible blocks I_B:



Shown circled with dashed lines are the classes chosen for the final cover set.

Figure 3.15: Compatibility and Incompatibility graphs for Function $F3$

$$I_B = (B_{13}, B_{25}, B_{35}, B_{37}, B_{56}, B_{58})$$

This set of pairwise incompatible blocks forms the incompatibility graph in Figure 3.15.

The set of pairwise compatible blocks C_B is obtained simply by removing the incompatible pairs from the set of all pairs of blocks:

$$C_B = (B_{12}, B_{14}, B_{15}, B_{16}, B_{17}, B_{18}, B_{23}, B_{24}, B_{26}, B_{27}, B_{28},$$
$$B_{34}, B_{36}, B_{38}, B_{45}, B_{46}, B_{47}, B_{48}, B_{57}, B_{67}, B_{68}, B_{78})$$

This set of pairwise compatible columns forms the compatibility graph in Figure 3.15. This completes the illustration of the **GCA** approach to create the compatibility graph.

Steps 2-4: The remaining steps in the decomposition(steps 2 thru 4) are completed

in the same way as in the example illustrating the **PCA** approach in Section 3.3.2.2. Therefore, they are not repeated here.

3.5 Analysis Of The New Approach Versus The Classical Approach

In this section an analysis of the new approach(**GCA**) introduced in Section 3.4 versus the classical approach(**PCA**) introduced in Section 3.3 is presented.

Note: The following analysis is done only for single output funtions. The reasons for this are:

1) because any multiple output function are replaced by multiple single output functions

   and

2) because the formula for single output functions for the new approach is much simpler to express mathematically.

The following are the formulas used to determine the number of calculations (intersection and union operations) required by each approach.

For the **PCA** approach of Luba et al[31], the expression for finding pair-wise column compatibility is:

$$P(A) \cdot (B_i \cup B_j) \subseteq P(F)$$

or more specifically,

$$A_k \cap (B_i \cup B_j) \subseteq F_m$$

and the number of required calculations is:

$$\mathbf{PCA} \;=\; R \times O \times \binom{C}{2}$$

where $\binom{n}{r} \;=\; \frac{n!}{r!\,(n-r)!}$

For the new approach(**GCA**), the expression for finding pairs of incompatible classes of columns is:

$$P(A) \cdot P(F) \cdot P(B)$$

or more specifically,

$$A_k \cap F_m \cap B_i$$

and the number of calculations required is:

$$\mathbf{GCA} = R \times O \times C$$

Variables defined:

$\mathbf{PCA}$ = "Pair Compatibility Approach",

$\mathbf{GCA}$ = "Group Compatibility Approach",

A_k = individual blocks of the free set $P(A)$,

B_n = individual blocks of the bound set $P(B)$,

F_m = individual blocks of the output set P_F,

$|A|$ = number of variables in the free set,

$|B|$ = number of variables in the bound set,

$C = 2^{|B|}$ = number of columns in the bound set,

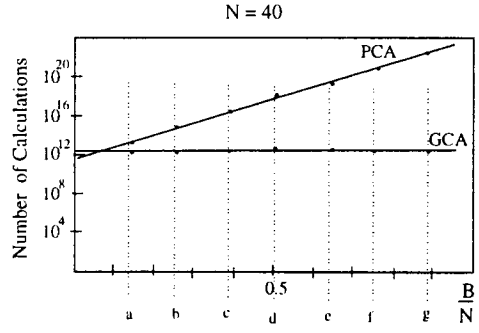
$R = 2^{|A|}$ = number of rows in the free set,

Y = number of output variables,

$O = 2^{|Y|}$ = number of blocks in the output partition.

Figure 3.16 presents plots of the two approaches represented by the formulas $\mathbf{PCA}$ and $\mathbf{GCA}$ for a constant total number of variables (N) with varying numbers

	N	|A|	|B|	PCA	GCA
a	40	35	5	3.4×10^{15}	2.2×10^{12}
b	40	30	10	1.1×10^{18}	2.2×10^{12}
c	40	25	15	3.6×10^{19}	2.2×10^{12}
d	40	20	20	1.1×10^{18}	2.2×10^{12}
e	40	15	25	3.7×10^{19}	2.2×10^{12}
f	40	10	30	1.2×10^{21}	2.2×10^{12}
g	40	5	35	3.8×10^{22}	2.2×10^{12}



	N	|A|	|B|	PCA	GCA
h	30	25	5	3.3×10^{11}	2.1×10^{9}
i	30	20	10	1.1×10^{15}	2.1×10^{9}
j	30	15	15	3.5×10^{16}	2.1×10^{9}
k	30	10	20	1.1×10^{17}	2.1×10^{9}
l	30	5	25	3.6×10^{17}	2.1×10^{9}

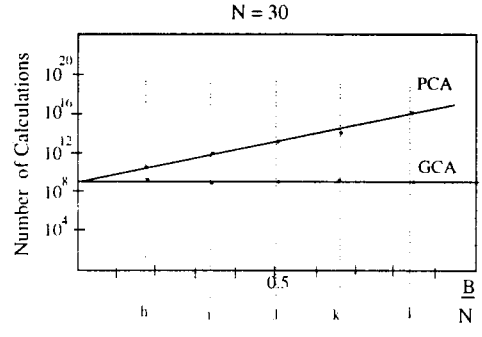


Figure 3.16: Plots of the two approaches represented by the formulas **PCA** and **GCA** for a constant total number of variables(N) and varying numbers of variables in the free set(A) and bound set(B).

of variables in the bound and free sets. Note that when the number of variables in the bound set is much larger than the number of variables in the free set, there exists several orders of magnitude difference in the number of calculations (intersections and unions) required.

Similarly in Figure 3.17, one can observe several orders of magnitude difference in the number of calculations when the number of variables in the bound set are much greater than the number of variables in the free set.

The analysis of the two approaches compared illustrates dramatic differences in the number of calculations required by each of the approaches when the number

	N	|A|	|B|	PCA	GCA
m	15	10	5	1.0×10^{7}	6.5×10^{4}
n	20	10	10	1.1×10^{9}	2.1×10^{7}
o	25	10	15	1.1×10^{12}	6.7×10^{9}
p	30	10	20	1.1×10^{15}	2.1×10^{12}
q	35	10	25	1.1×10^{18}	6.9×10^{14}
r	40	10	30	1.2×10^{21}	2.2×10^{17}
s	45	10	35	1.2×10^{24}	7.0×10^{19}

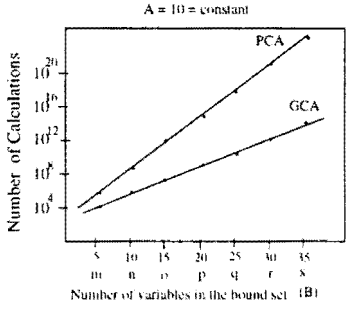


Figure 3.17: Plots of the two approaches represented by the formulas **PCA** and **GCA** when the number of variables in the bound set are much greater than the number of variables in the free set

of variables in the bound set is large.

However, it can be expected that actual results for computation time required for each approach would not differ in the number of calculations, as dramatically as indicated by the analysis. The comparisons made do however suggest a potential for significant savings using the new approach.

3.6 Experimental Results

The following tables show the comparisons of execution times for the implemented algorithms which were presented in this chapter. In Section 3.6.1, results are shown for the comparisons between the **PCA** and **GCA** algorithms in complete decompositions of MCNC benchmarks. In Section 3.6.2, results are shown for the comparisons between the **PCA** and **GCA** algorithms in partial decompositions of FLASH benchmarks with bound set sizes specified. The following is some general information explaining the versions of program GUD used in the comparisons:

GUD(**GCA**): Version of program GUD using a new algorithm to calculate column compatibility. This algorithm is based on checking compatibility of columns by classifying groups of columns as compatible or incompatible.

GUD(**PCA**): Version of program GUD using a commonly used algorithm to calculate column compatibility. This algorithm is based on checking compatibility of columns one pair of columns at a time. This method of calculating column compatibility is the same as used in program the program(DEMAIN) by Luba et. al.

Table 3.3: User time spent calculating column compatibility on MCNC benchmarks using different approaches(**PCA** vs **GCA**) with varying sizes of bound sets.

Program				GUD(**GCA**)	GUD(**PCA**)
Benchmarks	inputs	outputs	input cubes	time(s)	time(s)
5xp1	7	10	143	15	72
Z5xp1	7	10	141	19	75
adr2	4	3	24	0	0
b12	15	9	72	51	321
bw	5	28	97	6	17
c8	28	18	166	9	10
cc	21	20	96	18	3
con1	7	2	18	2	3
ex5	8	63	214	210	2303
f51m	8	8	154	38	165
misex1	8	7	40	10	18
rd53	5	3	63	1	2
rd73	7	3	274	20	120
rd84	8	4	515	229	787
root	8	5	256	121	803
squar5	5	8	56	1	2
xor5	5	1	32	0	0

3.6.1 Comparison Between PCA and GCA Algorithms in General Decompostions of MCNC Benchmarks

From the results in Tables 3.3 and 3.4, it can be observed that the **GCA** approach clearly out performs the **PCA** approach, in terms of execution time, on nearly every benchmark example. Recall that a reduction in execution time is the primary contribution of the new approach. For clarification, it should be noted what these execution times represent. The execution times for each approach are the sum total of user time spent in calculating column compatibility throughout the

decomposition process on each benchmark. For example, one benchmark may have several subfunctions to decompose in the decomposition process. Each subfunction may require many graphs to be constructed by either approach before a bound set is selected which yields an acceptable decomposition. It is important to note that each approach constructs the same identical graphs on each benchmark. The only difference in the decomposition process is which approach is used to construct the graphs. Another important point to make is that the basic partitioning strategy used always tries small bound sets first and then if no decomposition is found then the number of variables in the bound set is increased by one. The bound set size was limited to a maximum of twelve. However, the size of most of the bound sets which resulted in an acceptable decomposition were either two or three variables. This is a significant point to make in the comparison of the different approaches because the new approach(**GCA**) was expected to have a significant advantage when bound sets are large and no advantage when small. These results show that the new approach is faster even when bound sets are small.

If most of the high quality decompositions result when small bound sets are used, then why would you ever want to use large bound sets? An answer to this question is that future programs which have effective variable partitioning and encoding methods for large bound sets may result in the highest quality decompositions. At least a couple decomposition examples resulted in the highest quality decomposition when bound sets were large despite the fact that random partition-

ing and random encoding methods were used. Therefore, while the **GCA** approach requires less execution time even when bound sets are small, it also provides the capability to check large bound sets which can't be checked feasibly by the **PCA** approach.

Table 3.4: Summary of Results for Table 3.3

	Category					
Program	A(sec)	B(sec)	C(sec)	D	E	F
GUD(**GCA**)	750	44	229	15	88%	1
GUD(**PCA**)	4,701	277	2,303	1	6%	0

Explantion of the Categories listed in the Summmary of Results table.

```
Categories:

A-Total Time(all benchmarks).
B-Average Time per benchmark.
C-Maximum Time for any benchmark.
D-Number of times an algorithm had the lowest user time(including
ties).
E-% of times when an algorithm had the lowest user time(including
ties).
F-Number of times an algorithm had a user time which was at least
  one order of magnitude(x10) faster than the competing algorithm.
```

Table 3.5: Time spent calculating column compatibility on FLASH benchmarks using two different methods(**PCA** vs **GCA**) with two variables in the bound set.

Program				GUD(PCA)	GUD(GCA)
Benchmarks	inputs	outputs	cubes	time(s)	time(s)
psu_add0_90	12	1	410	0	0
psu_add2_90	12	1	410	1	1
psu_add4_90	12	1	410	2	0
psu_contains_4_ones_90	12	1	410	1	0
psu_greater_than_90	12	1	410	1	1
psu_interval1_90	12	1	410	1	0
psu_interval2_90	12	1	410	1	0
psu_majority_gate_90	12	1	410	1	1
psu_pal_90	12	1	410	1	0
psu_parity_90	12	1	410	1	1
psu_sim12_90	12	1	410	1	0
psu_substr1_90	12	1	410	1	0
psu_subtraction1_90	12	1	410	0	1
psu_subtraction3_90	12	1	410	1	0

Table 3.6: Summary of Results for Table 3.5

	Category						
Program	A(sec)	B(sec)	C(sec)	D	E	F	G
GUD(**GCA**)	5	0.3	1	13	93%	0	0
GUD(**PCA**)	13	0.9	2	6	43%	0	0

Explantion of the Categories listed in the Summmary of Results table.

```
Categories:

A-Total Time(all benchmarks).
B-Average Time per benchmark.
C-Maximum Time for any benchmark.
D-Number of times an algorithm had the lowest user time(including
ties).
E-% of times when an algorithm had the lowest user time(including
ties).
```

```
F-Number of times an algorithm had a user time which was at least
  one order of magnitude(x10) faster than the competing algorithm.
G-Number of times an algorithm had a user time which was at least
  two orders of magnitude(x100) faster than the competing algorithm.
```

3.6.2 Comparison Between PCA and GCA Algorithms on FLASH Benchmarks with Specified Bound Set Sizes

In this section, comparisons of results are shown for the **PCA** and **GCA** algorithms when the number of variables in the bound set is specified. For example, if the number of variables in the bound set is specified to be 10, then only bound sets of size 10 are used in the comparisons. This is unlike the comparisons in Section 3.6.1, where the number of variables in the bound set varied in size during the decomposition. In Tables 3.5 thru 3.9, only two graphs were constructed for each benchmark. The purpose of this was to control the decompositions so that comparisons could be made, not only between the each algorithm used, but also between execution times when the bound set size is varied.

Table 3.7: Time spent calculating column compatibility on FLASH benchmarks using two different methods(**PCA** vs **GCA**) with five variables in the bound set.

Program				GUD(PCA)	GUD(GCA)
Benchmarks	inputs	outputs	cubes	time(s)	time(s)
psu_add0_90	12	1	410	31	3
psu_add2_90	12	1	410	29	2
psu_add4_90	12	1	410	28	3
psu_contains_4_ones_90	12	1	410	31	1
psu_greater_than_90	12	1	410	33	2
psu_interval1_90	12	1	410	37	0
psu_interval2_90	12	1	410	35	1
psu_majority_gate_90	12	1	410	33	2
psu_pal_90	12	1	410	38	0
psu_parity_90	12	1	410	31	2
psu_sim12_90	12	1	410	32	1
psu_substr1_90	12	1	410	46	1
psu_subtraction1_90	12	1	410	35	2
psu_subtraction3_90	12	1	410	36	2

Table 3.8: Summary of Results for Table 3.7

	Category						
Program	A(sec)	B(sec)	C(sec)	D	E	F	G
GUD(**GCA**)	22	1.6	3	14	100%	14	0
GUD(**PCA**)	475	33.9	46	0	0%	0	0

Explantion of the Categories listed in the Summmary of Results table.

```
Categories:

A-Total Time(all benchmarks).
B-Average Time per benchmark.
C-Maximum Time for any benchmark.
D-Number of times an algorithm had the lowest user time(including
ties).
E-% of times when an algorithm had the lowest user time(including
ties).
F-Number of times an algorithm had a user time which was at least
  one order of magnitude(x10) faster than the competing algorithm.
G-Number of times an algorithm had a user time which was at least
  two orders of magnitude(x100) faster than the competing algorithm.
```

In Table 3.5, observe that there is very little difference in execution times when the number of variables in the bound set is 2. However, in Table 3.7, there is a substantial difference in execution times. The **GCA** approach consistently out performs the **PCA** approach when the number of variables in the bound set is 5. In fact, the **GCA** approach out performs the **PCA** approach by more than an order of magnitude in execution time on every benchmark. In Table 3.9, an even more dramatic difference can be observed between the execution times of the two approaches. When the number of variables in the bound set equal to 10,

the **GCA** approach out performs the **PCA** approach by more than two orders of magnitude in execution time on every benchmark! For additional information on the comparisons made, the reader is directed to the corresponding Summary of Results tables for each comparison made. The Summary of Results for each comparison are shown in Tables 3.6, 3.8, and 3.10.

Table 3.9: Time spent calculating column compatibility on FLASH benchmarks using two different methods(**PCA** vs **GCA**) with ten variables in the bound set.

Program				GUD(PCA)	GUD(GCA)
Benchmarks	inputs	outputs	cubes	time(s)	time(s)
psu_add0_90	12	1	410	358	1
psu_add2_90	12	1	410	202	1
psu_add4_90	12	1	410	141	1
psu_contains_4_ones_90	12	1	410	147	1
psu_greater_than_90	12	1	410	148	1
psu_interval1_90	12	1	410	142	0
psu_interval2_90	12	1	410	163	1
psu_majority_gate_90	12	1	410	115	1
psu_pal_90	12	1	410	114	1
psu_parity_90	12	1	410	130	2
psu_sim12_90	12	1	410	115	1
psu_substr1_90	12	1	410	122	1
psu_subtraction1_90	12	1	410	119	1
psu_subtraction3_90	12	1	410	127	1

Table 3.10: Summary of Results for Table 3.9

	Category						
Program	A(sec)	B(sec)	C(sec)	D	E	F	G
GUD(**GCA**)	14	1	2	14	100%	14	14
GUD(**PCA**)	2143	153	358	0	0%	0	0

Explantion of the Categories listed in the Summmary of Results table.

```
Categories:

A-Total Time(all benchmarks).
B-Average Time per benchmark.
C-Maximum Time for any benchmark.
D-Number of times an algorithm had the lowest user time(including
ties).
E-% of times when an algorithm had the lowest user time(including
ties).
F-Number of times an algorithm had a user time which was at least
  one order of magnitude(x10) faster than the competing algorithm.
G-Number of times an algorithm had a user time which was at least
  two orders of magnitude(x100) faster than the competing algorithm.
```

3.7 Concluding Remarks

In Tables 3.5-3.10, the results showed that the **GCA** approach clearly out performs the **PCA** approach in execution time by a substantial margin. From these comparisons, it was verified that the **GCA** algorithm does in fact perform much more efficiently than the **PCA** algorithm when larger bound sets are used in the decomposition process. In fact, it was demonstrated that when the bound set is large enough(five variables or more), the **GCA** approach out performs the

PCA approach by orders of magnitude in execution time.

It was expected that the actual results of the **GCA** algorithm to be better than the **PCA** algorithm when larger bound sets were used. However, it was not expected that the actual results would demonstrate such dramatic differences. Also, it was not expected that the results would be as consistent with the relative differences suggested by the pre-testing analysis. The importance of these results is that a partition based approach to Curtis style decompositions can now be used to perform column compatibility checking more efficiently. More importantly, these results show that the **GCA** algorithm can be used with larger bound sets to create the compatibility graph with little or no increase in the execution time. By being able to use larger bound sets, the search space of feasible decompositions is increased, thereby making it possible to find better decompositions.

CHAPTER 4

IMPROVING EFFICIENCY FOR ASHENHURST DECOMPOSITIONS USING THE GCA APPROACH

4.1 Introduction

This chapter is primarily concerned with a special case of the column minimization problem. The special case is referred to as an Ashenhurst decomposition. Recall, in Section 2.3.1.1 that an Ashenhurst decomposition is a special case of a Curtis style decomposition, where there is only one output from the predecessor block. It is important to note that the subject of column minimization is not a primary topic of this thesis. The only reason a column minimization approach is introduced here is to further illustrate yet another very significant application of the Group Compatibility Approach(**GCA**) presented in Chapter 3.

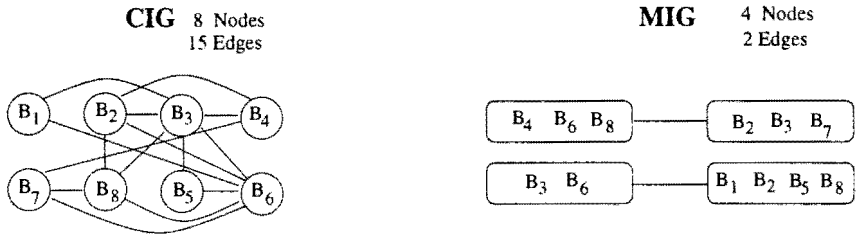
In addition to performing column compatibility checking very efficiently, intermediate data created using the **GCA** approach can be used with a modified column minimization algorithm(**MGCA**) to efficiently check for Ashenhurst decompositions when the number columns is greater than the number of rows or when the number of columns is sufficiently large. This condition can be expressed simply as ($|B|/|A| \geq 1$), where $|B|$ is the number of columns(or blocks in the bound set) and $|A|$ is the number of rows(or blocks in the free set). An analysis is

presented in Section 4.5 which illustrates how this condition was arrived at. When the number of blocks in the bound set is not sufficiently large then there may be little or no advantage to the **MGCA** approach. What is sufficiently large? The answer to this question depends on the particular function in question. This is because each function has its own distribution of minterms which affects the size and number of partitions blocks created for the bound set, the free set and the output set. The variations in functions make it difficult, if not impossible, to set a specific bound such that, above that specific bound, one approach will always perform more efficiently than another. However, it can be stated that for graph coloring approaches which operate directly on graphs with nodes representing columns, there is typically more calculations required to find the column multiplicity when there are many nodes and edges as opposed to when there are very few nodes and edges . Therefore, as the number of nodes(or columns) and edges increases, there is typically an increase in the number of calculations required for graph coloring, clique covering, or set covering. However, if the modified graph coloring approach can make use of a graph composed of the incompatibility classes of columns found using the **GCA** approach presented in Section 3.4, then the number of nodes and edges may be greatly reduced. It is the use of a reduced set of nodes and edges which is the primary basis for improved efficiency of the new approach(**MGCA**) presented in the following sections.

In this chapter, an incompatibility graph with nodes representing columns(or

a \ bcd	B_1 000	B_2 001	B_3 010	B_4 011	B_5 111	B_6 110	B_7 101	B_8 100
0	–	1	1	0	–	0	1	0
1	1	1	0	–	1	0	–	1

a) Karnaugh map



b) Conventional Incompatibility Graph with nodes representing columns and edges between nodes indicate that the columns connected by the edge are incompatible.

c) Modified Incompatibility Graph with nodes representing incompatibility classes of columns generated using the GCA approach to column compatibility checking. Edges between nodes indicate that all columns in one class(or node) are incompatible with all columns in the other class connected by the edge.

Figure 4.1: Illustration of number of nodes in a Conventional Incompatibility Graph and a new Modified Incompatibility Graph on the same Function

blocks in the bound set) will be referred to as a conventional incompatibility graph(**CIG**). Similarly, the incompatibility graph with nodes representing the incompatibility classes of columns(IB_{ij}) found using the **GCA** approach will be referred to as a modified incompatibility graph(**MIG**).

The following is an example presented to illustrate how the number of nodes that must be colored can be reduced by using the incompatibility classes of columns to represent nodes as in a **MIG** graph instead of using columns to represent nodes as in the **CIG** graph. More specifics about the **MIG** graph are presented in Section 4.2.1. In Figure 4.1a is shown a Karnaugh map used to illustrate the

difference in the number of nodes in a **CIG** graph and a **MIG** graph generated using the **GCA** approach. Observe from the graphs in Figure 4.1b and Figure 4.1c that there are significantly fewer nodes and edges in the modified incompatibility graph. Clearly, an algorithm which can take advantage of the **MIG** graph should require fewer calculations to color the graph as it has fewer nodes and edges.

The format of this chapter is as follows: In Section 4.2.1, a brief overview of the modified graph coloring approach(**MGCA**) is presented. Presented in Section 4.2.2 is the algorithm for the **MGCA** approach to column minimization. Detailed examples illustrating the the **MGCA** approach are presented in Sections 4.3 and 4.4. In Section 4.5, an analysis is presented which compares the number of nodes in **CIG** graphs vs. **MIG** graphs. Finally, in Section 4.6, some concluding remarks are presented.

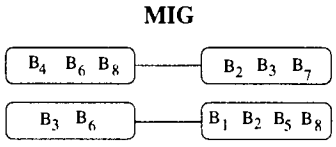
4.2 Modified Graph Coloring Approach: MGCA

Before presenting an overview of the **MGCA** approach to column minimization, I would like to briefly re-summarize what constitutes an Ashenhurst decomposition and illustrate what the input to the **MGCA** approach is.

Recall from Section 2.2, that an Ashenhurst decomposition, unlike a Curtis decomposition, is a decomposition with only one output from the predecessor block. This one output corresponds to a cover set Π_G having exactly 2 compatible classes of columns. The most important point is that in order to obtain an Ashenhurst de-

$$\mathbf{IB} = \{(\ \mathbf{IB}_{11}\ ;\ \mathbf{IB}_{12}\ )\ (\ \mathbf{IB}_{21}\ ;\ \mathbf{IB}_{22}\ )\}$$
$$= \{(\{B_4, B_6, B_8\}\ ;\ \{B_2, B_3, B_7\})\ (\{B_3, B_6\}\ ;\ \{B_1, B_2, B_5, B_8\})\}$$

a) Set of pairs of incompatibility classes of columns(**IB**) generated using the **GCA** approach to column compatibility checking.



b) Modified Incompatibility Graph(**MIG**) with nodes representing incompatibility classes of columns generated using the **GCA** approach to column compatibility checking. Edges between nodes indicate that all columns in one class(or node) are incompatible with all columns in the other class connected by the edge.

Figure 4.2: Illustration of set IB and its graphical representation

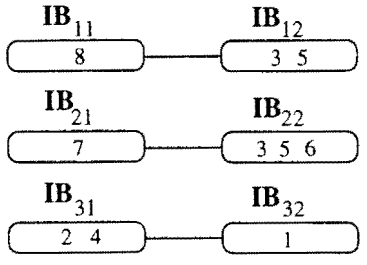
composition, one all columns must be combined into two classes of columns, where all columns within each class are mutually compatible. This is the minimum necessary criterium for the existence of an Ashenhurst decomposition. Therefore if not all columns within each class of a candidate cover set are mutually compatible, then that candidate cover set does not satisfy the criterium necessary for an Ashenhurst decomposition.

The second most important piece of information to understand is what the input to the **MGCA** approach is. The input to the **MGCA** approach is a set of pairs of incompatibility classes referred to as the set IB. Recall that the set IB is generated by the **GCA** approach in the process of column compatibility checking. Also, recall that within a pair of classes in the set IB, all columns within one class are incompatible with all columns within the other class. A **MIG** graph is simply the graphical representation of the set IB. Each pair of nodes connected by an edge in a **MIG** graph corresponds to a pair of classes $(IB_{i1}, IB_{i2}) \in IB$.

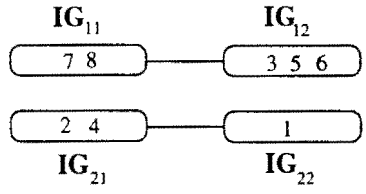
Shown in Figure 4.2 is an illustration of the relationship between the set IB and the corresponding **MIG** graph.

4.2.1 Brief Overview Of The MGCA Approach

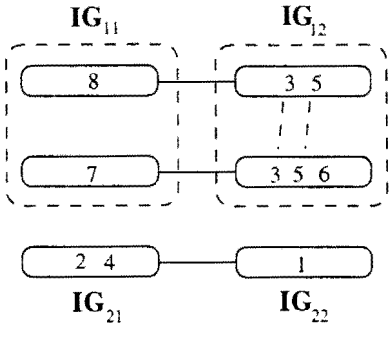
Step 1: In Figure 4.3a is shown an arbitrary input **MIG** graph with nodes labeled for referencing purposes. Contained in the nodes of the **MIG** graph are columns 1 thru 8, which for this example is the complete set of columns. No Karnaugh map is shown as it is not used in the explanation. Step 1 of the **MGCA** approach is to combine all nodes(IB_{ij}) which have at least one column in common into supernodes(combined nodes). In the remainder of this chapter, the requirement that nodes, which have at least one column in common, must be combined into a supernode will be referred to as Requirement 1. Also, all nodes, which are connected by an edge with any nodes that have been combined to a particular supernode, must be combined together to a different supernode. In the remainder of this chapter, this additional requirement will be referred to as Requirement 2. Requirements 1 and 2 are necessary requirements to determine if an Ashenhurst decomposition exists for a given bound set. The reason that they are necessary is because no column may be an element of both classes in the cover set unless they are compatible with all other columns. If Requirements 1 and 2 are complied with, then no column will be combined to both supernodes connected by an edge unless they are compatible with all other columns. Hence, because supernodes and classes



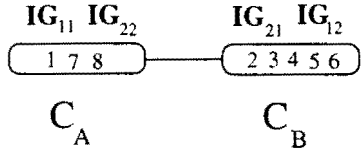
a) Input **MIG** graph



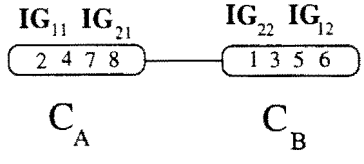
c) Reduced graph showing nodes combined together.



b) Columns 3 and 5 are contained in more than one node(or class) and therefore these nodes must be combined together into the super node IG_{12}. This supernode may be given color B. All nodes which are connected by an edge to one of the nodes assigned color B must be assigned color A.



d) Optional cover set number 1 showing the final classes given colors A and B.



e) Optional cover set number 2 showing the final classes given colors A and B.

Figure 4.3: Illustration of the MGCA Approach

are synonymous, this implies that no column will be combined to both classes in the cover set unless they are compatible with all other columns. This, of course, is assuming that an Ashenhurst decomposition exists for the given bound set. If an Ashenhurst decomposition doesn't exist for the given bound set, then there will be columns added to both classes. If this happens, then the algorithm would return no Ashenhurst decomposition exists and terminate. If a column is compatible with all other columns, then it will not be an element of any of the classes IB_{ij}(or nodes in the **MIG** graph) because all columns which are elements of the classes IB_{ij} are incompatible with at least one other column. Columns which are compatible with all other columns are added to either or both classes in the cover set in the last step of the **MGCA** algorithm.

In Figure 4.3b, observe that columns 3 and 5 are elements of more than one node(i.e., IB_{22} and IB_{12}). Therefore, these nodes must be combined in order to comply with Requirement 1. Similarly, nodes IB_{11} and IB_{21} must be combined in accordance with Requirement 2. Shown in Figure 4.3c is the reduced graph showing the resulting supernodes for the first step of the **MGCA** approach. For this simple example there were only two pairs of nodes that were combined into supernodes in accordance with Requirements 1 and 2. However, when there are many more nodes in the **MIG** graph(many rows), there will be many pairs of nodes combined into supernodes in accordance with Requirements 1 and 2.

Each time a pair of nodes have been combined with a supernode pair, then a

check is made to determine if an Ashenhurst decomposition is *not* possible for the given bound set as a result of the combination. This is done by simply checking if any columns are contained in both supernodes that are connected by an edge. From Figure 4.3c, observe that there are no columns contained in any two supernodes connected by an edge. Therefore, in this example, an Ashenhurst decomposition exists.

Step 2: After Step 1 has been completed and it has been determined that an Ashenhurst decomposition exists, then the reduced graph must be further reduced to two nodes to satisfy the minimum requirement of an Ashenhurst decomposition(i.e., two classes). If there were only two nodes in the reduced graph to begin this step, then no combining would be necessary. However, in this example there are four nodes remaining. This means that there is more than one "optional" combination of nodes to combine together and hence more than one "optional" cover set to select from. The method for selecting which combination of nodes to combine may be heuristic or random. For simplicity, a random selection method is used for the examples presented in this chapter. One such approach would be to randomly select two nodes to combine into a supernode. Then combine into a different supernode all nodes which are connected to the nodes combined in the previous supernode. For this example, there are 2 optional cover sets as shown in Figure 4.3d and Figure 4.3e. The cover set that is selected in this step is referred

to as the initial cover set because there may be other columns which have not been covered yet(added to at least one class in the cover set).

Step 3: The final step is to add, to both nodes, all columns that are compatible with all columns. These columns are columns which are not elements of the input **MIG** graph. In this example, there are no columns which are compatible with all columns. Therefore, either of the cover sets shown in Figure 4.3 would be acceptable as a final cover set. This concludes the overview of the **MGCA** approach to column minimization.

4.2.2 Algorithm For The MGCA Approach

The following is an algorithm for efficiently checking the existence of an Ashenhurst decomposition(i.e., which means, the incompatibility graph which can be reduced to 2 colors). The input to this algorithm is the set of incompatibility classes(**IB**) formed using the **GCA** algorithm from Section 3.4.

Algorithm parameters defined:

IB_LENGTH = Number of class pairs in IB.

IG_LENGTH = Number of class pairs in IG.

IB_REMAIN = List of columns that are compatible with all other columns.

ASH_EXISTS = Return value that indicates whether an Ashenhurst Decomposi-

tion exists.

Π'_G = The initial cover set.

Π_G = The final cover set.

Algorithm 4.2.1

Begin

Step 1: // Form pairs of supernodes(set IG) from the **MIG** *graph(set IB) and*

// determine if an Ashenhurst decomposition exists for the given bound set.

$(ASH\_EXISTS, IG, IG\_LENGTH) =$

$FORM\_SET\_OF\_PAIRS\_IG(IB, IB\_LENGTH);$

If $(ASH\_EXISTS = FALSE)$

Return No Ashenhurst Decomposition Exists.

Step 2: // Combine pairs of supernodes in set IG until only one pair remains.

$\Pi'_G = IG = COMBINE\_PAIRS\_IN\_IG(IG, IG\_LENGTH);$

Step 3: // Add all columns which are compatible with all other columns(i.e.,

// columns which do not appear in the **MIG** *graph) to both classes in* Π'_G.

$\Pi_G = ADD\_REMAINING\_COLUMNS(\Pi'_G, IB\_REMAIN);$

end.

Detailed explanation for functions used in algorithm 4.2.1.

Step 1: The basic idea of this step is to combine all nodes in the **MIG** graph(set IB), which have columns in common, to supernodes. More specifically, a pair of supernodes (IG_{i1},IG_{i2}) are initialized by setting them equal to the first pairs of nodes(IB_{11},IB_{12}) in the **MIG** graph. Remaining pairs of nodes (IB_{k1},IB_{k2}) are combined with the pair of supernodes if they have at least one column in common. After each pair (IB_{k1},IB_{k2}) is combined with a supernode, it is removed from the set of remaining pairs in set IB. Also, after each pair (IB_{k1},IB_{k2}) is combined with a supernode, a check is made to determine if there are any columns in IG_{i1} that are also in IG_{i2}. If so, then no Ashenhurst decomposition exists and the function returns immediately. If not, then the process of combining pairs of nodes continues until all pairs in IB have been checked for common columns. After all pairs in IB, which have columns in common with supernode pair (IG_{i1},IG_{i2}), have been combined with (IG_{i1},IG_{i2}), then a new supernode pair is initialized and the process is repeated. Once all pairs in IB have been included in a supernode, the function returns the set of supernodes(IG) and the return value ASH_EXISTS. The following are the pseudo code for the functions which perform Step 1 of this algorithm.

Pseudo code for Function $FORM\_SET\_OF\_PAIRS\_IG()$:

```
FORM_SET_OF_PAIRS_IG(IB, IB_LENGTH)
{
i = 1;
k = 1;
While(therearemorepairsofclassesinsetIB, dothefollowing)

{
(IB_k1, IB_k2) = GET_NEXT_PAIR(IB, k);
  // Initialize pair of classes(or supernodes) (IG_i1, IG_i1).
  IG_i1 = IB_k1;
  IG_i2 = IB_k2;
  // Remove (IB_k1, IB_k2) from the set IB.
  IB = REMOVE_PAIR(IB, (IB_k1, IB_k2));
  // Complete the formation of the pair of supernodes (IG_i1, IG_i1) from the
  // remaining set of pairs (IB_k1, IB_k2) ∈ IB as follows.
  (ASH_EXISTS, (IG_i1, IG_i2)) =
      MAKE_PAIR_OF_SUPERNODES(IB, (IG_i1, IG_i2));
  If(ASH_EXISTS = FALSE)
    Goto END_FUNCTION;
  // Add the new pair to the set of pairs of supernodes IG.
```

$IG = ADD\_PAIR\_TO\_IG(IG, (IG_{i1}, IG_{i2}))$;

$i++$;

$k = 1$;

} // *End While.*

$END\_FUNCTION$:

Return ASH_EXISTS, IG, *and* IG_LENGTH;

}

Pseudo code for Function $MAKE\_PAIR\_OF\_SUPERNODES()$:

$MAKE\_PAIR\_OF\_SUPERNODES(IB, (IG_{i1}, IG_{i2}))$

{

$NOT\_COMPLETE\_YET$:

$k = 1$;

$NEW\_PAIR\_ADDED = FALSE$;

// *The following pseudo code does the following:*

// *Add pairs* $(IB_{k1}, IB_{k2}) \in IB$ *to the pair of supernodes* (IG_{i1}, IG_{i2}) *if*

// *there is at least one column in common. Then check if an Ashenhurst*

// *decomposition is not possible. If no Ashenhurst decomposition is possible,*

// *then return* $ASH\_EXISTS = FALSE$. *While* $ASH\_EXISTS =$

// $TRUE$, *continue until all pairs in* IB *have been checked.*

$While(((IB_{k1}, IB_{k2}) = GET\_NEXT\_PAIR(IB, k)) \neq NULL)$

$\{$

$If(IG_{i1} \cap IB_{k1} \neq NULL)$

$\{IG_{i1} = IG_{i1} \cup IB_{k1};\ IG_{i2} = IG_{i2} \cup IB_{k2};\ \}$

$ElseIf(IG_{i1} \cap IB_{k2} \neq NULL)$

$\{IG_{i1} = IG_{i1} \cup IB_{k2};\ IG_{i2} = IG_{i2} \cup IB_{k1};\ \}$

$ElseIf(IG_{i2} \cap IB_{k1} \neq NULL)$

$\{IG_{i2} = IG_{i2} \cup IB_{k1};\ IG_{i1} = IG_{i1} \cup IB_{k2};\ \}$

$ElseIf(IG_{i2} \cap IB_{k2} \neq NULL)$

$\{IG_{i2} = IG_{i2} \cup IB_{k2};\ IG_{i1} = IG_{i1} \cup IB_{k1};\ \}$

$Else$

// There are no common columns in this pair of classes.

// Therefore, get the next pair to check.

$\{\ k++;\ Continue;\}$

$ASH\_EXISTS = CHECK\_IF\_ASH\_EXISTS(IG_{i1}, IG_{i2});$

$If(ASH\_EXISTS = FALSE)$

$Goto\ END\_FUNCTION;$

$ELSE$

$\{$

$IB = REMOVE\_PAIR(IB, (IB_{k1}, IB_{k2}));$

$NEW\_PAIR\_ADDED = TRUE;$

$\}$

$\}$ *// End While.*

```
If(NEW_PAIR_ADDED = TRUE)
  {
  // Make another pass to see if any more classes left in IB have any
  // columns in common with either supernode in the supernode pair.
  Goto NOT_COMPLETE_YET;
  }
END_FUNCTION :
Return ASH_EXISTS and (IG_i1, IG_i2).
}
```

Pseudo code for Function $CHECK\_IF\_ASH\_EXISTS()$:

```
CHECK_IF_ASH_EXISTS(IG_i1, IG_i2)
{
If(IG_i1 ∩ IG_i2 ≠ ∅)
  // No Ashenhurst decomposition exists for the bound set tried.
  Return ASH_EXISTS = FALSE
Else
  Return ASH_EXISTS = TRUE
}
```

No pseudo code is given for the functions listed below as they are trivial:

$(IB_{k1}, IB_{k2}) = GET\_NEXT\_PAIR(IB, k)$,

$IB = REMOVE\_PAIR(IB, (IB_{k1}, IB_{k2}))$,

$IG = ADD\_PAIR\_TO\_IG(IG, (IG_{i1}, IG_{i2}))$

A clarification should be made regarding removal of pairs from set IB. The set IB may be thought of as either an array of pairs or a list of pairs. For convenience, IB is expressed as an array in the pseudo code. When a pair is removed from set IB at index k, the empty slot at index k is replaced by the last pair in the array. This is stated in order to avoid confusion about how the pseudo code deals with empty slots in the array. The last element in the array is followed by a NULL pointer.

Step 2: Let C_A denote the set of columns given color A.

Let C_B denote the set of columns given color B.

Function $COMBINE\_PAIRS\_IN\_IG(IG, IG\_LENGTH)$;

If there is only one pair(IG_{i1}, IG_{i2}) in the set IG, then this pair forms the initial cover set $\Pi'_G = (C'_A,\ C'_B)$ where Π'_G may or may not cover(include) all columns. Columns which are not included in the initial cover set Π'_G(i.e.,

columns which are compatible with all other columns) are dealt with in the final step. Go to step 3.

If there is more than one pair(IG_{i1},IG_{i2}) in the set IG, then there is more than one combination of nodes(IG_{ij}) which can be given the same color. The selection of which combination to choose may be heuristic or random. However, if IG_{i1} is assigned to color A, then IG_{i2} must be assigned to color B. Combine all groups assigned with the same color to one group(or supernode). This will result in $\Pi'_G = (C'_A,\ C'_B)$

Function $COMBINE\_PAIRS\_IN\_IG(IG, IG\_LENGTH)$ returns Π'_G.

Step 3: Function $ADD\_REMAINING\_COLUMNS(\Pi'_G, IB\_REMAIN)$;

Add remaining columns not included in the initial cover set Π'_G(columns which are compatible with all columns) to either or both groups in Π'_G to complete the final cover set $\Pi_G = (C_A,\ C_B)$.

Function $ADD\_REMAINING\_COLUMNS(\Pi'_G, IB\_REMAIN)$ returns Π_G.

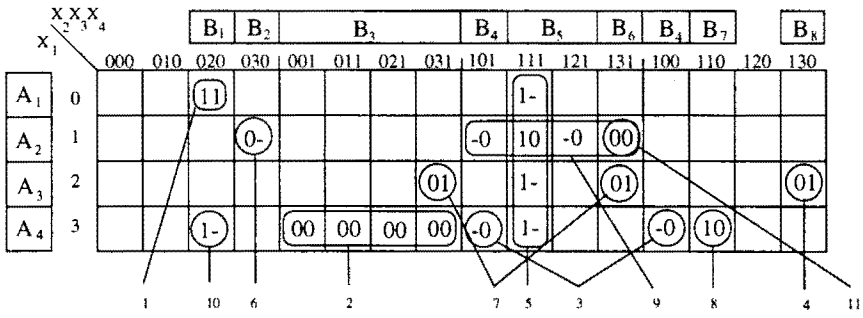


Figure 4.4: Multi-valued Map for function $F3$.

4.3 Example Of The MGCA Approach On Function $F3$

The following is an illustration of the **MGCA** approach to check for an Ashenhurst decomposition on function $F3$. Shown in Figure 4.4 is the multi-valued map for Function $F3$. Repeated here is the set of pairs of incompatibility classes formed previously in part d of the same example function presented in section 3.4.2.2.

$IB =$

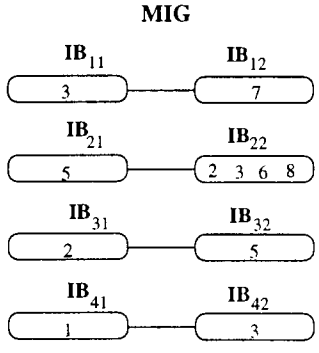
(((3), (7)),

((5), (2,3,6,8)),

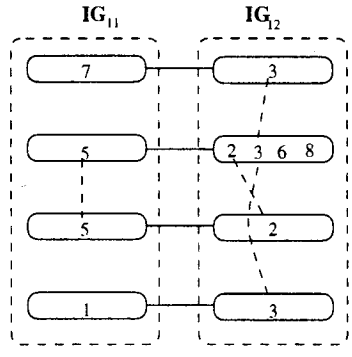
((2), (5)),

((1), (3))).

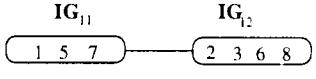
These pairs of incompatibility classes(IB) are used as the input to the Modified Graph Coloring Approach. These pairs of incompatibility classes are shown



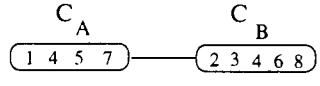
a) Modified Incompatibility Graph



b) Dashed lines indicate columns that are common to more than one node. Shown circled are nodes to be combined into supernodes according to requirements 1 and 2.

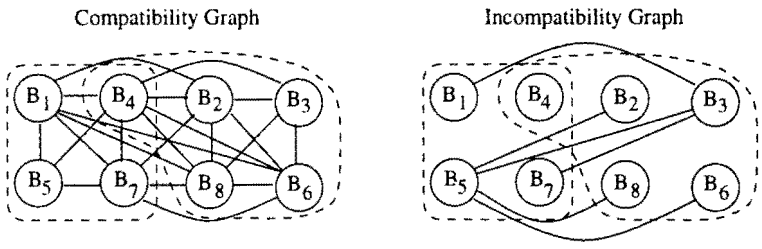


c) Resulting supernodes.



d) Final cover set showing the only remaining column(column 4) added to both nodes. Column 4 is the only column that is compatible with all other columns.

Figure 4.5: Illustration of steps in the **MGCA** Approach on Function $F3$



Shown circled with dashed lines are the classes chosen for the final cover set.

Figure 4.6: Compatibility and Incompatibility graphs for Function $F3$

in Figure 4.5a. Shown in Figure 4.6 is the conventional compatibility graph and conventional incompatibility graph for function $F3$. These two graphs are not used in the **MGCA** approach, but are shown to illustrate differences in types of graphs that can be used for column minimization. The algorithm is executed as follows:

Step 1: Start by initializing the first pair of supernodes(IG_{11},IG_{12}) to be equal to the first pair of nodes(IB_{11},IB_{12}) in the **MIG** graph. The assignment of IB_{11} and IB_{12} to either of the supernodes may be arbitrary. For this example, the assignments are as follows: $IG_{11} = IB_{12}$ and $IG_{12} = IB_{11}$. These assignments are done so that they were consistent with Figure 4.5b. Once a pair of nodes(IB_{ij},IB_{ik}) from the **MIG** graph are added to a pair of supernodes, they are removed from the set of nodes which must still be combined to a supernode.

Next, other nodes in the **MIG** graph are checked to see if there are any columns in common with either of the supernodes(IG_{11},IG_{12}). It is found that IB_{22} and IG_{12} have column 3 in common(i.e., $IG_{12} \cap IB_{22} = 3$). Therefore, IB_{22} is added to supernode IG_{12} in accordance with Requirement 1. Similarly, IB_{21} is added to supernode IG_{11} in accordance with Requirement 2.

Now a check is made to determine if the addition of the last pair of nodes with the supernodes(IG_{11},IG_{12}) violates the requirement necessary for an Ashenhurst decomposition.

If $IG_{11} \cap \ IG_{12} \neq \emptyset$, then the graph results in a column multiplicity greater than 2(Return no Ashenhurst decomposition exists). Otherwise continue.

Checking this condition for the pair(IG_{11},IG_{12}):

$$(IG_{11} \cap \ IG_{12}) = ((5,7) \cap (2,3,6,8)) = \emptyset$$

This process of combining nodes and checking the requirement necessary for an Ashenhurst decomposition is repeated until all pairs of nodes in the **MIG** graph have been added to supernodes which there are at least one column in common.

After checking the remaining pairs, it is found that nodes IB_{11}, IB_{22}, IB_{31}, and IB_{42} must be combined in accordance with Requirement 1. Similarly, nodes IB_{12}, IB_{21}, IB_{32}, and IB_{41} must be combined in accordance with Requirement 2. After combining these nodes to the supernodes in accordance with Requirements 1 and 2, it is found that all nodes in the **MIG** graph were able to be combined into just two supernodes(IG_{11} and IG_{12}). These are shown in Figure 4.5c.

Checking the requirement necessary for an Ashenhurst decomposition for the last pair of nodes combined with the supernodes results in the following:

$$(IG_{11} \cap IG_{12}) = ((1,5,7) \cap (2,3,6,8)) = \emptyset$$

The requirement necessary for an Ashenhurst decomposition is satisfied for all pairs of supernodes(in this case, there was only one pair). Therefore, return Ashenhurst decomposition exists. Go to step 2.

Step 2: Since there is only one pair(IG_{11}, IG_{12}) $\in IG$, then no combining of pairs is necessary. Therefore, this pair forms the initial cover set $\Pi'_G = (C'_A,\ C'_B)$. Go to step 3.

Step 3: Add columns not contained in any of the incompatible classes(columns which are compatible with all columns) to either or both groups in Π'_G to

complete the resulting cover set $\Pi_G = (C_A,\ C_B)$. Only column 4 has not been included in any of the incompatible classes. Therefore, column 4 is arbitrarily assigned to both classes in the initial cover set. Assigning a column to both classes in the cover set has the same meaning as assigning a column with both colors(i.e., colors A and B). This results in the final cover set $\Pi_G = ((1,4,5,7),(2,3,4,6,8))$ as shown in Figure 4.5d. This completes the illustration of the **MGCA** approach on Function $F3$.

4.4 Example Of The MGCA Approach On Function $F4$

Shown in Figure 4.7 is function $F4$ which is used in this example. Using the proceedure outlined in the **GCA** algorithm in section 3.4.2.1, the set of incompatible classes(IB) for function $F4$ is obtained:

$IB =$

(((7), (2)),

((5,6,7), (3)),

((1), (4,8))).

These pairs of incompatibility classes of columns are shown in Figure 4.8a. Shown in Figure 4.9 is the conventional compatibility graph and conventional incompatibility graph for function $F4$. These two graphs are not used in the **MGCA**

ab \ cde	000	001	011	010	100	101	111	110
00	1 -	1	-	-	-	-	0	8 -
01	9 -	-	1	-	0	0	0	16 -
11	17 0	-	-	1	-	-	-	24 1
10	25 0	-	-	0	-	-	-	32 0
Column reference numbers	1	2	3	4	5	6	7	8

Figure 4.7: Karnaugh map for function $F4$

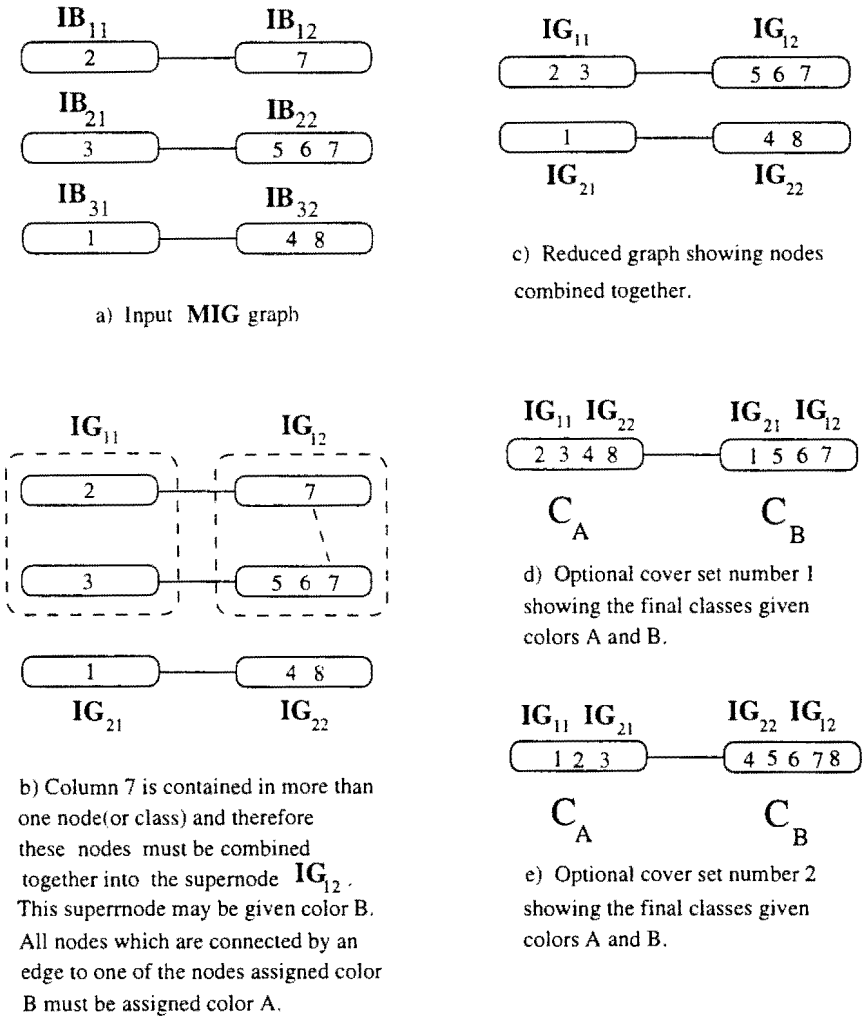
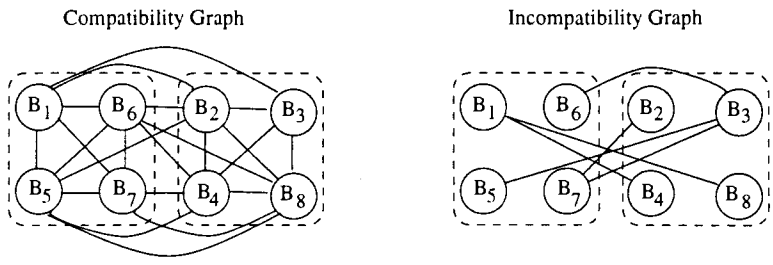


Figure 4.8: Illustration of steps in the **MGCA** Approach on Function $F4$



Shown circled with dashed lines are the classes chosen for the final cover set.

Figure 4.9: Compatibility and Incompatibility graph for Function $F4$

approach, but are shown to illustrate differences in types of graphs that can be used for column minimization. The algorithm is executed as follows:

Step 1: Start by initializing the first pair of supernodes(IG_{11},IG_{12}) to be equal to the first pair of nodes(IB_{11},IB_{12}) in the **MIG** graph. For this example, the assignments are as follows: $IG_{11} = IB_{11}$ and $IG_{12} = IB_{12}$.

Next, other nodes in the **MIG** graph are checked to see if there are any columns in common with either of the supernodes(IG_{11},IG_{12}). It is found that IB_{22} and IG_{12} have column 7 in common(i.e. $IG_{12} \cap IB_{22} = 7$). Therefore, IB_{22} is added to supernode IG_{12} in accordance with Requirement 1. Similarly, IB_{21} is added to supernode IG_{11} in accordance with Requirement 2. The supernodes that are formed are shown circled in Figure 4.8b.

Now a check is made to determine if combining the last pair of nodes with the supernodes(IG_{11},IG_{12}) violates the requirement necessary for an

Ashenhurst decomposition.

If $IG_{11} \cap IG_{12} \neq \emptyset$, then the graph results in a column multiplicity greater than 2(Return no Ashenhurst decomposition exists). Otherwise continue.

Checking this condition for the pair(IG_{11},IG_{12}):

$$(IG_{11} \cap IG_{12}) = ((2,3) \cap (5,6,7)) = \emptyset$$

Because the requirement is not violated, the process of adding nodes to supernodes continues. This process is repeated until all pairs of nodes in the **MIG** graph are added to supernodes which there are at least one column in common. After checking the remaining pairs, it is found that there are no other nodes which have any columns in common with either of the supernodes(IG_{11},IG_{12}). Therefore, the pair of supernodes(IG_{11},IG_{12}) are complete(i.e. no more nodes will be added to either supernode in this step).

The same process is repeated for a new pair of supernodes(IG_{21},IG_{22}) using the remaining nodes in the **MIG** graph. However, after removing all nodes in the **MIG** graph which are added to a supernode, only one pair of nodes remain(i.e., (IB_{31},IB_{32})). Therefore, the new pair of supernodes (IG_{21},IG_{22}) is simply assigned as the remaining pair of nodes(IB_{31},IB_{32}).

The requirement necessary for an Ashenhurst decomposition is not violated for supernodes(IG_{21},IG_{22})(i.e., $(IG_{21} \cap IG_{22}) = ((1) \cap (4,8)) = \emptyset$). Therefore, because the requirement necessary for an Ashenhurst decomposition was not violated for any of the supernode pairs, then an Ashenhurst decomposition exists for the given bound set. The complete set(IG) of pairs of supernodes is as follows:

$IG =$

$((IG_{11},IG_{12}),$

$(IG_{21},IG_{22})).$

$=$

$(((2,3),(5,6,7)),$

$((1),(4,8))).$

This is shown in Figure 4.8c.

Step 2: Since there is more than one pair(IG_{i1},IG_{i2}) $\in IG$, then there is more than one combination of classes(IG_{ij}) which can be given the same color. Shown in Figure 4.8d and Figure 4.8e are the optional assignments. For simplicity, a random selection is made for the combination of groups to add to color A as follows: $C_A = (IG_{11} \cup IG_{22})$. Therefore $C_B = (IG_{12} \cup IG_{21})$. This results in $\Pi'_G = (C_A, C_B) = ((2,3,4,8),(1,5,6,7))$.

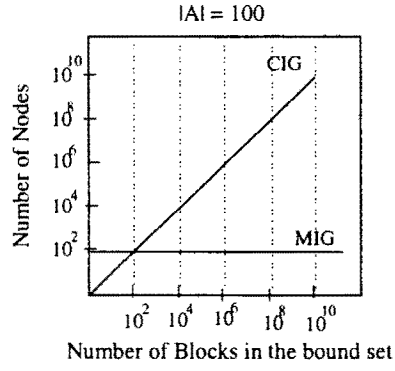
Step 3: Because there are no remaining columns(columns compatible with all other columns) which must be included in the initial cover set, then the final cover set is equal to the initial cover set Π'_G. Therefore, $\Pi_G = \Pi'_G = ((2,3,4,8),(1,5,6,7))$.

4.5 Analysis of CIG graphs vs. MIG graphs

In this section, comparisons are made between the number of nodes and the number of edges in **MIG** graphs vs. **CIG** graphs. Recall that the primary basis for the improved efficiency of the **MGCA** algorithm is that it can make use of a graph with fewer nodes when the number of blocks in the bound set is greater than the number of blocks in the free set(i.e., when $|B|/|A| \geq 1$). When the number of blocks in the bound set is **NOT** greater than the number of blocks in the free set, then the **MGCA** algorithm may be less efficient than an algorithm which uses a **CIG** graph or something similar. A more suitable algorithm for use when $|B|/|A| \geq 1$ can be found in [49]. For simplicity, all comparisons were made for single output functions.

|A|	|B|	MIG	CIG
100	100	200	100
100	10000	200	10000
100	1000000	200	1000000
100	100000000	200	100000000
100	10000000000	200	10000000000

a) Table for comparison of number of nodes in MIG graphs vs. CIG graphs. This comparison is for single output functions with a constant number of blocks in the free set and varying numbers of blocks in the bound set.



b) Graphical illustration for the comparison.

Figure 4.10: Comparison between number of nodes in a **CIG** Graph and a **MIG** Graph

4.5.1 Comparison of number of nodes in MIG graphs vs. CIG graphs

The number of nodes in a **CIG** graph is equal to the number of blocks in the bound set(denoted $|B|$). Therefore, no calculations are necessary to determine the number of nodes in a **CIG** graph.

$$\mathbf{CIG(nodes)} = |\mathbf{B}|$$

The number of nodes in a **MIG** graph is equal to:

$$\mathbf{MIG(nodes)} = |\mathbf{A}| \times |\mathbf{Y}|$$

$|A|$ is equal to the number of blocks in the free set and $|Y|$ is equal to the number of blocks in the output partition(or output class). For single output binary

functions, $|Y| = 2 =$ constant. Recall, the nodes in in the **MIG** graph represent the incompatibility classes of columns IB_{ij} obtained in the previous phase of the decomposition process(column compatibility checking) using the **GCA** approach. In order to clarify how the above formula was obtained for the number of nodes in a **MIG** graph, the following explanation is given: The classes IB_{ij} are obtained from the classes IC_{ij} in such a way that the number of classes IB_{ij} is equal to the number of classes IC_{ij} for single output binary functions(i.e., there is a one to one correspondence). Repeated here is the formula for creating the classes IC_{ij}.

$$\forall i \forall j : IC_{ij} \;=\; A_i \cap F_j$$

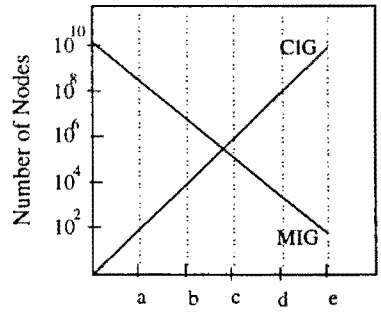
From this formula, the number of classes IC_{ij} is equal to $|A_i| \times |F_j|$, where $|A_i|$ is the number of blocks in the free set and $|F_j|$ is the number of blocks in the output set.

Presented in Figure 4.10 is a comparison of the number of nodes in **MIG** graphs vs. **CIG** graphs when the number of blocks in the bound set are varied. Observe from this figure that there are many more nodes in a **CIG** graph than a **MIG** graph when the number of blocks in the bound set is much greater than the number of blocks in the free set.

Similarly, in Figure 4.11 observe that there are many more nodes in a **CIG** graph than a **MIG** graph when the number of blocks in the bound set is much

	|A|	|B|	MIG	CIG
a	10000000000	100	20000000000	100
b	100000000	10000	200000000	10000
c	1000000	1000000	2000000	1000000
d	10000	100000000	20000	100000000
e	100	10000000000	200	10000000000

a) Table for comparison of number of nodes in MIG graphs vs. CIG graphs. This comparison is for single output functions with varying number of blocks in the free set and varying numbers of blocks in the bound set.



b) Graphical illustration for the comparison.

Figure 4.11: Another comparison between number of nodes in a **CIG** Graph and a **MIG** Graph

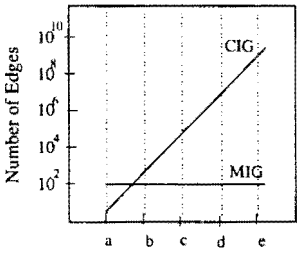
greater than the number of blocks in the free set. Another observation from this figure, is the cross-over point occurs approximately where $|B|/|A| = 1$. From the cross-over point, we can conclude, the **MIG** graph has fewer nodes than the **CIG** graph when $|B|/|A| \geq 1$.

4.5.2 Comparison of number of edges in MIG graphs vs. CIG graphs

Because it is not possible to formulate an exact expression for the average number of edges in **CIG** graphs without knowledge about specific functions, an assumption was made so that meaningful comparisons could be made between the number of edges in **MIG** graphs vs. **CIG** graphs. The assumption made was that the number of edges in **CIG** graphs is equal to 1/2 the number of edges in a complete graph(i.e., $\frac{n \times (n-1)}{4}$). Therefore, to calculate an approximate number of edges in a **CIG** graph, we simply need to know how many nodes are in the **CIG**

	|A|	|B|	MIG	CIG
a	100	10	100	22
b	100	100	100	2475
c	100	1000	100	249750
d	100	10000	100	24997500
e	100	100000	100	2499975000

a) Table for comparison of number of edges in MIG graphs vs. CIG graphs. This comparison is for single output functions with a constant number of blocks in the free set and varying numbers of blocks in the bound set. For CIG graphs, it was assumed that the number of edges were equal to half the number of edges in a complete graph.



b) Graphical illustration for the comparison.

Figure 4.12: Comparison between number of edges in a **CIG** Graph and a **MIG** Graph

graph and then divide the number of edges in a complete graph by two.

$$\mathbf{CIG(edges)} = \frac{|\mathbf{B}| \times (|\mathbf{B}| - 1)}{4}$$

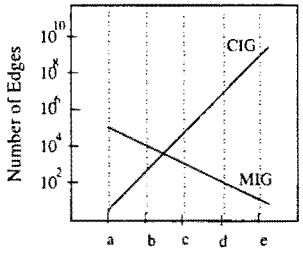
The number of edges in a **MIG** graph is equal to:

$$\mathbf{MIG(edges)} = \frac{|\mathbf{A}| \times |\mathbf{Y}|}{2}$$

For single output binary functions, $|Y| = 2 =$ constant. Recall, in **MIG** graphs, there is exactly one edge for every two nodes. Presented in Figure 4.12 is a comparison of the number of edges when the number of blocks in the bound set are varied.

	|A|	|B|	MIG	CIG
a	100000	10	100000	22
b	10000	100	10000	2475
c	1000	1000	1000	249750
d	100	10000	100	24997500
e	10	100000	10	2499975000

a) Table for comparison of number of edges in MIG graphs vs. CIG graphs. This comparison is for single output functions with varying number of blocks in the free set and varying numbers of blocks in the bound set. For CIG graphs, it was assumed that the number of edges were equal to half the number of edges in a complete graph.



b) Graphical illustration for the comparison.

Figure 4.13: Another comparison between number of edges in a **CIG** Graph and a **MIG** Graph

Observe from this figure that there are many more edges in a **CIG** graph than a **MIG** graph when the number of blocks in the bound set is much greater than the number of blocks in the free set. This of course is under the assumption that the average **CIG** graph has 1/2 the edges that are in a corresponding complete graph.

Similarly, in Figure 4.13 observe that there are many more edges in a **CIG** graph than a **MIG** graph when the number of blocks in the bound set is much greater than the number of blocks in the free set. Another observation from this figure, is the cross-over point occurs approximately where $|B|/|A| = 1$. From the cross-over point, we can conclude, the **MIG** graph has fewer edges than the **CIG** graph roughly when $|B|/|A| \geq 1$.

4.6 Concluding Remarks

It is known that execution time for graph coloring approaches generally increases with increases in the number of nodes and edges in a graph[65]. Therefore, if the comparisons illustrated in Figure 4.10 thru Figure 4.13 are representative of relative execution times required to color **CIG** graphs and **MIG** graphs, then it could be stated, using **MIG** graphs should require less execution time when $|B|/|A| \geq 1$ because **MIG** graphs would have fewer nodes and edges. Likewise, when $|B|/|A|$ is significantly less than one, then **CIG** graphs should require less execution time.

While the comparisons illustrated in Figure 4.10 thru Figure 4.13 showed that there can be significant advantages to using **MIG** graphs over **CIG** graphs when $|B|/|A| \geq 1$, it should not be over-looked that without an algorithm which is able to take advantage of **MIG** graphs, **MIG** graphs would be of no use. Examples in Sections 4.3 and 4.4 demonstrated that the **MGCA** algorithm is able to take advantage of **MIG** type graphs and therefore is able to efficiently check for Ashenhurst decompositions when $|B|/|A| \geq 1$.

CHAPTER 5

NEW APPROACH FOR COLUMN BASED INPUT/OUTPUT ENCODING

5.1 Introduction

Encoding in Curtis-style decompositions is the process of assigning codes to columns so that there is a mapping of output values from the predecessor sub-function to the successor sub-function. In doing so, the sub-functions created are functionally equivalent to the set of care values specified in the original function. Presented in the following sections is an input/output encoding approach designed to encode columns in such a manner as to achieve a more simplified total of the predecessor and the successor sub-functions.

For those unfamiliar with the basic difference between input and output encoding, a brief explanation is given as it applies to Curtis-style decompositions. Input encoding is the process of selecting codes for classes of the cover set Π_G so that the complexity of the successor sub-function is minimized. Examples of input encoding approaches are found in Iliev[24], Wan[65], Murgai[37], and Brayton[9]. Output encoding is the process of selecting codes for classes of the cover set Π_G so that the complexity of the predecessor sub-function is minimized. Examples of output encoding approaches can be found in Almeria[1] and Saldanha[57]. In

addition to input encoding approaches and output encoding approaches, other encoding approaches are designed to achieve a more simplified total of the predecessor and successor sub-functions concurrently. Encoding approaches of this type are referred to as combined input/output encoders. Examples of combined input/output encoding approaches can be found in Almeria[1], Devadas[16], Saldanha[57], and Selvaraj[58]. Of the different types of encoders mentioned, the input/output encoding approaches are typically more difficult to design but have the potential for a much simpler combination of the predecessor and successor functions.

There are many similarities between FSM minimization and Curtis type decompositions of switching functions. For example, state reduction in FSM minimization is very similar to column minimization in Curtis decompositions. Also, state assignment in FSM minimization is very similar to column encoding in Curtis decompositions.

However, there are significant differences between FSM minimization and Curtis type decompositions of switching functions. Most of the differences can be attributed to the fact that state machines have multiple states and switching functions do not(i.e., a switching function may be thought of as a state machine with only one state). Because the outputs of switching functions are not dependent on a state variable as in the case of state machines, the encoding problem is much simpler for the case of switching functions than for state machines. An important distinction between FSM minimization and Curtis type decompositions is

that states are reduced and encoded in FSM minimization whereas columns are reduced and encoded in Curtis decompositions. Consequently, the encoding constraints for state assignment are different from encoding constraints for Curtis decompositions. Constraints are groups of symbols that it is desired to assign codes to each of the symbols in the constraint such that the number of bits that differ in the codes is a minimum. Symbols in Curtis decomposition are multi-value labels corresponding to each partition block in a cover set. Symbols in state assignment are multi-value labels corresponding to each of the states in a state machine. Another important distinction between FSM minimization and Curtis type decompositions is that the primary goals are different. In typical FSM minimization problems, the primary goal is to minimize the number of product terms required to implement a state machine in a two-level PLA. However, in Curtis decompositions, the primary goal is typically to minimize the number of logic blocks(or DFC) required to implement a multi-level description of a given function.

Not surprisingly, it is difficult to make direct comparisons between encoding for state assignment and encoding for Curtis decompositions of switching functions. Because these significant differences, comparisons between encoding algorithms for state assignment and encoding algorithms for Curtis type decompositions are not presented except where subproblems are nearly identical. Two subproblems which are very similar in state assignment and encoding for Curtis decompositions

are: (1) creation of an edge-weighted connection graph and (2) embedding the the edge-weighted connection graph to the hypercube. The algorithms for these subproblems and relavent comparisons to state assignment are presented in Section 5.5.2.2. Additional reference to similarities and differences between FSM minimization and Curtis type decompositions can be found in [50].

Perhaps the most important criterion for determining what type of encoding program is best to use in Curtis decompositions, is the ratio of relative complexities of the predecessor and successor sub-functions. Unfortunately, it is difficult to assess the relative complexities of sub-functions especially when they are nearly the same size(in terms of the numbers of inputs and outputs). However, when the predecessor sub-function is much larger than the successor sub-function, one can reason that the predecessor will have a greater potential for further simplifications. For example, if the predecessor sub-function has 20 inputs and 3 outputs and the successor sub-function has 4 inputs and 3 outputs, then it would be obvious that the predecessor would have a greater potential for further simplification in terms of DFC(i.e. DFC=$(2^{20} * 3)$ vs. DFC=$(2^4 * 3)$) . Conversely, if the successor was much larger than the predecessor, then there would be greater potential for further simplification of the successor sub-function. Therefore, when the successor is much larger than the predecessor, then an input encoding approach should be used. Likewise, when the predecessor is much larger than the successor, then an output encoding approach should be used. And finally, when they are roughly

equal in size, then a combined input/output encoding approach should be used. An interesting characteristic of the new approach presented(**DC_ENC**) is that it will behave at times like each of the three types of encoders based on a heuristic cost function. This is explained in more detail in the algorithm presented.

Different applications of decomposition have different sets of objectives to be optimized. However, only three primary objectives are used in the proposed **DC_ENC** encoding approach presented in this chapter. These three primary encoding objectives are used to achieve the overall desired goal. The overall desired goal is to obtain a multi-level decomposition which has the minimum number of logic blocks(or DFC). One of the three encoding objectives is to minimize the Hamming distances between the columns in the Karnaugh map of the successor sub-function for the given bound and free sets. The second objective is to minimize the Hamming distances between the codes assigned to adjacent cells in the Karnaugh map of the predecessor sub-function. The third objective is to optimize the number of don't cares produced in the predecessor sub-function. The order of importance of these objectives varies depending on the sizes of the sub-functions. If the successor sub-function is much larger than the predecessor sub-function, then the first objective is the most important. If the reverse is true, then the last two objectives are most important. If the predecessor sub-function is much larger than the successor sub-function, then optimizing a combination of the last two objectives will most likely lead to an overall decomposition resulting in a lower

DFC. Each of the three objectives mentioned constitute a separate set of encoding constraints.

The following is the formulation of the encoding problem that is solved by the proposed **DC_ENC** encoding approach presented in this chapter. Given a set of various encoding constraints, use the minimum number of encoding bits to encode all columns in a Karnaugh map, such that at least 75% of the encoding constraints are satisfied. The optimal percentage and types of constraints to satisfy to achieve the overall goal of minimum DFC is different from function to function. Therefore, future work should include designing an algorithm which would give the exact or near exact percentage of constraints to satisfy on a function by function basis. However, the value of 75% was arrived at based on the following rationale. Each symbol receives a unique code. The size of a constraint is simply the number of symbols in the constraint. Therefore, if the number of bits that differ in the symbol codes is kept to a minimum, then more bits are required to assign codes to symbols in large constraints than in small constraints. An observation was made on several example decompositions that in every case there were very few large constraints compared to the number of small constraints. From these observations, 75% was chosen as a heuristic cut-off value.

Perhaps the most important characteristic of the **DC_ENC** approach is the ability to take advantage of overlap in compatible classes of columns to "produce" don't cares in the predecessor sub-function. Often times these don't cares produced

can greatly simplify the complexity of the predecessor sub-function. Techniques used in the **DC_ENC** encoding approach presented involve: (1) selection of suitable cover sets, (2) heuristics to optimize the quality of encoding at low computational cost, (3) multiple constraint satisfaction using an edge-weighted connection graph, and (4) use of Hamming distances to aid in assigning codes which results in simpler functions.

The format of this chapter is as follows: In Section 5.2, fundamentals of column-based encoding are presented. In Section 5.3, some of the basic definitions are introduced. In Section 5.4, a general encoding strategy is presented. In Section 5.5, a detailed description of the **DC_ENC** encoding approach is presented. In Section 5.6, a detailed example is presented to illustrate the **DC_ENC** encoding approach. Finally, in Section 5.7, conclusions are presented.

5.2 Definitions, Notations, and Terminology

Unless otherwise stated, a set which contains column indexes is the same as a set which contains columns(i.e., column is short for column index). Similarly, for sets containing cubes, symbols, classes, and etc., the word index is not included. The purpose for this is to avoid over use of the word index. However, where it is deemed necessary to distinguish between an element index and the contents of an element, then clarification is made.

Definition 5.2.1 *A* **disjoint cover set** *is a cover set which contains subsets of columns where no column is an element of more than one subset.*

Definition 5.2.2 *A* **nondisjoint cover set** *is a cover set which contains subsets of columns where at least one column is an element of more than one subset.*

Definition 5.2.3 Hamming Distance *between two code words(or vectors of variables) is defined as the number of digits in which these code words differ.*

Definition 5.2.4 *A* **Symbol** *is a multi-value label representing a set of mutually compatible elements. The elements in each set are either cubes or columns. More specifically, symbols are used to denote the individual sets(or classes) within a given cover set. A set of columns corresponding to a particular symbol may be referred to as a symbol set or symbol class or symbol group. For simplicity, a symbol set within the cover set is simply referred to as a symbol.*

Definition 5.2.5 *A* **Hypercube** *of dimension n is a set of 2^n vertices, where each vertex has exactly n edges connected between itself and n other vertices. No vertex in the hypercube is connected to the same set of edges as any other vertex in the hypercube.*

Definition 5.2.6 *A* **Supercube** *of a set of cubes is defined as the smallest cube containing all the minterms contained in the set of cubes.*

Example: Given cubes 000, 001, and 011. The supercube is 0 - -.

Definition 5.2.7 *A* **k-cube** *is a supercube of* 2^k *bit vectors where the number* k *indicates the number of don't cares in the cube.*

Example: Given the bit vectors 00, 01, 11, and 10, the resulting k-cube is "- -".

Definition 5.2.8 *A* **Face** *is a k-cube in a binary n-dimensional space, where* $k \leq n$. *Typically, a face refers to a sub-cube of an n-dimensional cube. A face in a hypercube can be thought of as a subhypercube.*

Definition 5.2.9 *A* **Column Constraint**, *as defined here, is a set(or group) of symbols in the cover set that a particular column is compatible with.*

Example:

The form of a constraint for column C_i is (S_1, S_3, S_7).

Definition 5.2.10 *A* **Face Embedding Constraint** *is a constraint which specifies that a set of symbols is to be assigned to one face of a binary n-dimensional cube, without any other symbol sharing the same face. A face embedding constraint is said to be satisfied if all the codes assigned to the symbols in the constraint occupy a single face in an n-dimensional cube. When a face embedding constraint is satisfied, it is possible that some of the codes in the face are unused.*

Example: Given is a three dimensional cube and the face embedding constraint for column $C_i = (S_1, S_3, S_4)$. If symbol S_1 is given code 000, symbol S_3 is given code 001, symbol S_4 is given code 011 and no symbol is assigned to code 010(unused

code), then the set of symbols in the constraint of column C_i are said to satisfy the face embedding constraint because the codes assigned to each of the symbols are contained in a single face(i.e., 0 - -) of the given three dimensional cube.

Definition 5.2.11 *A* **Hypercube Embedding Constraint** *is a special face embedding constraint containing exactly 2^k symbols. Like a face embedding constraint, a hypercube embedding constraint is said to be satisfied if all the codes assigned to the symbols in the constraint occupy a single face in an n-dimensional cube. However, when a hypercube embedding constraint is satisfied, then none of the codes in the face are unused.*

Example 1:

Given the hypercube embedding constraint for column $C_i = (S_1, S_4, S_3, S_7)$ and the following code assignments:

000 S_1
001 S_3
011 S_4
010 S_7
——
0 - - = Supercube of the codes in the constraint.

Remaining symbols not in the constraint.

110 S_2
111 S_5
101 S_6

The hypercube embedding constraint is satisfied because the supercube contains only codes contained in the constraint.

Example 2:

Given the hypercube embedding constraint for column $C_i = (S_1, S_4, S_3, S_7)$, and the following code assignments:

000 S_1
101 S_3
011 S_4
010 S_7

- - - = Supercube of the codes in the constraint.

Remaining symbols not in the constraint.

110 S_2
111 S_5
001 S_6

The hypercube embedding constraint is not satisfied because the supercube of the codes assigned to the symbols contains codes of symbols which are not contained in the constraint.

Definition 5.2.12 The Cost Function Ratio (CFR) *is an approximate measure of the relative sizes of the predecessor and successor functions in a Curtis-style decomposition. It is defined as follows:*

$$\mathbf{CFR} = \mathbf{Cost1} = \mathbf{G_{DFC}}/\mathbf{H_{DFC}}$$

$\mathbf{G_{DFC}}$ *is the DFC of predecessor sub-function G.*

$\mathbf{H_{DFC}}$ *is the DFC of successor sub-function H.*

Definition 5.2.13 Overlap Ratio $\mathbf{R_O}$ *is a measure of "overlap" of columns in symbols(or classes) of Π_G. Columns are said to "overlap" if they are compatible with more than one symbols in the cover set Π_G. R_O is defined as follows:*

$\mathbf{R_O = C_O / C_T}$.

$\mathbf{C_O}$ = *Number of columns which are compatible with more than one symbols in Π_G.*

$\mathbf{C_T}$ = *Total number of columns(excluding columns of all don't cares).*

5.3 Fundamentals Of Column-Based Encoding

The purpose of this section is to illustrate the fundamentals of column-based encoding. The following is a brief description of the general process of encoding columns in a Curtis style decomposition:

In a Curtis style decomposition, the encoding process follows the column minimization phase of the decomposition process. The primary input data to the encoding

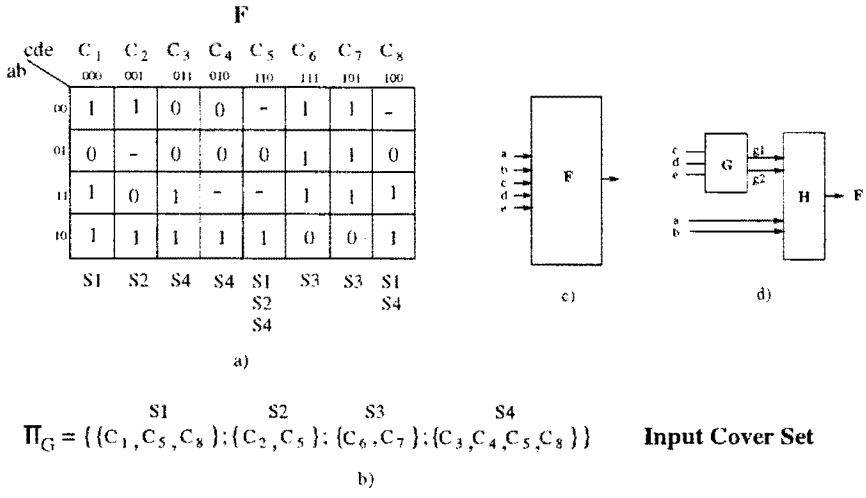


Figure 5.1: Function used to Illustrate Encoding of Disjoint vs. Nondisjoint Cover Sets

program are: the function to be decomposed, the cover set Π_G, and the column multiplicity. In column based encoding, the cover set Π_G is a set of subsets(CCs) containing columns. Though encoding approaches can be diverse and quite complicated, most column-based encoding approaches share two primary steps in the encoding process. The first step is to assign codes to the symbols(or classes) in the cover set Π_G. The second step is to assign codes to each column from among the symbol codes corresponding to the symbols that each column is compatible with.

Shown in Figure 5.1 is the Karnaugh map of an example function and an input cover set used to illustrate column based encoding of disjoint and nondisjoint cover sets. Below each column, in the Karnaugh map of function F, is a list of symbols that each column is compatible with. What is the primary advantage of encoding nondisjoint cover sets over disjoint cover sets? The primary advantage is that there are additional optional codes which may be assigned to columns when encoding

nondisjoint cover sets. If there are more codes to choose from using nondisjoint cover sets, why ever use disjoint cover sets? The answer is that many encoding approaches assume the input cover set is disjoint and therefore are not designed to handle the overlap in nondisjoint cover sets. Presented in Section 5.3.1 is an example of column based encoding of disjoint cover sets. Presented in Section 5.3.2 is an example of column based encoding of nondisjoint cover sets.

5.3.1 Encoding of Disjoint Cover Sets

If the choice is made to encode a disjoint cover set rather than a nondisjoint cover set, then it is necessary to remove all instances of each column from every subset(symbol) of Π_G except for one of them. There may be many optional cover sets which result from removing columns from the symbols of Π_G to make it a disjoint cover set. For the input cover set shown in Figure 5.2a, there are only two columns that are elements of more than one symbol. Column C_5 can be assigned to one of three symbols in Π_G. Column C_8 can be assigned to two of the symbols in Π_G. In total, there are six optional disjoint cover sets to choose from(i.e., there are six combinations of cover sets which can be produced by removing columns C_5 and C_8 from all but one symbol that each column is an element of). For simplicity, columns C_5 and C_8 are arbitrarily removed from all symbols except for one. Shown in Figure 5.2b, is the disjoint cover set produced by removing columns C_5 and C_8 from all but one symbol of the input cover set.

$$\Pi_G = \{\overset{S1}{\{C_1, C_5, C_8\}}; \overset{S2}{\{C_2, C_5\}}; \overset{S3}{\{C_6, C_7\}}; \overset{S4}{\{C_3, C_4, C_5, C_8\}}\}$$ **Input Cover Set**

a)

Encoding of columns with disjoint codes

$\Pi_G = \{\{C_1, C_5, C_8\}; \{C_2 \quad\}; \{C_6, C_7\}; \{C_3, C_4 \quad\}\}$ **Disjoint Cover**

b)

H

ab \ g1g2	00	01	11	10
00	1	1	1	0
01	0	-	1	0
11	1	0	1	1
10	1	1	0	1

S1 S2 S3 S4

c)

S1=00
S2=01
S3=11
S4=10

G

cd \ e	0	1
00	C_1	C_2
01	C_4	C_3
11	C_5	C_6
10	C_8	C_7

d)

G

de \ c	0	1
00	S1	S2
01	S4	S4
11	S1 S2 S4	S3
10	S1 S4	S3

e)

G

de \ c	0	1
00	00	01
01	10	10
11	00	11
10	00	11

f)

Columns Removed from Input cover set to form final cover sets for encoding.

Encoding of columns with nondisjoint codes

$\Pi_G = \{\{C_1, C_5, C_8\}; \{C_2 \quad\}; \{C_6, C_7\}; \{C_3, C_4, C_5, C_8\}\}$ **Nondisjoint Cover**

g)

H

ab \ g1g2	00	01	11	10
00	1	1	1	0
01	0	-	1	0
11	1	0	1	1
10	1	1	0	1

S1 S2 S3 S4

h)

G

cd \ e	0	1
00	C_1	C_2
01	C_4	C_3
11	C_5	C_6
10	C_8	C_7

i)

G

de \ c	0	1
00	S1	S2
01	S4	S4
11	S1 S2 S4	S3
10	S1 S4	S3

j)

G

de \ c	0	1
00	00	01
01	10	10
11	-0	11
10	-0	11

k)

Figure 5.2: Encoding of Disjoint vs. Nondisjoint Cover Sets

The cover set produced(or selected) determines what the column types will be in the sub-function H. Each of the column types in the sub-function H are obtained by performing the union on all columns contained in each of the symbols. Shown in Figure 5.2c, are the different column types in the sub-function H corresponding to the disjoint cover set selected. The actual position of the column types in the Karnaugh map of sub-function H are not known until the codes($g1g2$) have been assigned to each of the symbols.

Shown in Figure 5.2d are the cells of the sub-function G with column labels corresponding to each vector of input variables of the bound set in function F. Shown in Figure 5.2e are the cells of the sub-function G containing the symbols in Π_G that each column in function F is compatible with. Observe that, by making the input cover set disjoint, column C_5 may only be assigned to symbol $S1$(shown circled) because it is no longer an element of the subsets corresponding to symbols $S2$ and $S4$. Similarly, column C_8 may only be assigned to symbol $S1$ because it is no longer an element of the subset corresponding to symbol $S4$.

Finally, codes for each symbol are assigned. Because the column multiplicity is four, then two bits are required to encode each of the four symbols in Π_G. Ideally, the codes should be assigned to simplify both sub-functions simultaneously. However, because the purpose of this section is merely to show the basics of column-based encoding, codes for each symbol are chosen arbitrarily. The codes assigned to each symbol are shown in Figure 5.2c as variables $g1g2$ at the top of each column.

The assignment of codes to columns in the function F is done by assigning the code of each symbol as the code of every column that is an element of that symbol. Shown in the cells of the Karnaugh map in Figure 5.2f are the codes that each column is assigned with. This completes the encoding process for disjoint cover sets. Shown in Figure 5.1c and Figure 5.1d are block diagrams of the original function F and the decomposed set of sub-functions G and H. Note that the codes $g1g2$ assigned to columns correspond to outputs of sub-functions G and inputs to sub-functions H.

5.3.2 Encoding of Nondisjoint Cover Sets

The primary difference between the encoding of disjoint vs. nondisjoint cover sets is that the columns may be assigned with combined symbol codes(supercube of codes) in some instances. For example, if the codes assigned to the symbols are the same as they were in the disjoint case(i.e., S1=00, S2=01, S3=11, and S4=10), then column C_5 could be assigned codes 0- or -0, as well as 00, 01, or 10.

A column can only be assigned a combined symbol code if the supercube of the symbol codes does not contain any codes of symbols which the column is not compatible with. For example, column C_5 can't use the optional code 0- unless it is an element of the two symbols with symbol codes 00 and 01. Similarly, column C_5 can't use the optional code -0 unless it is an element of the two symbols with symbol codes 00 and 10. Also, we can't assign C_5 with the combined symbol codes

of symbols $S2 = 01$ and $S4 = 10$ because they differ by more than one bit. The combined code of symbols $S2$ and $S4$ results in code "- -". This code also includes the code of symbols $S1 = 00$ and $S3 = 11$. Symbol $S3$ is not among the acceptable code assignments for column C_5.

Shown in Figure 5.2g is a nondisjoint cover set selected which allows columns C_5 and C_8 to receive don't cares in their codes. Shown circled in Figure 5.2j, are combinations of symbols that columns C_5 and C_8 were assigned to. Assigning columns C_5 and C_8 to the combined codes of symbols $S1$(i.e., 00) and $S4$(i.e., 10) results in the combined code -0. The resulting column codes are shown in Figure 5.2k. Why not give column C_5 the combined code of symbols S1=00, S2=01, and S4=10? The combined code of symbols $S1$, $S2$, and $S4$ is "- -". This code includes the code of symbol $S3$ which is not among the acceptable code assignments for column C_5. For this reason, column C_5 was removed from symbol $S2$ in the initial cover to form the final cover set shown in Figure 5.2g. This completes the encoding process for the given example of nondisjoint cover sets. The block diagram for the decomposed set of sub-functions G and H is the same as for the disjoint case.

Using the same input function, it was shown how columns can be encoded using disjoint and nondisjoint cover sets. Also, it was shown in this example how don't cares can be introduced into the codes of the predecessor sub-function when an encoding method utilizes the overlap in nondisjoint cover sets. Often times

the added don't cares can greatly reduce the complexity of the predecessor subfunction.

5.4 General Strategy For The New Encoding Approach

Shown in Figure 5.3 is a flow diagram of a general encoding strategy which includes the flow of control in the **DC_ENC** encoding approach presented. Flow diagrams for other encoding approaches are very diverse and are therefore not discussed here. Though there are many different general strategies possible, the general encoding strategy referred to in this chapter is a simple strategy that I recommend for use with the **DC_ENC** encoding approach presented. The primarily purpose of the general encoding strategy described here is to evaluate whether conditions are appropriate for the **DC_ENC** encoding approach to be effective. If so, the **DC_ENC** encoding program would be called. Otherwise, a different encoding program would be called.

The criterion used by the general encoding strategy to determine whether the **DC_ENC** encoding approach is used or not, is based on the values of a Cost Function Ratio(CFR) and a overlap ratio(R_O). The following is a proposed heuristic, IF-THEN-ELSE statement, which determines if the **DC_ENC** encoding approach is used.

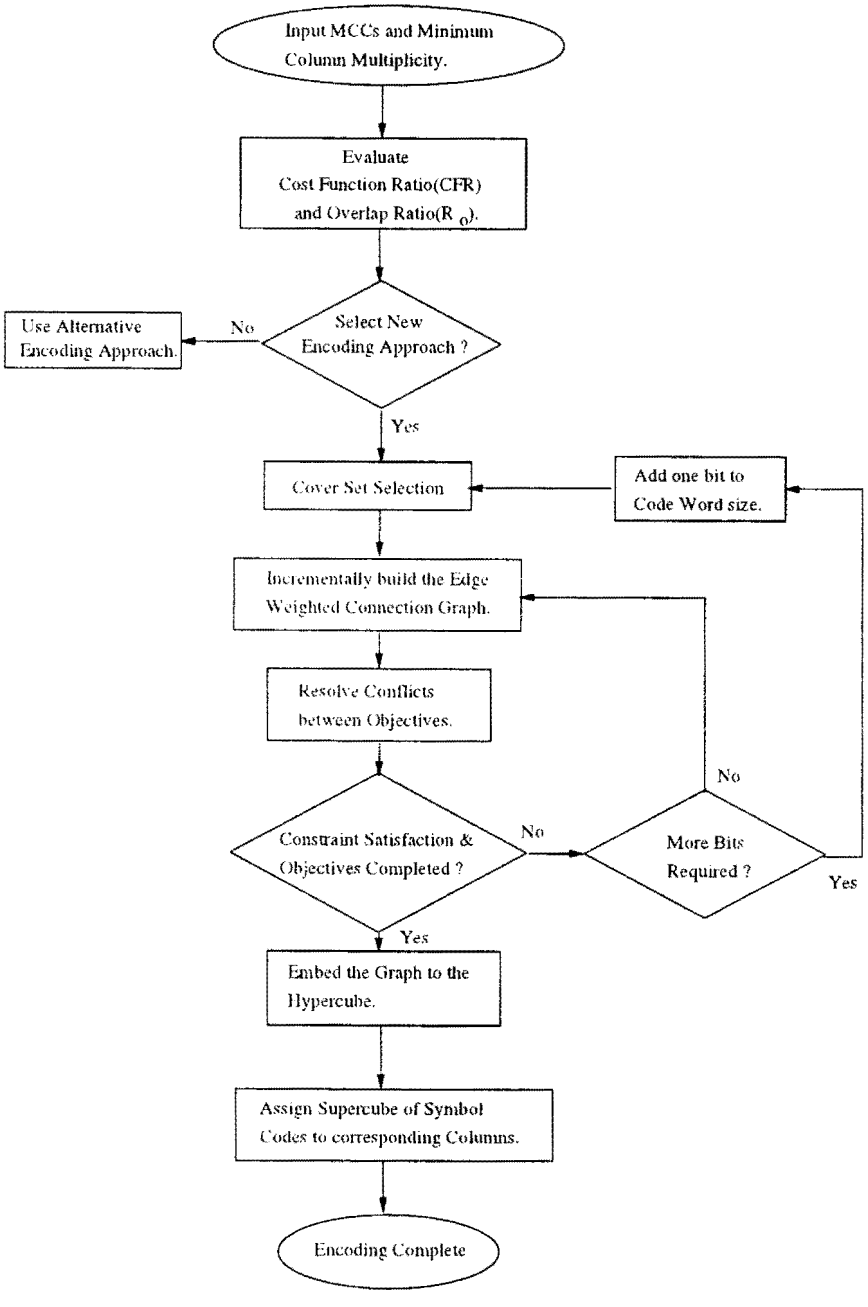


Figure 5.3: Flow Diagram For New Encoding Approach **DC_ENC**

If $(R_O > 1/5$ and $CFR \geq 1/2)$
  Then use the **DC_ENC** encoding approach.
Else
  Use an alternative encoding approach.

For referencing purposes, let the IF-THEN-ELSE statement be referred to as $Rule\#1$. Basically, $Rule\#1$ states two conditions to be satisfied before the **DC_ENC** encoding approach is used. The first condition specifies that there must be sufficient overlap of columns in the symbols of Π_G. Why ? Because if there is no overlap, then the **DC_ENC** encoding approach can't utilize overlap to produce don't cares in the codewords. The second condition specifies that the predecessor sub-function must be roughly equal to or larger than the successor sub-function in terms of inputs and outputs(DFC). Why ? Because the **DC_ENC** encoding approach(**DC_ENC**) tends to simplify the predecessor sub-function more than the successor sub-function. For some functions, **DC_ENC** may actually simplify the predecessor sub-function at the cost of making a more complex successor sub-function(i.e., **DC_ENC** produces don't cares in the G block in such a way that Hamming distances may increase in the H block). However, when the two conditions are satisfied which are necessary before using the **DC_ENC** encoding approach, then the **DC_ENC** approach is selected. Otherwise, an alternative encoding method is selected. When the two conditions above are satisfied, then the **DC_ENC** approach can greatly simplify the complexity of the predecessor block by introducing don't cares in the codes assigned to columns.

One of these conditions requires that there exists sufficient overlap in the cover set before using the **DC_ENC** encoding approach. Unfortunately, due to the differences in each decomposition, it is not known exactly when the overlap is sufficient for the **DC_ENC** encoding approach to be effective. However, based on the experience I acquired solving numerous examples by hand, I recommend that a different encoder be used if $R_O < 1/5$. This cutoff value I arrived at after analyzing example encodings for different functions and their corresponding graphs. Future work may include testing to refine this value, and/or to find other criterion to determine if and when the **DC_ENC** approach is effective.

The cost function ratio (CFR) provides a very rough estimate of the ratio of complexity and/or size of sub-function G relative to sub-function H. Ranges of the CFR ratio are used to make decisions in the general encoding strategy proposed. The following are ranges set for the CFR ratio:

Range 1: $CFR \leq 1/6$
Range 2: $1/6 < CFR \leq 1/2$
Range 3: $1/2 < CFR \leq 2$
Range 4: $2 < CFR \leq 6$
Range 5: $CFR > 6$

These ranges are used for two primary purposes. The first purpose is to determine whether or not one of the two conditions necessary for using the **DC_ENC** encoding approach is satisfied(IF-THEN-ELSE statement above). The second purpose is to determine values to assign to two parameters, X and U, used in the cover

set selection process. The values assigned to X and U are used heuristically to determine how a cover set should be selected in order to provide more efficient encoding for the larger of the two sub-functions in a decomposition. How CFR is used to determine values for these parameters and how the parameters affect the cover set selection process is outlined in Section 5.5.1.

Once the **DC_ENC** encoding approach has been selected, then the next stage in the flow diagram, cover set selection, is performed. Basically, the cover set selection step is a heuristic process of forming an enhanced cover set which is designed to facilitate satisfaction of the encoding objectives in a prioritized manner according to the value of the cost function ration CFR.

The next stage in the flow diagram of Figure 5.3 is the construction of the Edge Weighted Connection Graph($EWCG$). A detailed explanation of this step is given in Section 5.5.2.1. Briefly, the $EWCG$ is constructed in such a way as to obtain a set of weighted constraints which will maximize the number of don't cares produced in the G sub-function, minimize the Hamming distances between cofactors in the H sub-function, and minimize the Hamming distances between the codes assigned to the cells of the Karnaugh map in the G sub-function. When there are conflicts in the process of trying to satisfy certain sets of constraints concurrently, then evaluations are made using a cost function to determine what action should be taken to resolve each conflict. Notice the question "More Bits Required?" in the flow diagram. If less than 75% of all constraints can be satisfied

together in the $EWCG$ once it has been constructed, then an additional code bit is added to the length of codes to be assigned and the process of constructing the $EWCG$ is repeated.

Once the $EWCG$ is constructed and an acceptable percentage of constraints are satisfied in it, then the next stage in the flow diagram is performed which is to map the $EWCG$ to a hypercube of dimension n, where n is the number of bits in the code words to be assigned to each column. A detailed explanation of how the $EWCG$ is mapped to a hypercube is given in Section 5.5.2.2.

Finally, shown in the flow diagram is the assignment of the supercubes of the symbol codes to the columns in the input function. Basically, each column is assigned to the largest combination of 2^k symbol codes that the column is compatible with. For example, if column $C1$ is compatible with codes 00, 01, and 11, then it could be assigned either 00, 01, 11, 0-, or -1. The algorithm would select either column code 0- or -1. However, it could not be assigned the code "- -" because that code includes a symbol code which $C1$ is not compatible with(i.e., code 10). Once these steps in the **DC_ENC** encoding approach have been followed, then the encoding process is finished.

5.5 Detailed Description Of The New Encoding Approach: DC_ENC

5.5.1 PHASE I: Selection Of A Suitable Cover Set(Optional)

The purpose of this section is to outline the proposed heuristic procedure that is used to give higher priority to the encoding of the sub-function which is larger(and/or more complex) than its counterpart sub-function. It is believed that giving a higher priority to the encoding of the larger sub-function will ultimately result in a lower overall DFC of the decomposition, as well as to a reduction in the algorithm computation time. For selecting cover sets from a set of optional cover sets, I recommend different sets of procedures based on the type of encoding program that is used. Presented in Section 5.5.1.1, is the set of procedures I recommend when the **DC_ENC** encoding approach is used. Presented in Section 5.5.1.2, is the set of procedures I recommend when other encoding approaches are used.

5.5.1.1 Cover Set Selection For The New Encoding Approach

This section outlines a set of heuristic procedures used to create cover sets to increase overlap in classes of the cover set so that the **DC_ENC** encoding approach can produce encodings of columns with a greater number of don't cares. Ideally, the input to the cover set selection program would be the minimum column multiplicity and a minimum cover of maximum compatible classes of columns($MCCs$) which form the cover set Π_G. For highly unspecified functions, these $MCCs$ would have

columns which are elements of more than one MCC. Also, it would be desirable to have numerous additional $MCCs$ which are different from those in the cover set Π_G. These additional $MCCs$ can be used as optional classes in a cover set selection process to improve the quality of encoding. More optional classes to select from would increase the number of optional encodings possible.

Based on the range of CFR for a particular decomposition, values are assigned to parameters X and U. The parameter X corresponds to a percentage of highly specified columns in the cover set, and U is the parameter to be used to determine how many extra classes to add to the minimum cover set. The parameter X is used in two different ways. First, it is used to order classes of the minimum cover in descending order according to how many highly specified columns are in each class. Second, it is used to determine which columns should not be included in extra classes that are added. The parameter U is used to determine what fraction of unused codes should be used to increase the overlap in classes, so that additional don't cares may be introduced in the predecessor block. The following are heuristic assignments to parameters X and U based on the ranges established for CFR.

For CFR in Range 2, X = 75 U = .25
For CFR in Range 3, X = 55 U = .5
For CFR in Range 4, X = 35 U = .75
For CFR in Range 5, X = 15 U = 1.0

For CFR in Range 1, the proposed encoding approach is not used because the DFC of the successor sub-function is much larger than the DFC of the predecessor

sub-function. It is not recommended to use the proposed encoding approach when CFR is in Range 1 because the proposed encoding approach tends to produce encodings which simplify the predecessor sub-function more than the successor sub-function.

Different ranges are intended to give different priorities to predecessor and successor blocks according to the ratio of their sizes. Notice from the ranges established, as the value of CFR increases, the value of X assigned decreases and the value of U assigned increases. Likewise, as CFR decreases, the value of X increases and the value of U decreases. In simple terms, what do these changes in X and U correspond to? Larger X values will "tend" to preserve more don't cares in the successor block by not allowing the highly specified columns to be given more than one code. Larger U values will "tend" to introduce more don't cares in the predecessor block by creating more overlap in the cover set. Presented below is the algorithm to select(or form) the final cover set Π_G from the input cover set.

Algorithm 5.5.1

Begin

1. *Find the top $X\%$ of highly specified columns. Columns are considered highly specified if they are among the $X\%$ of columns which have the greatest number of specified output values.*

2. *Order classes of the minimum cover in descending order according to how many highly specified columns are in each class.*

3. *From the minimal cover of symbols(groups of compatible columns), remove from the symbols all the redundant occurrences of the columns found in Step 1.* [1] *Redundant occurrences of columns found in Step 1 should be removed from all symbols except the first symbol(in descending order) which they are elements of. All redundant occurrences of other columns should remain in the symbols.*

4. *Add the number of* $ROUND(U \times number\_of\_unused\_codes)$ *classes to the minimal cover from Step 3 to form the new cover. The function* $ROUND$ *simply returns a value rounded to the nearest integer. Selection of these additional classes (symbols) is performed as follows: From the original minimal cover set, select the symbol(s) which have the greatest number of columns that are compatible with* $2^k - 1$ *symbols(k=1,2,...). Add these new symbols to the cover set resulting from Step 3.*

End.

Adding additional classes to the cover set will, in many instances, allow the encoding program to use additional codes to give combined codes or optional codes

[1] The number of symbols in the minimal cover is equal to the minimum column multiplicity, and the symbols or classes do not necessarily correspond to MCC's.

to certain columns. For example, let there be three classes in a minimal cover set, $S1$, $S2$, and $S3$. Suppose that the codes assigned to $S1$, $S2$, and $S3$ are 00, 01, and 10 respectively. Now suppose an additional class S_4 is added to the minimal cover set and given the unused code 11. Also, let column C_2 be an element of classes S_3 and S_4. Then, there would be 3 codes which could be assigned to column C_2, that is, codes 10, 11, or the combined code 1-. It can be observed that by adding an additional class and assigning to it the unused code 11, the column C_2 was able to receive a combined code which results in a don't care for one of the code bits. For large classes of columns, there may be many columns which may receive don't cares in their code assignments. By assigning don't cares to the codes of columns, the complexity of the predecessor sub-function in a decomposition can be greatly reduced.

5.5.1.2 Cover Set Selection For Other Encoding Approaches

While the **DC_ENC** encoding approach requires cover sets with significant overlap, other encoding approaches may require cover sets with no overlap(disjoint cover sets). For encoders of this type, there may be no benefit from adding additional classes. Selection of suitable cover sets for other encoding approaches is probably done best with specific knowledge of individual encoder's characteristics. However, if specialized selection algorithms are not available, then using a simple algorithm which minimizes Hamming distances may be a good way of

selecting cover sets. Using minimum Hamming distances should lead to simpler sub-functions than if a random selection approach were used.

5.5.2 PHASE II: Primary Encoding Phase

In this section, the essential algorithms and procedures necessary for the new encoding approach(**DC_ENC**) are presented. While the algorithm for cover set selection can greatly enhance encoding results using the **DC_ENC** encoding approach, they are not required for the encoding process(i.e., the primary encoding phase can be used as a stand alone encoding approach whereas the cover set selection algorithm is merely an enhancement to the encoding approach). In the following three sections, the major steps in the **DC_ENC** encoding approach are presented. These major steps are: *(1) construction of the edge-weighted connection graph, (2) embedding the graph to the hypercube, and (3) assignment of the supercube of symbol codes to corresponding columns.*

5.5.2.1 Constructing the Edge-Weighted Connection Graph

What exactly is the edge-weighted connection graph($EWCG$)?

The $EWCG$, as defined here, is an undirected connection graph with nodes corresponding to symbols and edges between any two nodes signifying that there exists at least one column which is compatible with both symbols connected by the edge.

Each edge has an associated weight which is equal to the combined total of sub-weights. These sub-weights are explained in more detail later in this section along with the formulas used to evaluate edge weights and the procedure for constructing the $EWCG$.

What is the purpose of using an $EWCG$?

There are two main reasons why an $EWCG$ is used in this encoding approach. The first reason is to enable multiple types of constraints to be satisfied together in a systematic and controlled manner which might not otherwise be feasible to satisfy in a reasonable amount of time. The other major reason is that once the graph has been constructed following the set of three rules, then the nodes(symbols) and edges in the $EWCG$ can be mapped directly to vertices and edges of a hypercube(i.e., relationships between all nodes and edges in the $EWCG$ will be maintained in the hypercube). Therefore, all nodes(symbols) connected by an edge in the $EWCG$ may be given code words which differ by only one bit.

The classical problem of graph embedding is to embed(or map) the nodes and edges in an edge weighted connection graph to the vertices and edges in a hypercube in such a way that the sum of the edge weights that are embedded to the hypercube is maximized. This is the goal of the heuristics employed in the **DC_ENC** encoding approach to create the $EWCG$. Therefore, the rationale be-

hind the way in which the $EWCG$ is created in the **DC_ENC** encoding approach has two primary objectives. One objective is to create a graph which can be embedded in a hypercube so that all relationships between nodes and edges in the $EWCG$ are maintained in the hypercube(i.e., create a graph which is embeddable). The other objective is to assign the subset of edges to this $EWCG$ which will result in the maximum sum of edge weights that are embedded to the hypercube. This is a new approach. More specifics about comparisons between the new methods presented here and other methods for graph creation and embedding are presented in Section 5.5.2.2. Shown in Figure 5.4 are hypercubes of dimension 1, 2, 3, and 4, and an example of an edge weighted connection graph. From the figure, one can clearly see the similarity between the structure of the $EWCG$ and the hypercubes shown. The $EWCG$ shown is an example of the type of structure created in the graphs by the graph creation algorithm in the **DC_ENC** encoding approach. Also, observe that the $EWCG$ shown in the figure can be embedded in the hypercube of dimension 4 in such a way that all the relationships between nodes and edges in the $EWCG$ can be maintained in the hypercube.

How is the $EWCG$ constructed ?

A general description of how the $EWCG$ graph is constructed is as follows: The graph is constructed by incrementally adding edges between nodes (symbols) in

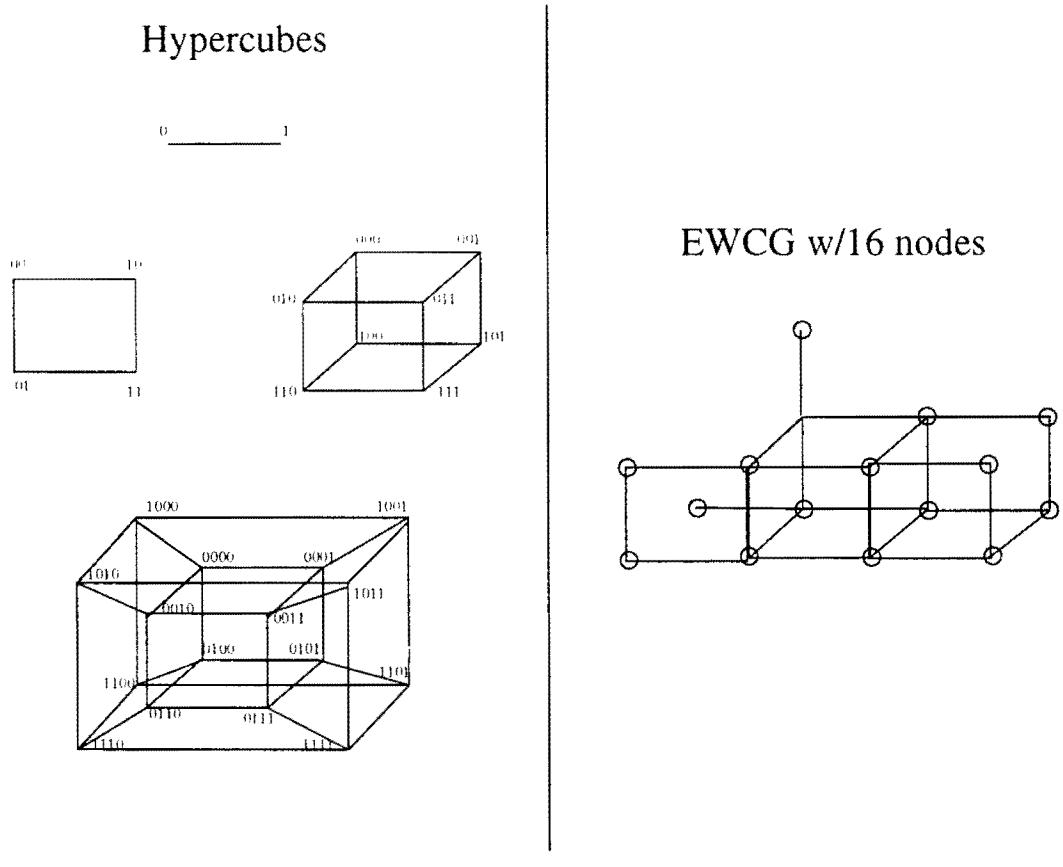


Figure 5.4: Illustration of an $EWCG$ created by the **DC_ENC** encoding approach and its relationship to hypercubes.

the graph, and systematically checking whether any of the specified graph constructing rules are violated. The rules specified for constructing the graph and the procedure for resolving rule violations are explained later in this section. If no rules are violated, an edge with an associated edge weight is assigned between the corresponding two nodes in the graph. If adding an edge between two nodes causes a rule violation, then a procedure is followed which determines what action to take in order to resolve the rule violation. The process of adding edges, checking for rule violations, and resolving rule violations is repeated until all edges have been checked which are in the list of edges to be considered. When this process is completed, then the $EWCG$ has been constructed and is ready to be mapped to a hypercube.

When rule violations are encountered in the process of constructing the $EWCG$, then a cost function($Cost2$) is evaluated. Based on the value of this cost function, a decision is made whether to a), add the new edge(N) under consideration and remove a previously placed edge(P) from the $EWCG$, or b), not to add the new edge to the $EWCG$. The cost function is not called unless a certain rule is violated in the process of constructing the $EWCG$. The cost function is formulated as follows:

$$Cost2 = N - P \tag{5.1}$$

$$N = EW_{jk} + .5W^{u}_{jk} \tag{5.2}$$

$$P = EW_{yz} + .5W^{u}_{yz} \tag{5.3}$$

$$EW_{xy} = f_{xy} + W^{p}_{xy} + 1 \tag{5.4}$$

$$W_{xy} = \sum_{i=4}^{m} f_{xy_i} \times w_i \tag{5.5}$$

$$w_i = FLOOR(Log_2 i) \tag{5.6}$$

Variables defined:

N is a weighted value given to the new edge being considered to be added to the $EWCG$.

P is a weighted value given to one of the edges previously assigned to the $EWCG$ which is being considered for removal from the graph.

m is the size of the largest column constraint(i.e., group of symbols compatible with any column).

EW_{xy} is the edge weight for the edge between symbols S_x and S_y.

W^{p}_{xy} is a weight given to the pair of symbols S_x and S_y corresponding to placed hypercube embedding constraints. Placed hypercube embedding constraints are defined as the hypercube embedding constraints which have already been satisfied in the $EWCG$. Once the $EWCG$ has been constructed, each symbol(node), in a satisfied hypercube embedding constraint will be connected by an edge to at least one other symbol in the constraint. Each pair of sym-

bols connected by an edge will receive a code word which differs by only one bit.

$.5W^u_{xy}$ is a temporary weight given to the pair of symbols S_x and S_y corresponding to unsatisfied hypercube embedding constraints. Unsatisfied hypercube embedding constraints correspond to hypercube embedding constraints which have not been satisfied in the $EWCG$ yet(i.e., some unsatisfied constraints may be satisfied later in the construction of of the $EWCG$).

f_{xy} (frequency) is the number of columns that are compatible with constraints(of size 2 or 3) and which include symbols S_x and S_y.

f_{xy_i} is the number of columns compatible with constraints of size i which contain symbols S_x and S_y.

w_i is a weight associated with constraints of size i and is equal to the number of don't cares that a resulting column code will have if $2^{(w_i)}$ symbols in a constraint of size i are satisfied in a single hypercube embedding constraint. Example: Given the constraint of column C_z, equal to (S_0,S_3,S_5,S_6,S_7) and corresponding symbol codes equal to 000, 001, 011, 010, and 110 respectively. If symbols S_0,S_3,S_5, and S_6 are satisfied in a single face constraint, then column C_z may be given the supercube of four symbol codes(i.e., 0 - -). Observe that the value of w_i is equal to two which is the number of don't cares in the column code.

In the process of constructing the $EWCG$, three sets of edges are considered to be added to the $EWCG$. The union of the three sets is equal to the complete set of edges to be considered to be added to the $EWCG$. Initially, the $EWCG$ consists of as many nodes as there are symbols in the cover set and no edges. Each edge to be considered to be added to the $EWCG$ corresponds to at least one constraint. For example, given three constraints((S_1,S_2), (S_1,S_3,S_4), and (S_1,S_2,S_6)), the edge between symbols S_1 and S_2 corresponds to two constraints(i.e., those constraints containing symbols S_1 and S_2).

The first set of edges to be considered for assignment to the graph is the set $SET1$ of the symbol pairs S_{ij} containing the top 1/3 of symbol pairs that are compatible with the greatest number of columns. The order that edges are considered to be added to the $EWCG$ is in descending order, based on the frequencies $f_{xy_{total}}$ for each pair of symbols which are not assigned yet. Where $f_{xy_{total}}$ is the total number of columns which are compatible with symbols S_x and S_y.

The second set $SET2$ of edges to be considered for assignment to the graph are the edges corresponding to symbol pairs that are contained in groups of symbols of size four or larger(i.e., constraints containing at least four symbols). The order of consideration of these edges is in descending order based on the weights W^u_{ij}.

Hence, the first edge in this set to be considered is the edge with the largest weight W_{ij}^{u}.

The last set $SET3$ of edges to be considered for assignment are all remaining pairs of symbols not considered yet. The order of consideration is in descending order of $f_{xy_{total}}$.

Algorithm for Incrementally Constructing the $EWCG$ Graph

Ideally, it is desired to satisfy all constraints simultaneously without any conflicts between constraints. If a constraint can't be satisfied, then it means that one or more of the symbols in a constraint can't be assigned codes in the same face. If all constraints are able to be satisfied, then every edge to be considered for addition to the $EWCG$ is added without any rule violations(conflicts). However, it is more likely that there are some constraints which can't be satisfied simultaneously. Whenever there is a constraint which can't be satisfied in the $EWCG$, then there is at least one rule violation. The set of graph construction rules and the procedure for resolving rule violations are presented following the algorithm for constructing the $EWCG$. The following is the algorithm for incrementally constructing the $EWCG$:

Algorithm 5.5.2

Begin

Step i) *Form the lists LCC and UHC. LCC is the list of column constraints and UHC is the list of unsatisfied hypercube embedding constraints. Each constraint in LCC is found by simply forming a set of symbolic values corresponding to the symbols in the cover set which a column is compatible with. The list UHC can be formed by adding all combinations of sets of 2^k symbols in each constraint from the list LCC, where k takes on integer values greater than or equal to two. Using the list LCC, calculate the values for f_{xy} and W_{ij}^u. Using the cover set, calculate the value for $f_{xy_{total}}$.*

Step ii) *From the first set($SET1$) of edges to be considered, check the edge S_{xy} having the greatest $f_{xy_{total}}$ for rule violations. If there are rule violations, Go to Step iv. Otherwise continue.*

Step iii) *If there are no rule violations, then assign an edge between symbols(nodes) S_x and S_y in the $EWCG$ with an associated edge weight EW_{xy}. Move the hypercube embedding constraints, which are satisfied as a result of the new edge added, from the list UHC to the list SHC. Also, update the edge weights(EW_{xy}) for symbol pairs which are contained in any of the hypercube embedding constraints which are added to the list SHC. Remove S_{xy} from the first set of edges to be considered for assignment to the graph. Go*

to Step vii.

Step iv) *If there are rule violations, follow the procedure outlined for resolving rule violations. If the rule violations are able to be resolved, then Go to Step vi. Otherwise continue.*

Step v) *If the rule violations were not able to be resolved, then remove edge S_{xy} from the first set of edges to be considered for assignment to the graph. Go to Step vii.*

Step vi) *If the rule violations are able to be resolved, then it means that it was determined by the cost function $Cost2$ that it is worth the cost to remove a previously assigned edge(S_{jk}) and assign the new edge(S_{xy}). Therefore, unassign edge S_{jk} from the $EWCG$. Move any hypercube embedding constraints from the list SHC back to the list UHC which contains both symbols S_j and S_k(this is to unassign previously assigned constraints that are dependent on the previously assigned edge S_{jk}). Update edge weights(EW_{wz}) in the $EWCG$ for any hypercube embedding constraints moved from the list SHC. Now that the rule violation has been resolved by removing the previously assigned edge, complete Step iii as if there had been no rule violations.*

Step vii) *Repeat Step ii until there are no more pairs of symbols S_{xy} in the first set to be considered.*

Step viii) *Repeat Step ii for the second set($SET2$) of edges to be considered for assign-*

ment to the graph.

Step ix) *Repeat Step ii for the third set($SET3$) of edges to be considered for assignment to the graph.*

Step x) *Now that the graph has been constructed, check each of the unsatisfied constraints to see if removing the necessary edge(s) from the graph and adding the unsatisfied constraint is worth the cost (i.e., $Cost2 < 0$). If so, then update the graph accordingly.*

Step xi) *Finally, assign any of the* Optional Constraints *that can be satisfied without removing any edges from the graph.*

"Optional Constraints" are constraints that if satisfied will not yield more don't cares in a particular code word, but will provide an optional encoding for certain columns. This can be used to reduce Hamming distances between the outputs in subfunction G or Hamming distances between columns in the subfunction H.

end.

There are three rules that must be satisfied in the process of constructing the $EWCG$. These rules are necessary to ensure that the $EWCG$ can be embedded in a hypercube in such a way that the connectivity relationships between nodes and edges in the $EWCG$ are maintained in the hypercube embedding. Given below

are the rules that must be satisfied before each new edge is added to the $EWCG$:

Rule 1: No node may have more edges than the number of bits in the code word. Observation: In a hypercube of dimension n, each vertex is connected by edges to exactly n other vertices.

Rule 2: No cycles of odd length. Observation: In a hypercube, there are no cycles of odd length.

Rule 3: Each pair of intersecting faces(face A and face B) of size four(i.e., four symbols) must have exactly two symbols in common. Observation: In a hypercube of dimension $n > 2$, every two intersecting subhypercubes of dimension 2(i.e., hypercubes with four vertices) have exactly two vertices in common.

In the following procedure, steps are presented that decribe what actions to take if one of the three rules are violated as a consequence of adding a new edge to the $EWCG$. If any of the three rules are not satisfied, then at least one edge in the graph($EWCG$) can't be mapped to a hypercube of dimension n where n is the number of bits in the column code words. Therefore, a heuristic cost function $Cost2$ is used to determine whether to add the new edge(N) and remove one or more of the previously placed edges(P) or to make no changes to the graph. The purpose of removing one or more of the previously placed edges is to maintain a

graph which satisfies all three rules.

Procedure for resolving rule violations:

Begin

1. If there is a violation in *Rule* 1, then a determination is made as to whether the value of the new edge is worth the cost of removing a previously placed edge(s).

2. If it is not worth the cost ($Cost2 \leq 0$), then no changes are made and the edge in question is removed from the list of edges yet to be considered.

3. If so, then *Rule* 2 is checked.

4. If *Rule* 2 is violated then a similar determination is made as to whether the removal of another edge(s) is worth the cost of assigning the new edge being considered.

5. If it is not worth the cost ($Cost2 \leq 0$), then no changes are made and the edge in question is removed from the list of edges yet to be considered.

6. If so, then *Rule* 3 is checked.

7. If *Rule* 3 is violated, then no changes are made and the edge in question is removed from the list of edges yet to be considered.

8. If there is not a violation in *Rule* 1, then *Rule* 2 is checked.

9. If *Rule* 2 is violated then a determination is made as to whether the removal of an edge(s) is worth the cost of assigning the new edge being considered.

10. If it is not worth the cost ($Cost2 \leq 0$), then no changes are made and the edge in question is removed from the list of edges yet to be considered.

11. If so, then *Rule* 3 is checked.

12. If *Rule* 3 is violated, then no changes are made and the edge in question is removed from the list of edges yet to be considered.

13. If there is not a violation in *Rule* 1 or *Rule* 2, but there is a violation in *Rule* 3, then a determination is made as to whether the removal of a different edge(s) is worth the cost of assigning the new edge being considered.

14. If it is not worth the cost($Cost2 \leq 0$), then no changes are made and the edge in question is removed from the list of edges yet to be considered.

15. If the cost function determines that the new edge should be added to the graph and one or more of the previously placed edges should be removed, then a final check must be done for any rule violations which have not been checked after the previously placed edges have been removed.

end.

5.5.2.2 Embedding The Graph To The Hypercube

This step simply involves embedding nodes in the graph($EWCG$) to vertices in the hypercube in such a way that nodes connected by an edge in the graph($EWCG$) will be assigned to vertices connected by an edge in the hypercube. This is a classical combinatorial optimization problem called graph embedding. Embedding the graph to the hypercube can be done by any one of several of the known methods for hypercube embedding. One of the known methods for hypercube embedding is implemented in $MUSTANG$[17]. An improved approach named $JEDI$ was later introduced[27]. Yet another approach, $MUSE$[18], produced slightly better results than either $MUSTANG$ or $JEDI$. However, in lieu of the known methods for hypercube embedding, a new algorithm is proposed here for embedding the $EWCG$ to the hypercube. The previously known methods and the new approach presented here have the same basic goal. This goal is to assign a subset of edges from a graph to a hypercube in such a way that the sum of the edges weights assigned in the hypercube is maximized. The algorithm presented here might actually be a more efficient algorithm for embedding the proposed $EWCG$ than the known methods mentioned. Unlike the know methods mentioned, the algorithm presented here makes use of the detailed structure created in the $EWCG$ in order to efficiently embed the $EWCG$ to the hypercube. The methods $MUSTANG$, $JEDI$ and $MUSE$ start with one or more complete graphs, then add edge weights to each of the edges in the graph. Then the graphs are combined together into

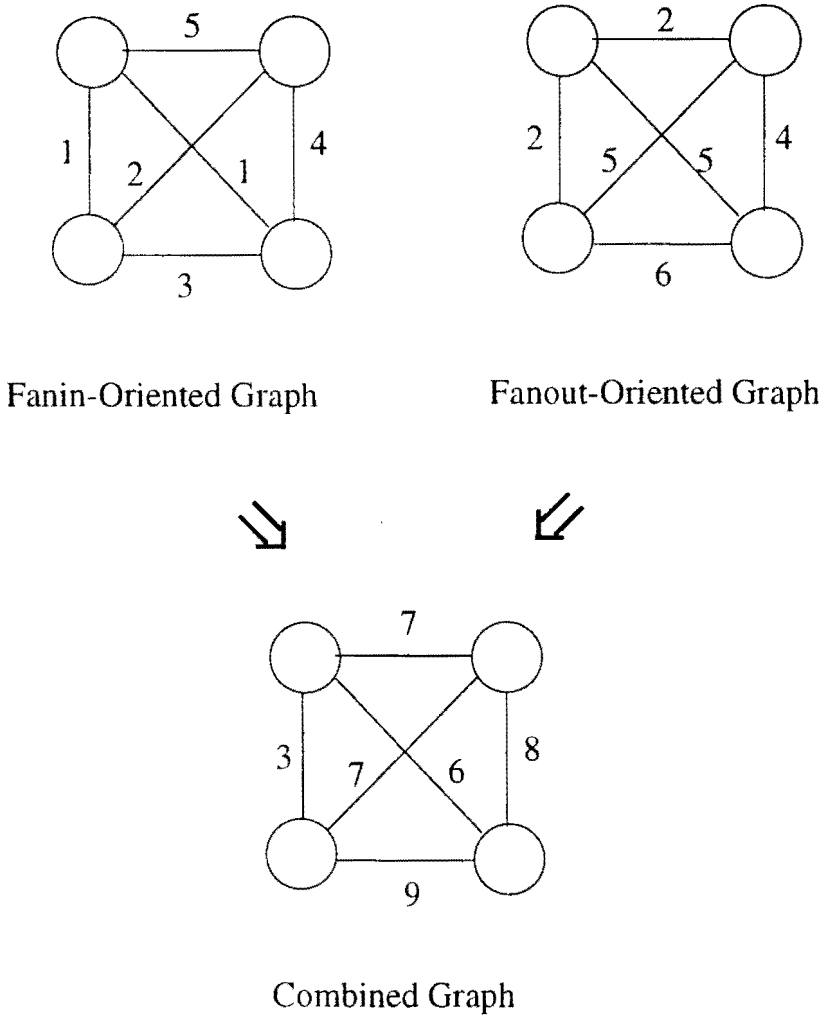


Figure 5.5: Illustration of the graph creation process used in the graph embedding approach of **MUSTANG**.

one complete graph. The weights of each of the edges then becomes the sum of the weights from the corresponding edges in each of the graphs used. This is illustrated in Figure 5.5. Once the edge weights have been assigned to the graph, then a subset of the edges in the graph are embedded directly to a hypercube using the largest edge weights to select which edges to assign to the hypercube. By contrast, in the **DC_ENC** encoding approach, edge weights are only used to create the structure in the graph that can be easily embedded in the hypercube and which will result in the maximum sum of edge weights assigned. The detailed structure created in the $EWCG$ consists of a record of the satisfied hypercube embedding constraints. These hypercube embedding constraints are essentially subhypercubes which are linked together. Once the graph has been constructed using the **DC_ENC** encoding approach, then the graph is embedded directly to a hypercube. Shown in Figure 5.6 is an illustration of the primary difference between typical embedding methods and the embedding method proposed here. Typical embedding methods embed a subset of edges from a complete graph directly to a hypercube whereas the new method proposed here first creates a graph which is easily embedded in a hypercube and then embeds the graph.

Without extensive formal proofs or experimental results, one can only speculate at the effectiveness of the **DC_ENC** encoding approach. Therefore, future work should include testing to evaluate the effectiveness of the graph creation and graph embedding algorithms in the **DC_ENC** encoding approach. Tests should

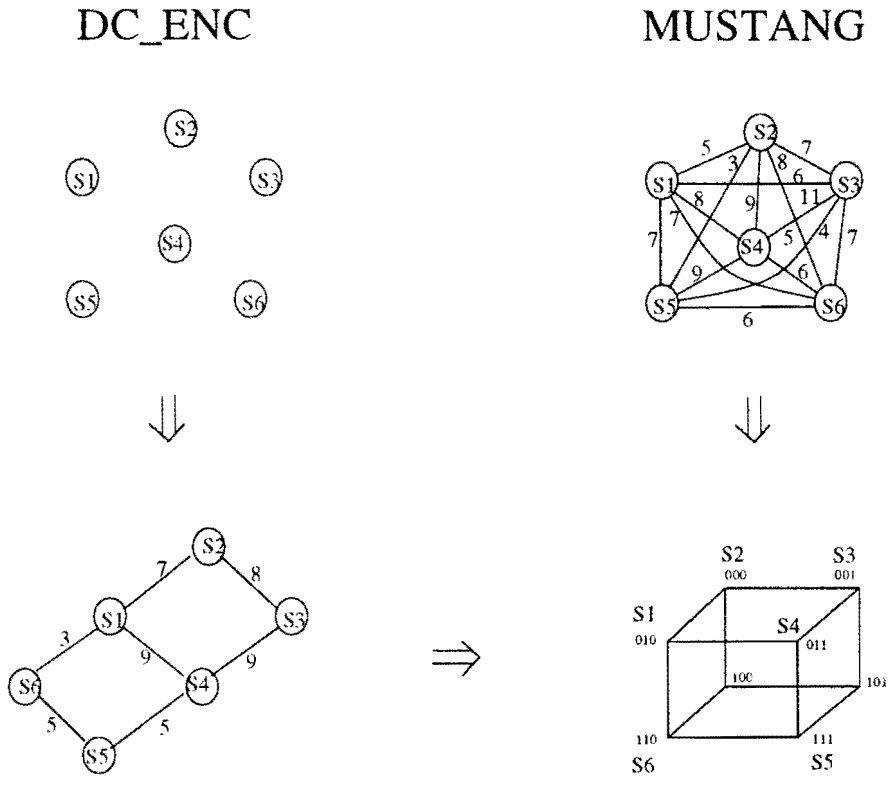


Figure 5.6: Illustration of differences in the direct embedding approach used in **MUSTANG** vs. the approach used in **DC_ENC**.

include comparisons of these algorithms for use in two-level and multi-level state assignment. Also, tests should include comparisons of these algorithms for use in two-level and multi-level decompositions of switching functions. The algorithm for embedding the $EWCG$ to the hypercube is as follows:

Algorithm 5.5.3

Begin

1) *Start with the largest satisfied hypercube embedding constraint(satisfied constraint containing 2^k symbols) and assign the symbols to 2^k codes such that the supercube of these codes does not contain any of the codes of symbols not in the constraint.*

2) *Find the largest satisfied hypercube embedding constraint which has the largest intersection with the constraint assigned in Step 1. Assign codes to symbols in this constraint which are not already assigned, such that the supercube of these codes does not contain any of the codes of symbols not in the constraint.*

3) *Find the largest satisfied hypercube embedding constraint which has the largest intersection with more than one of the previously assigned constraints(if one exists). Assign codes to symbols in the constraintas done in Step 2.*

4) *Repeat Step 3 until no more constraints intersect more than one of the previously placed(assigned) constraints.*

5) *Repeat Step 2 until all constraints which intersect with the constraint assigned in Step 1 have been placed.*

6) *If there exist any constraints which have not been assigned yet, then repeat Step 1 for the largest of the remaining constraints.*

7) *If there exist any nodes(symbols) in the graph that do not have any edges, then they are assigned to any of the remaining codes.*

end.

A symbol in the graph will not have any edges is one if none of the constraints that contain the symbol were satisfied or if all columns in the symbol are incompatible with all other symbols. For symbols in the graph which do not have any edges, use of Hamming distances can greatly improve the selection from among the remaining codes.

Two subproblems which are very similar in state assignment and encoding for Curtis decompositions are: (1) the creation of an edge-weighted connection graph and (2) embedding the the edge-weighted connection graph to the hypercube. However, there are important distinctions in the approaches used to solve these problems. Basically, these differences in the approaches used for state assignment and encoding for Curtis decompositions are as follows: In state assignment, weights are assigned incrementally to edges in a complete graph, followed by embedding

the nodes and edges in the graph to a hypercube. The embedding of nodes and edges to the hypercube is done based on edge weights. This is contrasted with the encoding approach **DC_ENC** which starts with a graph containing nodes and no edges, and incrementally assigns edges and edge weights to the graph. After the graph has been constructed, then the graph is embedded in the hypercube using the structure, created in the graph, to guide the embedding process. In the **DC_ENC** approach, edge weights are not used in the graph embedding process.

More specific differences will become clear after a more complete description is given for the two-step approach used commonly for state assignment. The two-step approach used for state assignment is followed by the corresponding two-step approach used in the **DC_ENC** encoding approach. In state assignment, creation of the edge-weighted connection graph and embedding the graph to the hypercube is typically done in a two step process as follows:

1. Creating the edge-weighted connection graph: Start with a complete graph, or nearly complete graph(i.e., graph with edges between each node and every other node), having as many nodes as there are symbols. Incrementally add weights to edges for each of the encoding constraints. For specifics on how the edge weights are calculated, refer to the algorithms in [17][27][18].

2. Next, embed the edge-weighted connection graph to a hypercube of dimension n, where n is the number of bits to be assigned in the symbol codes: This is done by assigning nodes and edges in the graph to vertices and edges

in the hypercube. There are two primary formulations of the embedding problem for state assignment. These are: (1) Assign all edges in the graph to edges in the hypercube, maintaining the same connectivity relationships that exist between nodes and edges in the graph. If this is not possible for the given n dimensional hypercube, then increase the size of the dimension by one. Increasing the size of the dimension by one means doubling the number of vertices in the hypercube. This also means that the symbol codes will increase in length by one bit. Continue increasing the dimension of the hypercube until all the edges(or a very high percentage) can be embedded in the hypercube. (2) Instead of attempting to assign all the edges in the hypercube without limiting the number of bits in the symbol codes, limit the number of encoding bits to some maximum number. Then assign as many edges as possible to the hypercube in such a way that the sum of the edge weights assigned to the hypercube is maximized.

In this second formulation of the embedding problem, there are two reasons why only a subset of the edges can be assigned to the hypercube. One reason is that each vertice in a hypercube is limited to the number of other vertices that it is connected to. This number is equal to the dimension of the hypercube(i.e., for an "n" dimensional hypercube, each vertice is connected by an edge to exactly "n" other vertices). For example, the dimension of

a given hypercube may be eight, but the graph may have some nodes with many more edges than eight. In such cases, choices must be made to add some edges and to discard others. The second reason that some edges in the graph can't be assigned in the hypercube is because there may be conflicts as to what nodes should be adjacent to what other nodes(i.e., if nodes connected by an edges in the graph are not assigned to adjacent vertices in the hypercube, then the edges in the graph can't be assigned in the hypercube).

The following is the two step process in the **DC_ENC** encoding approach for creating the edge-weighted connection graph and embedding the graph to a hypercube as presented in Algorithms 5.5.2 and 5.5.3:

1. Creating the edge-weighted connection graph: Start with a graph having as many nodes as there are symbols and no edges. Incrementally add edges between nodes along with corresponding edges weights following a set of three graph construction rules. Only a subset of the edges in a complete graph are considered. The edges considered to add to the graph are those which connect a pair of symbols that are both elements(i.e., together) of at least one column constraint. The graph construction rules are specifically designed to maintain structure in the graph which can be easily embedded in the hypercube. Basically, the graph construction rules restrict assignments of edges in the graph to a set of edges which conform to subgraphs of the hypercube. In the process of constructing the graph, a record is kept of the number of

constraints which could not be satisfied as a result of the restrictions imposed by the graph construction rules. If a large percentage(75%) of the constraints were not able to be satisfied for the given number of encoding bits specified(i.e., the number of encoding bits determines how many edges that are connected to each vertice in a hypercube), then increase the number of encoding bits by one. Increasing the number of encoding bits by one, allows an additional edge to be connected to each of the nodes in the graph. If additional bits were added, then repeat the graph construction process. If the graph construction process is repeated, then the graph begins again with all the nodes but with no edges.

2. Next, embed the edge-weighted connection graph to a hypercube: The goal is to assign nodes and edges in such a way that every edge connected between two nodes in the graph will be asssigned between two vertices in the hypercube. Once the graph has been constructed, begin embedding specific subgraphs(i.e., subhypercubes that correspond to hypercube embedding constraints assigned to the graph) of the edge-weighted connection graph to sets of vertices in the hypercube. This is done in such a way that the connectivity relationships in the graph are preserved in the hypercube. To begin this process, the largest subgraph corresponding to the largest hypercube embedding constraint is embedded first. Basically, after the first subgraph has been embedded, then the remaining subgraphs are embedded one by one

in descending order according to the greatest number of nodes that each subgraph has in common with the previously embedded subgraphs. This process is repeated until all subgraphs have been embedded in the hypercube.

By analyzing each of these two-step approaches, one can note the following differences: In state assignment, construction of the edge-weighted graph is reduced to a problem of assigning weights to edges in a complete or nearly complete graph. By contrast, in the **DC_ENC** approach, the construction of the edge-weighted graph begins with a graph with no edges. The addition of edges to the graph follows a strict set of graph construction rules. The rules are designed to restrict the graph to a structure that can be easily embedded(or folded) into the hypercube.

Also, observe that very different schemes are used for embedding the graph to the hypercube. In the state assignment approaches, the problem is reduced to embedding as many edges as possible in the hypercube such that the sum of the edge weights assigned to the hypercube is maximized. This is contrasted with the scheme used in the **DC_ENC** approach which does not use edge weights in the embedding process(i.e., edge weights are only used to create the structure in the graph). The problem of embedding in the **DC_ENC** approach is reduced to simply "folding" the specially created graph structure into the hypercube. The specially created graph structure can be thought of as various sizes of subhypercubes linked together.

5.5.2.3 Assignment Of Supercube Of Symbol Codes To Corresponding Columns

For many of the columns, the task of assigning codes to them is trivial once the symbol codes have been established. However, certain columns can't be assigned to 2^k symbol codes (where k is an integer) or certain combinations of 2^k symbol codes are not allowed. Basically, a trivial encoding is when there is only one obvious code to choose from. Encoding of columns which are not considered trivial are considered non-trivial. Encoding of columns which are considered non-trivial is slightly more complicated than encoding the columns that are considered trivial.

There are two conditions that must be checked for each column in order to determine if the encoding of a column will be trivial or not. The following are the two conditions required for trivial encodings:

1) The number of symbols that a column is compatible with is equal to 2^k(where k is an integer). Stated another way, the number of symbols in a column constraint is 2^k.

2) The supercube of 2^k symbol codes, which column C_i is compatible with, does not include any of the codes not in the columns' constraint.

If both of these conditions are satisfied, then the column is considered to be a trivial case. If either condition is not satified then the column is treated as a non-trivial case. The following is the algorithm for assigning codes to columns.

Algorithm 5.5.4

Begin

1. *For each column to be encoded, determine if the encoding of the column is considered trivial or non-trivial.*

2. *For each of the trivial cases, encode the column with the supercube of the symbol codes in the columns constraint.*

3. *For non-trivial cases, the assignment of codes to columns is as follows: Find all combinations of* 2^k *symbols which column* C_i *is compatible with(where* $2^k \leq m < 2^{k+1}$ *and* m *is the total number of symbols that* C_i *is compatible with). Store the supercube of each combination in a temporary assignment list* TAL*. Remove any codes from this list which violate a hypercube embedding constraint. Then from the remaining candidate codes determine which code results in the smallest Hamming distances between adjacent codes in subfunction* G *using the cost function* $Cost3$*. Assign the code found to the column under consideration.*

End.

For the trivial cases, the supercube of 2^k symbol codes are assigned as the column's code word.

Example:

Given the column constraint for column $C_i = (S_3, S_4, S_6, S_9)$ and the following symbol codes.

100 S3
110 S4
101 S6
111 S9
---
1 - - = Supercupe of the codes in the constraint.

The supercube can be assigned as the column code.

Examples for non-trivial cases:

Given is the column constraint for column $C_i = (S_1, S_2, S_3, S_5, S_7)$ and the corresponding symbol codes for all symbols in a cover set. The column constraint and the following symbol codes are used in the next two examples.

000 S1 100 S5
001 S2 101 S6
010 S3 011 S7
110 S4 111 S8

Example 1:

000 S1
001 S2
010 S3
100 S5
—
- - - = Supercupe of the codes in the constraint.

This is an example of an encoding that is not allowed. The supercube of the codes contains codes of symbols not in the constraint of column C_i(i.e., 101,110, and 111). Therefore the supercube can't be assigned as the code of column C_i.

Example 2:

000 S1
001 S2
010 S3
011 S7
—
0 - - = Supercupe of the codes in the constraint.

This is an example of an encoding that is allowed. The supercube of the codes contains only codes of symbols in the constraint of column C_i. Therefore the supercube can be assigned as the code of column C_i.

When a non-trivial case has more than one supercube code to choose from, then a cost function is called to determine which of the optional codes to select. This is done using the cost function $Cost3$ which determines the sum of the Hamming distances between all codes in subfunction G which are adjacent to the code of column C_i. The candidate code with the lowest cost is assigned as the column code. The cost function is as follows:

$$Cost3 = \sum_{j=1}^{n} HAM\_DIST(CODE(C_i), CODE(C_j))$$

where C_i is the column which is being considered for an encoding and C_j is a column which has its corresponding code in G adjacent to the code of C_i. The value n is the total number of codes in G which are adjacent to the code of C_i. $CODE(C_i)$ and $CODE(C_j)$ are the corresponding codes of the two adjacent columns.

The HAM_DIST function returns the number of bits in two code words which are not compatible (i.e., a 0 in one code word where there is a 1 in the other code word—don't cares are considered compatible with any bit value).

5.6 Encoding Example Solved Step-by-Step

Shown in Figure 5.7 is the function $F5$ which is being decomposed in this example. Refer back to algorithms and procedures in Section 5.5.2.1 thru Section 5.5.2.3 as needed for further clarification of the encoding process. In order to avoid

F

ab \ cdef	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	0	1	0	0	-	0	0	1	1	0	-	-	0	0	1	-
01	-	-	-	0	-	1	0	-	-	0	0	-	1	-	-	0
11	0	1	-	0	1	0	-	-	1	1	-	-	1	-	-	1
10	-	-	-	0	-	-	0	-	1	-	0	0	-	-	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	S2 S3	S5 S6	S1 S2 S3 S4	S3	S1 S4 S5 S6	S2	S3 S4	S5 S6	S6	S4	S3 S4 S5	S1 S2 S3 S4 S5	S1	S1 S2 S3 S4	S5	S4 S5

a)

$$\Pi_G = \{\overset{S1}{\{2,4,11,12,13\}},\overset{S2}{\{0,2,5,11,13\}},\overset{S3}{\{0,2,3,6,10,11,13\}},\overset{S4}{\{2,4,6,9,10,11,13,15\}},\overset{S5}{\{1,4,7,10,11,14,15\}},\overset{S6}{\{1,4,7,8\}}\}.$$

b)

H

ab \ xyz							
0	0	0	0	1	1	-	-
1	1	0	0	0	-	-	-
1	0	0	1	1	1	-	-
0	0	0	0	0	1	-	-
S1	S2	S3	S4	S5	S6		

c)

a) Function F shown with corresponding symbols each column is compatible with.

b) Input cover set showing columns which are elements of each symbol.

c) Subfunction H showing the column types obtained from the union of all columns within each symbol.

Figure 5.7: Function $F5$ for Encoding Example

confusion, it should be noted that the steps shown in this example do not correspond to the steps in algorithms and procedures because the steps in algorithms and procedures are not executed in sequential order. Rather, steps shown in the example correspond to major phases in the encoding process and not to steps in algorithms and procedures.

5.6.1 Constructing the Edge-Weighted Connection Graph

Execution of Algorithm 5.5.2 on Function $F5$:

Step 1) In this step, the lists LCC and UHC are generated and values of parameters used in the algorithms and procedures are calculated. The parameters calculated are f_{xy}, $f_{xy_{total}}$, and W_{ij}^{u}. It can be observed, that generating these lists and parameter values can be done simply by applying the corresponding definitions and formulas. Because these lists and parameters are trivial to generate, only the resulting lists and formulas are shown here(Figure 5.8).

Step 2) In this step, the first set(SET1) of edges is considered to be added to $EWCG$ using the algorithms and procedures in Section 5.5.2.1. The first set of edges are the symbol pairs shown in Figure 5.8c marked with asterisks. Begin assigning edges to the graph starting with the edge corresponding to the symbol pair denoted $S_{ij}(S_i$ and $S_j)$ that has the greatest frequency $f_{ij_{total}}$. $f_{ij_{total}}$ is the

a)

Constraint Size	Column Constraints(sets of symbol numbers) in list LCC.
5	C_{11}-(1,2,3,4,5)
4	C_2-(1,2,3,4) C_{13}-(1,2,3,4) C_4-(1,4,5,6)
3	C_{10}-(3,4,5)
2	C_{15}-(4,5) C_6-(3,4) C_7-(5,6) C_1-(5,6) C_0-(2,3)
1	C_3-3 C_5-2 C_8-6 C_9-4 C_{12}-1 C_{14}-5

b)

Constraint Size	Hypercube Embedding Constraints in list UHEC.
5	(1,2,3,4)(1,2,3,5)(1,3,4,5)(2,3,4,5)(1,2,4,5)
4	(1,2,3,4)(1,2,3,4)(1,4,5,6)
3	Redundant constraints may be eliminated.
2	There are no hypercube embedding constraints generated from constraints smaller than size 4.
1	

c)

Pairs of Symbols	$f_{ij_{total}}$	W^u_{ij}	f_{ij}
S_{12}	3	6	0
S_{13}	3	6	0
S_{14}	* 4	8	0
S_{15}	2	4	0
S_{16}	1	2	0
S_{23}	* 4	6	1
S_{24}	3	6	0
S_{25}	1	2	0
S_{26}	0	0	0
S_{34}	* 5	6	2
S_{35}	2	2	1
S_{36}	0	0	0
S_{45}	* 4	4	2
S_{46}	1	2	0
S_{56}	3	2	2

a) Table showing all columns and the corresponding sets of symbol indexes they are compatible with. b) Table showing the hypercube embedding constraints generated from each size of constraint from the list LCC. c) Table showing the total number of columns($f_{ij_{total}}$) compatible to each symbol pair(S_{ij}), the corresponding weight W^u_{ij} (maximum number of don't cares that can be introduced from constraints(greater than size 3) which contain symbols S_i and S_j, and the number f_{ij} of column constraints(less than size 4) which contain symbols S_i and S_j.

Figure 5.8: Tables showing the primary lists and parameters used in the Example

total number of columns which are compatible with symbols S_i and S_j. Initially the graph appears as in Figure 5.9a.

Therefore the first edge to be considered is S_{34}. Since there are no other edges in the graph, no rules are violated and the edge and it's corresponding edge weight are assigned to the graph. Using the formula for calculating edge weights, the edge weight for edge S_{34} is calculated as follows:

$$EW_{34} = f_{34} + W_{34}^{p} + 1 = 2 + 0 + 1 = 3$$

$W_{34}^{p} = 0$ because no constraints of size four or greater(i.e., hypercube embedding constraints), which contain symbols S_3 and S_4, have been satisfied. The graph now appears as in Figure 5.9b.

The next edge assigned is the one with the next largest value of $f_{ij_{total}}$. It turns out that there is a three-way tie between edges S_{14}, S_{23}, and S_{45} for the next largest frequency. Use the following criterion for breaking ties:

For each $f_{ij_{total}}$(corresponding to S_{ij}) that are tied(equal), find the symbol $S_k \in \{S_{ij}\}$ which has the most edges. Then compare these values and select the pair which has the lowest number of edges from these(i.e., select the least from the greatest). If there is still a tie, select randomly.

In this case there is still a tie so S_{14} is chosen randomly and checked for rule violations. Since there are no violations, it is assigned to the graph.

$$EW_{14} = f_{14} + W_{14}^{p} + 1 = 0 + 0 + 1 = 1$$

Similarly for the next two edges which had the next largest $f_{ij_{total}}$. No rules were violated so the edges S_{23} and S_{45} are assigned to the graph.

$$EW_{23} = f_{23} + W_{23}^{p} + 1 = 1 + 0 + 1 = 2$$

$$EW_{45} = f_{45} + W_{45}^{p} + 1 = 2 + 0 + 1 = 3$$

This concludes the assignment of the first set of edges. The graph now appears as in Figure 5.9c.

Step 3)

In this step, the second set($SET2$) of edges is considered to be added $EWCG$. The second set of edges to be considered can be found under the column heading W_{ij} in Figure 5.8c which are not equal to zero. The second set of edges are assigned in descending order according to the values of W_{ij}^{u}. An example of how one of these

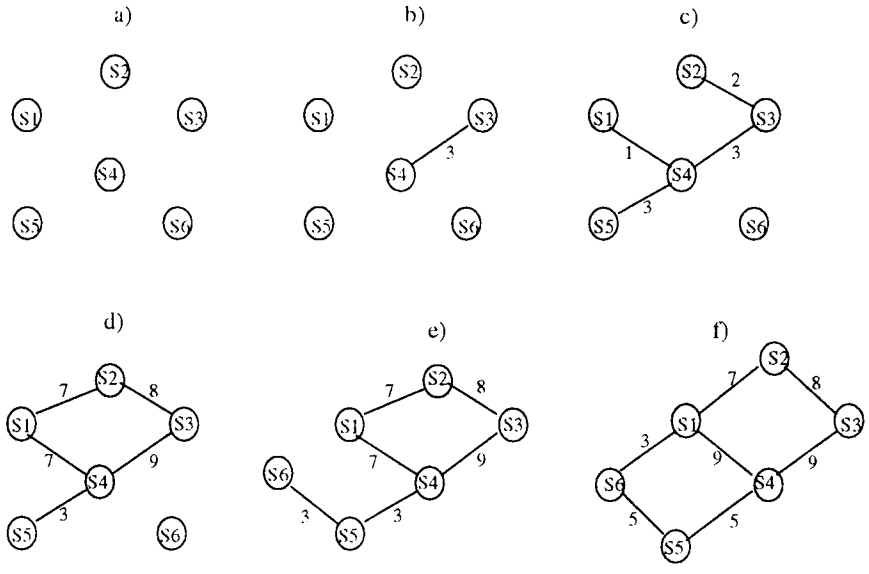


Figure 5.9: Construction of the Edge Weighted Connection Graph.

weights is calculated is as follows:

$$W_{xy} = \sum_{i=4}^{m} f_{xy_i} \times w_i \tag{5.7}$$

$$W_{14}^{u} = \sum_{i=4}^{5} f_{14_i} \times w_i = (f_{14_4} \times w_4) + (f_{14_5} \times w_5) = (3 \times 2) + (1 \times 2) = 8 \tag{5.8}$$

Recall from the definition, that f_{xy_i} is the number of columns compatible with groups of symbols(constraints) of size i which contain symbols S_x and S_y

The first edge to be considered is either S_{23} or S_{14} since they tie for the greatest

W_{ij}^{u}. However, since both have already been placed, simply remove S_{23} and S_{14} from the second set of edges to be considered. Next, find the next largest W_{ij}^{u}.

There is a three way tie for the next edge to be considered(S_{12},S_{13} and S_{24}). Use the same criterion for breaking ties as done for the first set of edges. The tie is broken and S_{12} is selected next. The following is the tie breaking criterion used:

Max edges from S_{12} is $S_1 = S_2 = 1$.

Max edges from S_{13} is $S_3 = 2$.

Max edges from S_{24} is $S_4 = 3$

MIN(MAX(S_1, S_2),MAX(S_1, S_3),MAX(S_2, S_4)) = MIN(1,2,3) = 1.

Because there are no rule violations for edge S_{12}, it is assigned to the $EWCG$ along with its corresponding edge weight. This satisfies the first group constraint(or hypercube embedding constraint) of size 4 where the set of symbols in the group constraint is (1,2,3,4). Once edge weights have been updated, the graph will appear as in Figure 5.9d. Note that 3 columns are compatible with the group of symbols(1,2,3,4). For each edge in the group constraint, an additional weight of 2(for 2 don't cares) is added to each edge weight for each column that is compatible with the group constraint(i.e., $W_{12}^{p} = 3 \text{ x } 2 = 6$). Using the formula for assigning edge weights, the edges are updated as follows:

$$EW_{12} = f_{12} + W_{12}^{p} + 1 = 0 + 6 + 1 = 7 \tag{5.9}$$

W_{12}^{p} is calculated as follows:

$$W_{12}^{p} = \sum_{i=4}^{5} f_{12_i} \times w_i = (f_{12_4} \times w_4) + (f_{12_5} \times w_5) = (2 \times 2) + (1 \times 2) = 6 \tag{5.10}$$

Similarly, the other edges in the group constraint are updated.

$$EW_{14} = f_{14} + W_{14}^{p} + 1 = 0 + 6 + 1 = 7 \tag{5.11}$$

$$EW_{23} = f_{23} + W_{23}^{p} + 1 = 1 + 6 + 1 = 8 \tag{5.12}$$

$$EW_{34} = f_{34} + W_{34}^{p} + 1 = 2 + 6 + 1 = 9 \tag{5.13}$$

The next edge selected is S_{13}. *Rule* 1 is not violated, but *Rule* 2 is violated, because a cycle of odd length is identified (S_{12}, S_{23}, S_{13}). Therefore the cost function $Cost2$ must be evaluated to determine whether adding the new edge(N) to the graph and removing one of the other edges in the cycle is worth the cost. Calculate the weights of the new edge as though the new edge had already been added to the graph(i.e., the value of EW_{13} should not be zero). The following are

two evaluations of $Cost2$ if either edge S_{12} or edge S_{23} are to be removed:

For S_{12}, $Cost2 = N - P = (EW_{13} + .5W^u_{13}) - (EW_{12} + .5W^u_{12}) = (1 + .5(6))$ - $(7 + .5(0)) = 4 - 7 = -3$

Because Cost2 ≤ 0, this change is not worth the cost and edge S_{12} remains in the graph.

For S_{23}, $Cost2 = N - P = (EW_{13} + .5W^u_{13}) - (EW_{23} + .5W^u_{23})$
$= (1 + .5(6)) - (8 + .5(0)) = 4 - 8 = -4$

Again, because Cost2 ≤ 0, this change is not worth the cost. Therefore no changes are made and edge S_{13} is removed from the list of edges to be considered.

The next edge selected is S_{24}. Again *Rule* 1 is not violated, but *Rule* 2 is violated, because a cycle of odd length is identified (S_{12}, S_{24}, S_{14}). The cost function $Cost2$ is evaluated again for each edge that can break the cycle.

For S_{12}, $Cost2 = 4 - 7 = -3$

Since $Cost2 \leq 0$, this change is not worth the cost.

For S_{14}, $Cost2$ = 4 - 8 = -4

Since Cost2 $\leq$ 0, this change is not worth the cost. Therefore no changes are made and edge S_{24} is removed from the list to be considered.

The next edge considered is S_{15}. Again, a cycle of odd length is identified (S_{15}, S_{45}, S_{14}).

For S_{14}, Cost2 = 3 - 8 = -5

For S_{45}, Cost2 = 3 - 5 = -2

Since Cost2 $\leq$ 0, no changes are made and edge S_{15} is removed from the list to be considered.

For the next largest W_{ij}^{u} there is a five-way tie between edges S_{16}, S_{25}, S_{35}, S_{46}, and S_{56}. Using the procedure outlined for breaking ties, the following results are obtained:

Max edges from S_1 or S_6 = 2

Max edges from S_2 or S_5 = 2

Max edges from S_3 or S_5 = 2

Max edges from S_4 or S_6 = 3

Max edges from S_5 or S_6 = 1

MIN(MAX(1,6),MAX(2,5),MAX(3,5),MAX(4,6),MAX(5,6))

= MIN(2,2,2,3,1) = 1.

Therefore S_{56} is the next edge to be considered. No rules are violated so edge S_{56} is added to the graph. The corresponding edge weight is $EW_{56} = 3$. The graph now appears as in Figure 5.9e.

Next, there is a four-way tie. Randomly, S_{25} was selected from those above which had 2 as the maximum number of edges connected to one of its symbols. *Rule* 1 and *Rule* 2 are satisfied, but *Rule* 3 is violated because this edge created a new face of size 4 which intersects a different face with more than 2 nodes (symbols) in common. Select edges to be evaluated with S_{25} which are not in the same face(i.e., S_{12} and S_{14} are edges in face($S_{12}, S_{23}, S_{34}, S_{14}$) but not in face ($S_{25}, S_{45}, S_{34}, S_{23}$)). If $Cost2 > 0$, make changes to the graph accordingly(add the new edge and remove the other).

For S_{12}, $Cost2$ = 3 - 7 = -4

For S_{14}, $Cost2$ = 3 - 8 = -5

Since Cost2 $\not> 0$, no changes are made and edge S_{25} is removed from the list to be considered.

The next edge considered is S_{35}. Rule 2 is violated because a cycle of odd length is identified(S_{35}, S_{45}, S_{34}).

For S_{34}, Cost2 = 2 - 9 = -7

For S_{45}, Cost2 = 2 - 4 = -2

Since Cost2 $\leq$ 0, no changes are made and edge S_{35} is removed from the list to be considered.

The next edge considered is S_{16}. There are no rule violations therefore S_{16} is assigned to the graph with its corresponding edge weight. Update the edge weights of those pairs which are elements of the group. Edges updated are:

$$EW_{16} = f_{16} + W_{16}^{p} + 1 = 0 + 2 + 1 = 3$$

$$EW_{56} = f_{56} + W_{56}^{p} + 1 = 2 + 2 + 1 = 5$$

$$EW_{45} = f_{45} + W_{45}^{p} + 1 = 2 + 2 + 1 = 5$$

$$EW_{14} = f_{14} + W_{14}^{p} + 1 = 0 + 8 + 1 = 9$$

The graph now appears as in Figure 5.9f.

This completes the second set of edges to be considered for assignment to the graph. As it turns out, this is the last set of edges for consideration because

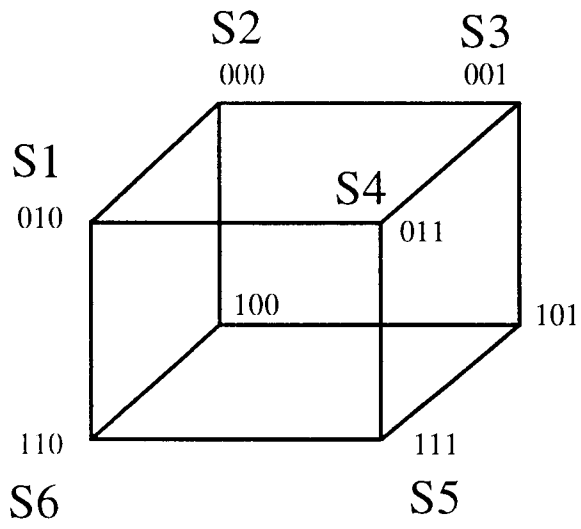


Figure 5.10: Assignment of codes resulting from the mapping of the Edge-Weighted Connection Graph to the Hypercube.

there are no remaining edges which have not been considered yet. Therefore this completes the construction of the $EWCG$ graph.

5.6.2 Embedding The Graph To The Hypercube

The next step is to embed the graph directly to a hypercube. The following mapping of symbols to codes was found using the procedure outlined in Algorithm 5.5.3 in Section 5.5.2.2

First, select the largest group(hypercube embedding constraint) of 2^k symbols and assign it to 2^k codes such that the supercube does not contain any of the codes not in the constraint. This can be ensured by maintaining the connectivity relationship in the hypercube that is found in the graph($EWCG$). It turns out that

there is a tie for the size of the largest constraint. The two groups in the graph which are largest are (S_1, S_2, S_3, S_4) and (S_1, S_4, S_5, S_6). Select the group which is compatible with the greatest number of columns. Therefore group (S_1, S_2, S_3, S_4) is selected. Arbitrarily four codes which differ only in one bit are chosen for the constraint

000, 001, 011, and 010.

Now the supercube of these codes is calculated. This results in the supercube "0- -" which does not contain any codes not in the constraint.

These codes are assigned so that the connectivity relationship in the graph is maintained in the hypercube.

000 S_2
001 S_3
011 S_4
010 S_1

Next find the largest group (constraint) of 2^k symbols which has the largest intersection with the group already assigned. Assign this group. The group found is (S_1, S_4, S_5, S_6). Since S_1 and S_4 are already assigned, only S_5 and S_6 need to receive codes. The remaining codes available are:

100, 101, 111, and 110.

S_5 is adjacent to S_4 in the graph and therefore must receive a code which differs by only one bit from the code of S_4. From the available codes, the only one which differs from code 011 by one bit is code 111. Therefore S_5 is assigned to code 111. Next, S_6 must be assigned a code which is adjacent to S_1 and S_5. Therefore, it must be given a code from the available codes which differs by only one bit from 010 and 111. Therefore S_6 is coded with 110.

111 S_5
110 S_6

These code assignments and the relationships between the groups in the graph can be seen in the hypercube of Figure 5.10.

5.6.3 Assignment Of Supercube Of Codes To Corresponding Columns

Finally, the last part of the primary encoding phase is to assign the largest supercube of symbol codes possible to the corresponding columns. Using the procedure outlined in Section 5.5.2.3, begin by assigning codes to columns for cases considered trivial. At first, all columns which are compatible with 2^k symbols are considered trivial to encode. The supercube is checked to see if any codes are included which are not in the constraint. If so, that column is placed in the list of

Columns	Symbols	Symbol Codes	Supercubes	Acceptable Encodings
0	S2,S3	000 001	00-	000 001 00-
1	S5,S6	111 110	11-	111 110 11-
2	S1,S2,S3,S4	010 000 001 011	0--	010 001 01- 0-1 000 011 00- 0-0 0--
3	S3	000	000	000
4	S1,S4,S5,S6	010 011 111 110	-1-	010 111 11- -10 -1- 011 110 01- -11
5	S2	000	000	000
6	S3,S4	001 011	0-1	001 011 0-1
7	S5,S6	111 110	11-	111 110 11-
8	S6	110	110	110
9	S4	011	011	011
* 10	S3,S4,S5	001 011 111	--1	001 011 111 0-1 -11
* 11	S1,S2,S3,S4,S5	010 000 001 011 111	---	010 001 111 01- 0-1 000 011 00- 0-0 0-- 11
12	S1	010	010	010
13	S1,S2,S3,S4	010 000 001 011	0--	010 001 01- 0-1 000 011 00- 0-0 0--
14	S5	111	111	111
15	S4,S5	011 111	-11	011 111 -11

Circled codes indicate the column codes determined by the algorithm presented in this thesis.

Figure 5.11: Application of Supercube Operation in Encoding.

non-trivial cases $LNTC$ to be coded after all trivial cases. If not, then the column is assigned to the supercube of all codes in the constraint.

For trivial cases, the order of assignment of codes does not matter. Therefore, these columns are assigned in binary counting order. The first column to be encoded is column 0 which is compatible with symbols S_2 and S_3. Since there are 2^k symbols compatible with column C_0 (where $k = 1$ in this case), proceed to check the second condition required for a trivial encoding.

000 S_2

001 S_3

00- = The supercube of codes.

Since the supercube does not contain any codes that are not in the constraint of column C_0, then C_0 is encoded with "00-". Similarly, the remaining columns are coded which do not fall into the category of non-trivial. It turns out that all columns except for C_{10} and C_{11} are encoded as trivial cases (they receive the supercube of all symbols they are compatible with). These can be seen in the Table from Figure 5.11.

For non-trivial cases, encode columns starting with the largest number of symbols that any one column is compatible with. Therefore column C_{11} is encoded first. The largest group of 2^k symbols that C_{11} is compatible with is 4. There are 5 possible groups of 4 symbols ((1,2,3,4), (1,3,4,5),(1,2,4,5),(1,2,3,5),(2,3,4,5)). Of these, only one is possible because the supercubes of the others include codes which are not in the constraint. For example, the supercube of symbols (1,2,3,5) is "$- \; - \; -$". This includes the code of symbol S_6 which is not in the constraint. Therefore C_{11} is encoded with the supercube of symbols (1,2,3,4) resulting in the code "$0 \; - \; -$".

For the only other non-trivial case(C_{10}), there are 3 possible groups of 2^k symbol codes that C_{10} may be encoded with(k = 1 in this case). The 3 groups are ((3,4),(3,5),(4,5)). Group (3,5) is not allowed since its supercube ($- \; - \; 1$) includes the code of an unused code (1 0 1). More is discussed later in this section regarding unused codes. But first, the assignment of codes to columns is completed. For the remaining groups (3,4) and (4,5) we apply the cost function $Cost3$ to determine which codes(($0 \; - \; 1$) or ($- \; 1 \; 1$)) to select. The following are the columns that are adjacent to column C_{10} in G:

0010 (column C_2)

1000 (column C_8)

1011 (column C_{11})

1110 (column C_{14})

$$Cost3 = \sum_{j=1}^{n} \text{HAM\_DIST}(\text{CODE}(C_i),\text{CODE}(C_j)).$$

$$Cost3 = HAM\_DIST(CODE(C_{10}), CODE(C_2)) +$$

$$HAM\_DIST(CODE(C_{10}),\ CODE(C_8)) \ +$$

$$HAM\_DIST(CODE(C_{10}),\ CODE(C_{11})) \ +$$

$$HAM\_DIST(CODE(C_{10}),\ CODE(C_{14}))$$

For C_{10} encoded with "0 $-$ 1" the cost is:

$Cost3(C_{10}) = 0 + 2 + 0 + 1 = 3$

For C_{10} encoded with " $-$ 1 1" the cost is:

$Cost3(C_{10}) = 0 + 1 + 0 + 0 = 1$

The code resulting in the lowest cost is chosen. Therefore column C_{10} is assigned

H

ab \ xyz	000	001	011	010	110	111	101	100
00	0	0	0	0	1	1	-	-
01	1	0	0	1	-	-	-	-
11	0	0	1	1	1	-	-	-
10	-	0	0	-	1	0	-	-
	S2	S3	S4	S1	S6	S5		

G

cd \ ef	00	01	11	10
00	000	111	001	010
01	010	000	110	011
11	010	000	111	111
10	110	011	000	111

a) G and H resulting from encoding with disjoint column codes.

H

ab \ xyz	000	001	011	010	110	111	101	100
00	0	0	0	0	1	1	-	-
01	1	0	0	1	0	-	-	-
11	0	0	1	1	1	1	-	-
10	0	0	0	0	1	0	-	-
	S2	S3	S4	S1	S6	S5		

G

cd \ ef	00	01	11	10
00	00-	11-	001	0--
01	-1-	000	11-	0-1
11	010	0--	-11	111
10	110	011	0--	-11

b) G and H resulting from the new encoding approach presented. It can be observed that many don't cares have been introduced into the G subfunction by encoding with nondisjoint column codes.

Figure 5.12: Resulting Sub-functions G and H for the Example.

the code " – 1 1". That completes the encoding process for this example. The final results of the encoding process for **DC_ENC** are shown in Figure 5.12b. Observe that many don't cares have been introduced into the G sub-function. Also, observe that the Hamming distances between columns in the H sub-function have been minimized. Shown for comparison in Figure 5.12a is an arbitrary encoding using disjoint column codes.

Note that the use of the unused codes can improve the quality of the encoding in some cases. The encoding approach presented in this chapter does use the unused codes when the heuristics in the cover set selection algorithm determines that they should be used. However, there is a trade-off and complicated heuristics may be required to ensure that there is a benefit from particular uses of the unused codes. In the example presented, one would not benefit "overall" if he would encode C_{10} with the supercube of the three symbols it is compatible with plus an unused code. If one would use the unused code(1 0 1) then the code " – – 1" can be given to column C_{10}. But this makes it necessary to assign one of the don't care columns in H to either symbol S_3, S_4 or S_5. If S_5 is chosen, since it has the fewest care bits, then column "1 0 1" in the successor function H will be changed from all don't cares to a partially specified column, as illustrated below:

From	To
–	1
–	–
–	–
–	0

This results in adding one don't care to the predecessor function G and removing two don't cares from the successor function H. In some cases it may be a good trade-off, but in other cases it may happen that many more don't cares would be removed from H (changed to specified values) than would be introduced into G.

5.7 Conclusions

Unlike some approaches which optimize only the input encoding or only the output encoding, the approach presented in this section allows to satisfy multiple constraints and objectives for input and output encodings concurrently, and in a relatively systematic manner. A description of a method for cover set selection was outlined which may be used to satisfy the objective to increase the number of don't cares in the sub-function G while minimizing the loss of don't cares in the sub-function H by preselecting compatible classes with large overlap for the cover set.

The encoding approach presented in this chapter can significantly reduce the resulting complexity of the predecessor logic when $|B|/|A| \geq 1$ and when functions are highly unspecified. The expression $|B|/|A| \geq 1$ simply implies that the number of blocks in the bound set($|B|$) is greater than the number of blocks in the free set($|A|$). How is the complexity of the predecessor logic reduced by using the proposed encoding approach **DC_ENC** ? The complexity of the predecessor logic

is reduced by utilizing overlap of classes in the cover set to produce don't cares in the outputs(codes) of the predecessor sub-function. Why the condition $|B|/|A| \geq 1$? This is by no means an exact cut-off value for all functions. Rather this condition is given to indicate when the potential for utilizing overlap of classes in the cover set is greatest. The potential for overlap of classes in the cover set increases as the ratio $|B|/|A|$ increases. The potential for utilizing overlap of classes in the cover set is greatest when there are many columns which are compatible with multiple classes in the cover set. Why the additional condition that functions must be highly unspecified ? If functions are not at least partially unspecified then there will not be any overlap in the cover set. No overlap in the cover set means no potential for assigning don't cares in codes of columns without increasing the number of encoding bits.

For functions which are highly specified or when $|B|/|A| << 1$ this encoding approach may result in little or no benefit over existing encoding approaches. However, it should be noted that any highly specified function can be made highly unspecified by simply performing nondisjoint decompositions. Therefore, the primary criterion for determining whether conditions are appropriate for this encoding approach is whether or not $|B|/|A| \geq 1$.

The example of the **DC_ENC** encoding approach illustrated how a substantial number of don't cares can be introduced into sub-function G as a result of encoding columns with codes of multiple symbols. Also demonstrated was how

multiple constraint satisfaction can be accomplished through a relatively simple set of heuristics applied to an edge-weighted connection graph. Use of minimum Hamming distances between the optional codes and the adjacent codes assigned illustrated how the complexity of the resulting function could be further simplified.

Though the results from small examples calculated by hand are very promising, numerous tests on the implemented algorithm would need to be performed in order to get an accurate assessment of the algorithms' potential. Also, it should be emphasized that if a decomposition program is designed only for small bound sets, then it would be better to use a much simpler encoding approach(input-only encoder) which would be specifically designed for small bound sets.

CHAPTER 6

FUNCTIONAL DECOMPOSITION PROGRAM MULTIS

6.1 Introduction To Program MULTIS

MULTIS(MULTI-Strategy decomposer) is the main calling program. of a binary input, binary output Curtis-style functional decomposition program. It was implemented at Portland State University by the Portland Oregon Logic Optimization group(POLO) as a testbed to compare various decomposition algorithms. The program was written in C and C++ programming languages. Shown in Figure 6.1 is a block diagram representation of the program MULTIS and the subprograms which are called from it.

The main program MULTIS is primarily responsible for general strategy and file management. The general strategy consists of specifying the number of variables in the bound set and free set, selection of partitioning algorithms, and selection of serial or parallel decomposers.

After each level of a decomposition, MULTIS checks whether a block(or subfunction) needs to be decomposed further. A block doesn't need to be decomposed further if it is less than or equal to the final block size specified. The final block

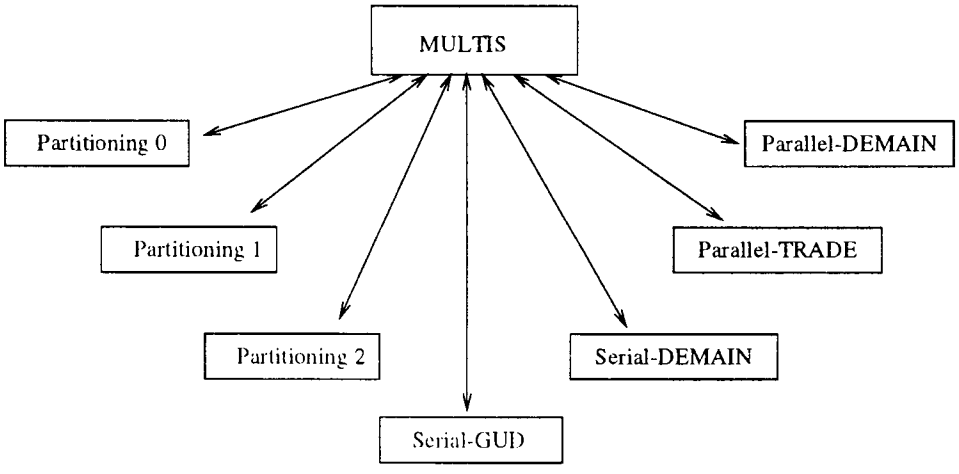


Figure 6.1: MULTI-Strategy decomposer(**MULTIS**)

size is specified on the command line, by the user, as the maximum number of block inputs and outputs allowed to any logic block in the resulting multi-level decomposition.

Input to the program is a file in Espresso format which typically represents a truth table for a circuit. Output of the program consists of a blif result file, a set of all intermediate result files in Espresso format, and a brief summary of results to standard I/O. The blif result file is a description of the final circuit after it has been decomposed into a set of gates(or subfunctions). The brief summary of results to standard I/O lists the total DFC, total number of blocks, number of literals and rows in the blif result file, and the total user time spent in the decomposition. The "number of ones" is a total of the number of non-negated input literals in all subfunctions of the decomposition. The "number of zeros" is a total of the number of negated input literals in all subfunctions of the decomposition.

An example of the output summary of results is as follows:

- DFC = 512
- number of ones = 372
- number of zeros = 228
- number of rows = 205
- number of CLBs = 128

Decomposition subprograms DEMAIN and GUD shown in Figure 6.1 are introduced in Sections 6.2 and 6.3. For details on partitioning subprogram #0, refer to the paper by Gatlin[22]. For details on partitioning subprograms #1 and #2, refer to the paper by Wan[65]. Details on parallel decomposition subprogram N_TO_ONE are not repeated here as they were presented in Section 2.3.2.1.

6.2 Decomposition Program DEMAIN

DEMAIN is a multi-level functional decomposition program. The theoretical basis of DEMAIN is based on a number of papers by Luba[28][30][31]. DEMAIN was modified by the POLO group so that it can be incorporated into a larger testing program(MULTIS) with additional decomposers and additional options. There are two primary components in DEMAIN. These are serial and parallel decomposition.

6.2.1 Serial Decomposition:

The serial decomposition of DEMAIN can be characterized as a Curtis like serial decomposition Recall that in a Curtis like decomposition the number of outputs of the predecessor block(also referred to as the G block) is less than the number of inputs to that block. The serial decomposition consists of four main sub-components:

1. partitioning of variables into the free and bound set
2. column compatibility checking to create compatibility graph
3. column minimization to create a cover set
4. encoding of compatible classes in the cover set

The partitioning of variables into the free and bound set is done by starting with a seed partition, having the number of variables equal to the number of inputs for the predecessor block, and exchanging one variable for each new partition tried. This is continued until a decomposition is found or until all partitions of the specified number of inputs have been exhausted. A decomposition is found for a given partition if the encoding is possible for the number of outputs specified by the user.

The compatibility graph is constucted by checking the compatibility of pairs of columns(or blocks). If a pair of columns are compatible, then an edge is added to

the graph between the nodes corresponding to the two columns. All combinations of pairs of columns are checked.

Once the compatibility graph has been constructed, a process of building maximum cliques is performed. Once complete, the minimum number of maximum cliques are chosen which cover all the blocks in the bound set(i.e. partition Π_B). This minimum number of maximum cliques forms the cover set Pi_G which is passed on to be encoder.

6.2.2 Parallel Decomposition:

The Parallel Decomposition of DEMAIN splits a multi-output function into two new functions. Unlike the parallel decomposition approach N_TO_ONE, DEMAIN does not split up all outputs so that each new function has only one output. Also different is that the new functions created by DEMAIN do not typically share all the same inputs(i.e. one of the two new functions may have inputs X_1, X_3, X_4, X_5 while the other function may have inputs X_1, X_2, X_5). None of the outputs of one of the new functions is also an output of the other.

For a detailed description of the parallel decomposition algorithm used in DEMAIN, refer to [58][31].

6.3 Decomposition Program GUD

The General Universal Decomposer(GUD-binary version) is a functional decomposition program which is capable of decomposing binary single output and multi-output functions. The program was written in C and C++. Another version of GUD(GUD_MV), is capable of decomposing multi-valued as well as binary single output and multi-output functions. It should be noted that, unlike program GUD, which was written by several programmers, GUD_MV was written/(rewritten) exclusively by Stanislaw Grygiel. Version GUD is capable of working as a decomposer by itself. However, it has been modified to be incorporated into a larger testing program with additional decomposers and additional options(MULTIS). GUD uses two types of data structures, BDDs and cube arrays. The input data is read into BDDs and cube arrays. The BDDs are used for various steps in the decomposition. The arrays of cubes are used for encoding purposes. An output blif file is used to verify that the resulting decomposed function is equivalent to the ON-set and OFF-set of the original function. Verification was done using the verifier SIS(Berkley software package) program. The following are itemized lists of algorithms implemented in the program GUD:

GUD has two algorithms for partitioning variables to free and bound sets:

- Pseudo Random Partitioning

- Sequential Binary Partitioning

GUD has two algorithms for column compatibility checking:

- Pair Compatibilty Approach
- Group Compatibilty Approach

GUD has three algorithms for column minimization:

- Clique Covering.
- Graph Coloring.
- Graph Coloring using domination.

GUD has one algorithm for Cover Set Selection:

- Cover Set Selection

GUD also has two encoding algorithms:

- Dichotomy Encoding
- Sequential Binary Encoding

The implementation of the partitioning algorithms was done by Paul Burkey[52]. The implementation of the Pair Compatibilty Approach was done by Jinsong Zhang, Zhi Wang, and Roger Shipman[52]. The implementation of the Group Compatibilty Approach was done by myself[10]. The implementation of the Clique

Covering algorithm was done by Jinsong Zhang, Zhi Wang, and Roger Shipman[52]. Graph Coloring algorithms were implemented by Roger Shipman[59]. The implementation of Cover Set Selection was done by myself[11][52]. Dichotomy Encoding was implemented by Nick Iliev[24]. Sequential Binary Encoding was implemented by Rahul Malvi[52]. For a general understanding of the flow of control in the program GUD, see the flow diagram in Figure 6.2. It should be noted that a function call to program GUD(subprogram to MULTIS) is only one step of a multi-level hierarchical decomposition.

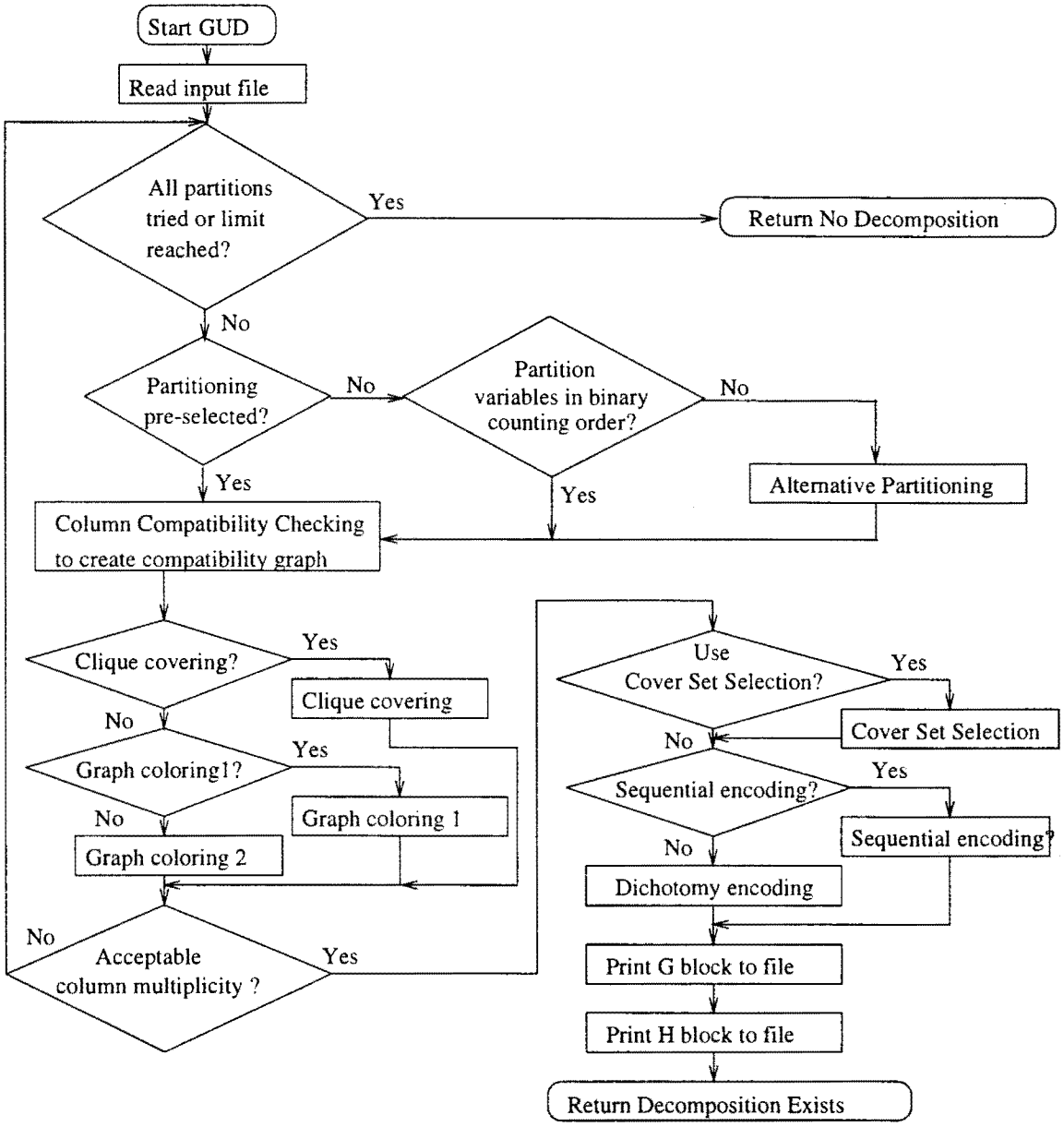


Figure 6.2: Flow Diagram for program **GUD**

CHAPTER 7

VARIOUS EXPERIMENTAL RESULTS OF PROGRAM MULTIS/GUD

7.1 Introduction

In this chapter, results from various tests on different decomposition programs are presented. Unless otherwise specified, all results were obtained from subprograms called from the main program MULTIS. For example, decomposition programs DEMAIN and GUD were called from the main program MULTIS. For each of the various comparisons made, there is a corresponding *Summary of Results* table which summarizes details relavent to each comparison. For each set of tables, corresponding to a particular comparison, there is a brief explanation of the comparison and a discussion of the results obtained. In Section 7.2, comparisons are made between different serial decomposition programs and between results on fully specified vs. highly unspecified functions. In Section 7.3, comparisons are made between different encoding approaches and between results on functions with 70% don't cares vs. 90% don't cares. In Section 7.4, comparisons are made between different partitioning approaches. In Section 7.5, some additional comparisons are made between several decomposition programs on MCNC benchmarks. These additional comparisons were made with the most recent version of program

GUD(binary version).

The following are descriptions for categories and labeling used in Table 7.1 through Table 7.12. The reader may wish to skip past these descriptions and refer back only as needed.

Strategy Code Labeling for program GUD:

The number "1" after a program name, such as GUD(1-a), designates the general strategy used for selecting bound sets. The "dash a" is explained below. The number "1" signifies that the strategy used always started by trying Ashenhurst(single output) decompositions first. If none exist or the limit has been reached(maximum of 100 partitions have been checked) then Curtis like decompositions are searched for next. In the comparisons of program DEMAIN and GUD, the general strategy option "1" is the same for both programs. This is controlled by the calling program MULTIS.

An "a" signifies the following options selected for the program GUD:

1) GUD partitioning method 1

2) column compatibility method 1 (set covering)

3) encoding method 3 (dichotomy encoding w/ set cover selection)

A "b" signifies the following options selected for the program GUD:

1) GUD partitioning method 2 (all bound sets of a specified size)

2) column compatibility method 1 (set covering)

3) encoding method 1 (binary encoding)

A "c" signifies the following options selected for the program GUD:

1) GUD partitioning method 2 (all bound sets of a specified size)

2) column compatibility method 1 (set covering)

3) encoding method 2 (dichotomy encoding)

A "d" signifies the following options selected for the program GUD:

1) GUD partitioning method 2 (all bound sets of a specified size)

2) column compatibility method 1 (set covering)

3) encoding method 3 (dichotomy encoding w/ set cover selection)

Strategy Code Labeling for program DEMAIN:

Unlike program GUD, which has several combinations of algorithm options, the program DEMAIN does not have optional algorithms to call for each phase

of the decomposition process(i.e., there is only one algorithm for partitioning, one algorithm for clique covering and one algorithm for encoding). To indicate there are no optional algorithms to select, the code "z" is given. Therefore, in the comparisons between DEMAIN and GUD, the strategy code for DEMAIN is always (1-z).

For more details on algorithms used in program GUD, see the documentation for program MULTIS/GUD[52]. For more details on algorithms used in program DEMAIN, see papers by Luba[28] and Selvaraj[58].

7.2 Comparisons Between Decomposition Programs On Fully Specified vs. Highly Unspecified Functions

The first set of results is for a comparison between programs DEMAIN and two versions of GUD. In Table 7.1 and Table 7.2 are the multi-level decomposition results for each of these programs ran on a suite of benchmark functions referred to as the FLASH functions. In this comparison, the functions are only 30% specified(70% don't cares). These functions were generated from the original set of fully specified FLASH functions. All partially unspecified FLASH functions were generated using the program FLASH. FLASH is the Pattern Theory group's program at Wright Patterson Air Force Base. The purpose of comparing results from fully specified and highly unspecified functions was to see how dramatically

DFC values were affected and to see how similar the decomposed functions were. The primary criteria used in these comparisons is DFC. Time is also shown for additional comparison.

A convenient way to compare the results of each of these programs is by analyzing the Summary of Results in Table 7.3. However, for specific information relating to specific benchmarks, one can refer to the previous two tables. From categories A, B, C, F and G in the Summary of Results table, one can see that both versions of GUD performed better than DEMAIN in terms of lower DFC values. However, these versions of GUD had a major drawback in terms of execution time as shown in categories D and E.

Because the times are listed in real time, there is not an accurate comparison between the programs in terms of user time. However, the program DEMAIN always completes decompositions in much less time than either version of GUD. It should be noted that the results in Table 7.1 through Table 7.15 were obtained with early versions of GUD. In later versions of GUD, execution time became a higher priority and therefore subsequent results were obtained in terms of user time. Why use real time in the first place ? Because statistics were already output in real time and because time was not the primary issue of comparison when these results were obtained.

Table 7.1: Comparison of DEMAIN and Variations of GUD(70% don't cares)

Program	DEMAIN		GUD			
	Strategy 1-z		Strategy 1-a		Strategy 1-d	
Benchmark	DFC	time(s)	DFC	time(s)	DFC	time(s)
psu_add0_70	28	1.1	64	209.3	28	9.1
psu_and_or_chain8_70	28	1.2	24	23.6	32	21.6
psu_ch176f0_70	28	0.9	24	3.8	24	3.9
psu_ch177f0_70	20	0.7	20	3.3	20	3.9
psu_ch22f0_70	52	1.4	40	30.1	44	10.2
psu_ch30f0_70	40	1.3	44	43.3	44	15.5
psu_ch47f0_70	56	2.7	64	140.6	44	48.6
psu_ch52f4_70	68	2.3	64	172.1	68	456.9
psu_ch70f3_70	68	2.6	52	386.8	28	7.1
psu_ch74f1_70	64	2.5	76	227.0	40	42.5
psu_check_fail_70	56	2.2	56	170.4	60	57.4
psu_contains_4_ones_70	80	24.4	76	688.1	76	2118.2
psu_greater_than_70	28	1.6	44	954.5	44	19.7
psu_interval1_70	56	2.5	52	325.2	56	108.0
psu_interval2_70	124	2.6	76	785.3	120	2814.3
psu_kdd1_70	28	1.1	16	4.5	16	6.2
psu_kdd2_70	28	0.8	28	3.7	24	4.7
psu_kdd3_70	28	1.2	28	23.3	24	5.7
psu_kdd4_70	12	0.7	28	10.3	12	3.9
psu_kdd5_70	44	1.7	40	15.9	40	11.4
psu_kdd6_70	28	1.0	28	3.9	28	5.1
psu_kdd7_70	64	2.3	28	3.9	28	4.7
psu_kdd8_70	28	1.0	24	20.2	24	5.6
psu_kdd9_70	28	0.9	28	25.5	28	5.4
psu_kdd10_70	28	0.9	28	23.9	24	5.8
psu_majority_gate_70	60	2.7	56	220.3	56	68.3
psu_modulus2_70	60	2.2	72	182.2	56	423.2
psu_monkish1_70	28	1.0	28	7.8	24	4.8
psu_monkish2_70	40	1.9	40	71.3	40	47.3
psu_monkish3_70	40	1.4	28	10.9	28	5.1
psu_mux8_70	28	1.8	52	54.7	48	128.2
psu_nnr1_70	40	2.5	40	48.5	40	55.1
psu_nnr2_70	40	1.4	28	31.8	32	10.5
psu_nnr3_70	68	1.8	68	716.2	68	460.4

Table 7.2: Continuation of Table 7.1

Program	DEMAIN		GUD			
	Strategy 1-z		Strategy 1-a		Strategy 1-d	
Benchmark	DFC	time(s)	DFC	time(s)	DFC	time(s)
psu_or_and_chain8_70	32	0.9	28	4.5	28	4.8
psu_pal_70	28	0.9	28	4.1	28	3.8
psu_pal_dbl_output_70	132	2.2	104	371.6	92	522.6
psu_parity_70	28	0.9	28	3.8	28	3.5
psu_primes8_70	92	20.3	80	741.5	56	367.5
psu_remainder2_70	84	1.9	80	164.2	60	372.5
psu_rnd1_70	140	31.3	140	1447.5	176	2789.4
psu_rnd2_70	148	12.4	132	1306.6	172	3565.0
psu_rnd3_70	144	6.4	112	960.5	160	2702.3
psu_rnd_m1_70	28	1.3	28	3.4	28	3.7
psu_rnd_m5_70	28	1.6	28	8.1	28	4.7
psu_rnd_m10_70	28	1.7	28	8.4	28	5.5
psu_rnd_m25_70	64	4.6	64	187.4	68	393.0
psu_rnd_m50_70	100	23.1	92	661.8	116	2453.5
psu_rndvv36_70	88	3.5	88	257.1	92	80.4
psu_substr1_70	92	6.9	92	292.4	92	461.4
psu_substr2_70	144	6.0	80	232.4	92	2363.7
psu_subtraction1_70	128	9.7	128	392.8	140	723.3
psu_subtraction3_70	12	0.8	20	3.6	20	3.6

Table 7.3: Summary of Results for tables 7.1 and 7.2 for FLASH functions with 70% don't cares

		Category						
Program	Strategy	A	B	C	D(sec)	E(sec)	F	G
DEMAIN	1-z	4	19	43%	6	31.3	3056	57
GUD	1-a	10	26	68%	220	1447.0	2844	53
GUD	1-d	11	23	64%	420	3565.0	2872	53

Categories for Summary of Results table:

A: Total number times when DFC was the lowest(not a tie)

B: Total number times when the lowest DFC was a tie

C: percentage of tests when DFC was the

lowest(i.e., (A+B)/(total # functions) * 100)

D: Average execution time per function(seconds)

E: Longest execution time(seconds)

F: Cumulative DFC

G: Average DFC

Table 7.4: Comparison of DEMAIN and GUD on Fully Specified FLASH Benchmarks

Program				GUD		DEMAIN	
				Strategy 1-d		Strategy 1-z	
Benchmarks	in	out	cubes	DFC	time(s)	DFC	time(s)
psu_add0	8	1	256	28	53	28	12
psu_and_or_chain8	8	1	256	28	50	28	14
psu_ch176f0	8	1	256	24	18	28	4
psu_ch177f0	8	1	256	20	15	20	3
psu_ch22f0	8	1	256	28	21	28	4
psu_ch30f0	8	1	256	44	62	52	7
psu_ch47f0	8	1	256	76	92	80	10
psu_ch52f4	8	1	256	224	14107	220	331
psu_ch70f3	8	1	256	56	557	56	21
psu_ch74f1	8	1	256	112	6239	112	197
psu_contains_4_ones	8	1	256	76	4511	76	220
psu_greater_than	8	1	256	28	40	28	12
psu_interval1	8	1	256	152	7838	136	317
psu_interval2	8	1	256			120	368
psu_kdd1	8	1	256	16	10	28	4
psu_kdd2	8	1	256	24	11	28	4
psu_kdd3	8	1	256	24	12	28	5
psu_kdd4	8	1	256	12	8	20	4
psu_kdd5	8	1	256	64	434	68	21
psu_kdd6	8	1	256	28	12	28	4
psu_kdd7	8	1	256	28	11	28	5
psu_kdd8	8	1	256	24	11	28	5
psu_kdd9	8	1	256	28	12	28	5
psu_kdd10	8	1	256	24	14	28	5
psu_majority_gate	8	1	256	76	4807	80	216
psu_modulus2	8	1	256	76	517	68	21
psu_monkish1	8	1	256	24	10	28	4
psu_monkish2	8	1	256	56	60	56	11
psu_monkish3	8	1	256	32	9	32	4
psu_mux8	8	1	256	52	55	32	5
psu_nnr1	8	1	256	36	338	36	25
psu_nnr2	8	1	256	32	17	32	5
psu_nnr3	8	1	256	240	8614	244	365

Table 7.5: Continuation of Table 7.4

Program				GUD		DEMAIN	
				Strategy 1-d		Strategy 1-z	
Benchmarks	in	out	cubes	DFC	time(s)	DFC	time(s)
psu_or_and_chain8	8	1	256	28	64	28	13
psu_pal	8	1	256	28	18	28	7
psu_pal_dbl_output	8	1	256	188	679	180	50
psu_parity	8	1	256	28	11	28	4
psu_primes8	8	1	256	256	7024	212	326
psu_remainder2	8	1	256	256	7070	180	316
psu_rnd1	8	1	256			256	438
psu_rnd2	8	1	256	256	6713	256	902
psu_rnd3	8	1	256	256	5824	256	926
psu_rnd_m1	8	1	256	28	12	28	5
psu_rnd_m5	8	1	256	80	6450	80	137
psu_rnd_m10	8	1	256	108	6708	108	185
psu_rnd_m25	8	1	256			256	301
psu_rnd_m50	8	1	256	256	6233	256	778
psu_rndvv36	8	1	256	92	86	92	14
psu_substr1	8	1	256	80	3839	80	536
psu_substr2	8	1	256	112	3916	92	512
psu_subtraction3	8	1	256	20	10	20	3
psu_add2	8	1	256	28	15	28	4
psu_add4	8	1	256	20	14	20	3
psu_ch15f0	8	1	256	84	107	80	11
psu_ch83f2	8	1	256			168	213
psu_ch8f0	8	1	256	52	85	44	7
psu_pal_output	8	1	256	256	5528	256	431

Table 7.6: Summary of Results for tables 7.4 and 7.5.

		Category						
Program	Strategy	A	B	C	D(sec)	E(sec)	F	G
GUD	1-d	13	30	81 %	2056.0	14107	4304	81
DEMAIN	1-z	10	30	75 %	132.83	926	4164	79

Categories for Summary of Results table:

A: Total number times when DFC was the lowest(not a tie)

B: Total number times when the lowest DFC was a tie

C: percentage of tests when DFC was the

lowest(i.e., (A+B)/(total # functions) * 100)

D: Average execution time per function(seconds)

E: Longest execution time(seconds)

F: Cumulative DFC

G: Average DFC

In Table 7.4 and Table 7.5 are the multi-level decomposition results for programs DEMAIN and GUD ran on the fully specified versions of the FLASH benchmarks. By comparing the results in the Summary of Results Table 7.3 with the results in Summary of Results Table 7.6, the following observations are made:

1) Average DFC was again lower for GUD(1-d) than it was for DEMAIN.

2) Execution time was again higher for GUD(1-d) than it was for DEMAIN.

3) Average DFCs for GUD and DEMAIN were higher for the set of fully specified functions than they were for the highly unspecified functions.

4) Average execution times for GUD and DEMAIN were higher for the set of fully specified functions than they were for the highly unspecified functions.

An important observation to make, is that in several instances, the same decomposition result was obtained for the fully specified and highly unspecified version of particular benchmarks. For example, benchmark "psu_ch177f0_70" was decomposed into the same identical set of logic blocks by programs DEMAIN and GUD when the benchmark was fully specified and when it was highly unspecified. This is important from the perspective of the Pattern Theory group at Wright Labs. It indicates, at least in certain instances, that some patterns are easily recognized despite having a significant amount of noise(or missing information). Here, the noise(or missing information) is represented by the don't care minterms in a given function.

7.3 Encoding Results

In this section, three different encoding schemes are compared. Each of these encoding schemes were tested as optional versions of the program GUD. For specifics on the differences between encoding options, see the description of categories and labeling in Section 7.1. In Table 7.7 through Table 7.12 are the comparison of results between the different encoding schemes ran on the versions of the FLASH benchmarks with 70% don't cares and 90% don't cares respectively. By comparing the results in the Summary of Results Table 7.9 with the results in Summary of Results Table 7.12, the following observations is made:

1) The difference in results for the different encoding approaches is not dramatic. However, the results indicate that either encoding scheme #1 or #3 yields the best overall results in terms of DFC.

2) Average DFCs for all encoding schemes were higher for the functions which were more highly specified(i.e., between versions of FLASH functions with 70% vs. 90% don't cares).

3) Average execution times for all encoding schemes were higher for the functions which were more highly specified(i.e., between versions of FLASH functions with 70% vs. 90% don't cares).

Table 7.7: Comparison of Different Encoding Approaches Used on FLASH Functions with 70% don't cares

Program	GUD					
ENCODING METHOD	#1(1-b)		#2(1-c)		#3(1-d)	
Benchmarks	DFC	time(s)	DFC	time(s)	DFC	time(s)
psu_add0_70	28	13	28	16	28	14
psu_add2_70	28	8	28	9	28	9
psu_add4_70	20	5	20	7	20	9
psu_and_or_chain8_70	32	20	48	101	32	22
psu_ch15f0_70	80	93	88	133	80	84
psu_ch176f0_70	24	9	24	8	24	8
psu_ch177f0_70	20	8	20	7	20	8
psu_ch22f0_70	40	18	40	21	40	14
psu_ch30f0_70	40	44	40	30	44	18
psu_ch47f0_70	40	59	44	91	44	52
psu_ch52f4_70	68	341	68	878	68	330
psu_ch70f3_70	28	11	28	17	28	11
psu_ch74f1_70	56	55	56	157	40	51
psu_ch83f2_70	76	389	92	939	68	367
psu_ch8f0_70	44	18	44	31	44	19
psu_contains_4_ones_70	76	2298	76	5266	76	2172
psu_greater_than_70	44	23	44	38	44	23
psu_interval1_70	56	62	56	110	56	58
psu_interval2_70	116	2511	176	6358	120	2386
psu_kdd1_70	16	7	16	8	16	7
psu_kdd10_70	24	7	24	10	24	9
psu_kdd2_70	24	7	24	9	24	8
psu_kdd3_70	24	7	24	11	24	8
psu_kdd4_70	12	5	12	7	12	6
psu_kdd5_70	44	13	40	21	40	14
psu_kdd6_70	28	7	28	9	28	8
psu_kdd7_70	44	46	28	9	28	8
psu_kdd8_70	24	7	24	10	24	8
psu_kdd9_70	28	8	28	11	28	10
psu_majority_gate_70	56	58	56	112	56	56
psu_modulus2_70	56	351	88	763	56	335
psu_monkish1_70	24	6	24	9	24	7
psu_monkish2_70	40	21	40	36	40	22
psu_monkish3_70	28	7	28	9	28	8

Table 7.8: Continuation of Table 7.7

Program	GUD					
ENCODING METHOD	#1(1-b)		#2(1-c)		#3(1-d)	
Benchmarks	DFC	time(s)	DFC	time(s)	DFC	time(s)
psu_mux8_70	48	63	48	114	48	59
psu_nnr1_70	40	53	40	82	40	45
psu_nnr2_70	32	9	32	13	32	9
psu_nnr3_70	68	388	68	810	68	368
psu_or_and_chain8_70	28	8	28	11	28	9
psu_pal_70	28	7	28	11	28	8
psu_pal_dbl_output_70	88	466	164	1546	92	444
psu_pal_output_70	0	0	0	1	0	1
psu_parity_70	28	8	28	9	28	8
psu_primes8_70	56	355	88	704	56	317
psu_remainder2_70	64	392	92	870	64	336
psu_rnd1_70	144	2795	180	6280	176	2726
psu_rnd2_70	152	2701	180	7473	172	2746
psu_rnd3_70	116	2468	152	6291	160	2734
psu_rnd_m1_70	28	6	28	8	28	8
psu_rnd_m10_70	28	8	28	11	28	10
psu_rnd_m25_70	64	396	68	972	68	399
psu_rnd_m5_70	28	8	28	10	28	8
psu_rnd_m50_70	108	2428	80	6244	80	2404
psu_rndvv36_70	96	91	92	144	92	93
psu_substr1_70	92	435	100	820	92	448
psu_substr2_70	100	2399	104	7499	92	2399
psu_subtraction1_70	124	624	176	2276	140	668
psu_subtraction3_70	20	6	20	6	20	9

Table 7.9: Summary of Results for tables 7.7 and 7.8.

		Category						
Program	Encoding	A	B	C	D(sec)	E(sec)	F	G
GUD	#1	8	43	88 %	390.6	2795	2968	51
GUD	#2	0	41	71 %	990.6	7499	3356	58
GUD	#3	3	46	84 %	386.6	2746	3016	52

Categories for Summary of Results table:

A: Total number times when DFC was the lowest(not a tie)

B: Total number times when the lowest DFC was a tie

C: percentage of tests when DFC was the

lowest(i.e., (A+B)/(total # functions) * 100)

D: Average execution time per function(seconds)

E: Longest execution time(seconds)

F: Cumulative DFC

G: Average DFC

Table 7.10: Comparison of Different Encoding Approaches Used on FLASH Functions with 90% don't cares

Program	GUD					
ENCODING METHOD	#1(1-b)		#2(1-c)		#3(1-d)	
Benchmarks	DFC	time(s)	DFC	time(s)	DFC	time(s)
psu_add0_90	24	6	32	6	24	7
psu_add2_90	56	30	44	10	52	28
psu_add4_90	20	4	20	5	20	9
psu_and_or_chain8_90	40	9	24	5	24	6
psu_ch15f0_90	44	11	32	9	32	12
psu_ch176f0_90	40	9	28	5	40	12
psu_ch126f0_90	20	4	20	5	20	8
psu_ch22f0_90	20	4	20	4	20	7
psu_ch30f0_90	28	6	20	5	20	7
psu_ch47f0_90	28	7	28	5	28	7
psu_ch52f4_90	24	5	28	5	28	9
psu_ch70f3_90	20	5	24	5	20	7
psu_ch74f1_90	24	5	28	6	28	8
psu_ch83f2_90	56	27	40	9	56	27
psu_ch8f0_90	0	1	0	0	0	1
psu_contains_4_ones_90	28	8	40	9	28	7
psu_greater_than_90	28	7	28	5	28	8
psu_interval1_90	32	5	32	4	32	5
psu_interval2_90	28	5	44	9	28	7
psu_kdd1_90	24	4	56	29	24	8
psu_kdd10_90	24	5	44	9	24	8
psu_kdd2_90	24	3	24	4	24	7
psu_kdd3_90	24	5	28	6	24	6
psu_kdd4_90	12	4	12	4	12	4
psu_kdd5_90	32	9	28	5	28	6
psu_kdd6_90	24	5	24	5	24	5
psu_kdd7_90	28	6	28	6	28	5
psu_kdd8_90	44	10	28	5	32	5
psu_kdd9_90	44	8	32	5	44	9
psu_majority_gate_90	44	8	44	9	40	11
psu_modulus2_90	28	5	28	5	28	8
psu_monkish1_90	0	1	0	0	0	1
psu_monkish2_90	44	8	44	10	44	11
psu_monkish3_90	24	4	24	4	24	7

Table 7.11: Continuation of Table 7.10

Program	GUD					
ENCODING METHOD	#1(1-b)		#2(1-c)		#3(1-d)	
Benchmarks	DFC	time(s)	DFC	time(s)	DFC	time(s)
psu_mux8_90	24	5	24	5	24	7
psu_nnr1_90	44	11	40	10	40	12
psu_nnr2_90	32	5	32	5	32	7
psu_nnr3_90	28	5	24	5	24	7
psu_or_and_chain8_90	28	5	32	8	28	7
psu_pal_90	0	3	0	3	0	4
psu_pal_dbl_output_90	56	30	44	24	52	32
psu_pal_output_90	32	4	40	9	32	6
psu_parity_90	44	11	28	6	44	11
psu_primes8_90	24	5	28	5	24	5
psu_remainder2_90	32	8	56	26	32	8
psu_rnd1_90	32	5	44	9	44	11
psu_rnd2_90	40	9	40	10	40	11
psu_rnd3_90	44	10	32	5	44	12
psu_rnd_m1_90	0	2	0	3	0	3
psu_rnd_m10_90	0	3	0	4	0	3
psu_rnd_m25_90	24	5	24	5	24	5
psu_rnd_m5_90	0	3	0	3	0	3
psu_rnd_m50_90	28	4	28	5	28	6
psu_rndvv36_90	44	10	44	9	44	12
psu_substr1_90	40	11	40	10	40	12
psu_substr2_90	24	7	36	9	24	8
psu_subtraction1_90	44	9	44	10	44	13
psu_subtraction3_90	20	7	20	4	20	7

Table 7.12: Summary of Results for tables 7.10 and 7.11.

		Category						
Program	Encoding	A	B	C	D(sec)	E(sec)	F	G
GUD	#1	3	40	74 %	7.2	30	1664	28.7
GUD	#2	8	34	72 %	7.0	29	1676	28.9
GUD	#3	1	46	81 %	8.5	32	1612	27.8

Categories for Summary of Results table:

A: Total number times when DFC was the lowest(not a tie)

B: Total number times when the lowest DFC was a tie

C: percentage of tests when DFC was the

lowest(i.e., (A+B)/(total # functions) * 100)

D: Average execution time per function(seconds)

E: Longest execution time(seconds)

F: Cumulative DFC

G: Average DFC

7.4 Variable Partitioning Results

Table 7.13 through Table 7.15 shows results comparing three separate partitioning approaches. These tests were ran in order to determine the relative effectiveness of the different approaches in finding disjoint Ashenhurst decom-

positions(single output). RGP is a partitioning method programmed by Robert Gatlin[22]. GUD(1-b) is a partitioning method programmed by Paul Burkey[52]. DEMAIN is a partitioning method programmed by Luba[28].

The following are descriptions for categories used in Table 7.13 and Table 7.14.

Categories:

N: Number of Partitions tried

F: Ashenhurst decomposition was found using one of the partitions selected(yes or no).

Table 7.13: Comparison of Partitioning Approaches Used with FLASH Functions Having 70% don't cares

Partitioning Method	RGP			DEMAIN			GUD(1-b)		
Benchmarks	N	F	t(sec)	N	F	t(sec)	N	F	t(sec)
psu_add0_70	1	y	0.2	2	y	0.4	5	y	2.9
psu_and_or_chain8_70	1	y	0.3	4	y	0.2	5	y	2.9
psu_ch176f0_70	1	y	0.3	1	y	0.3	1	y	1.1
psu_ch177f0_70	1	y	0.2	1	y	0.1	1	y	0.1
psu_ch22f0_70	2	y	0.2	1	y	0.2	1	y	1.1
psu_ch30f0_70	3	y	0.4	2	y	0.3	6	y	3.3
psu_ch47f0_70	4	y	0.3	2	y	0.2	4	y	2.2
psu_ch52f4_70	4	n	0.3	1	y	0.3	1	y	1.0
psu_ch70f3_70	1	y	0.2	2	y	0.2	3	y	1.9
psu_ch74f1_70	1	y	0.4	2	y	0.3	2	y	1.5
psu_check_fail_70	3	y	0.4	1	y	0.2	4	y	2.5
psu_contains_4_ones_70	none	exist							
psu_greater_than_70	3	n	0.3	16	y	0.5	13	y	6.8
psu_interval1_70	3	n	0.3	5	y	0.3	15	y	8.0
psu_interval2_70	none	exist							
psu_kdd1_70	1	y	0.2	2	y	0.3	2	y	1.5
psu_kdd2_70	1	y	0.2	1	y	0.2	1	y	1.1
psu_kdd3_70	2	y	0.3	2	y	0.2	2	y	1.4
psu_majority_gate_70	2	y	0.3	7	y	0.4	7	y	3.9
psu_modulus2_70	2	y	0.3	2	y	0.3	1	y	1.0
psu_monkish1_70	2	y	0.3	1	y	0.3	1	y	1.1
psu_monkish2_70	2	y	0.3	18	y	0.6	10	y	4.9
psu_monkish3_70	2	y	0.3	1	y	0.3	1	y	1.0
psu_mux8_70	1	y	0.1	1	y	0.1	1	y	1.1
psu_nnr1_70	3	n	0.3	27	y	0.7	26	y	13.2
psu_nnr2_70	1	y	0.4	2	y	0.2	4	y	2.4
psu_nnr3_70	4	n	0.3	20	y	0.5	24	y	12.2
psu_or_and_chain8_70	1	y	0.2	2	y	0.2	3	y	1.8
psu_pal_70	1	y	0.2	1	y	0.2	1	y	1.1
psu_pal_dbl_output_70	1	y	0.2	1	y	0.3	1	y	1.1

Table 7.14: Continuation of Table 7.13

Partitioning Method	RGP			DEMAIN			GUD(1-b)		
Benchmarks	N	F	t(sec)	N	F	t(sec)	N	F	t(sec)
psu_parity_70	1	y	0.2	1	y	0.2	1	y	1.1
psu_primes8_70	none	exist							
psu_remainder2_70	3	n	0.3	4	y	0.3	25	y	12.2
psu_rnd1_70	none	exist							
psu_rnd2_70	none	exist							
psu_rnd3_70	none	exist							
psu_rnd_m10_70	1	y	0.3	2	y	0.2	3	y	2.0
psu_rnd_m1_70	1	y	0.2	1	y	0.2	1	y	1.0
psu_rnd_m25_70	4	y	0.3	2	y	0.3	23	y	11.9
psu_rnd_m50_70	none	exist							
psu_rnd_m5_70	1	y	0.2	2	y	0.2	3	y	1.9
psu_rndvv36_70	2	y	0.3	3	y	0.3	2	y	1.4
psu_substr1_70	1	y	0.4	1	y	0.2	1	y	1.2
psu_substr2_70	none	exist							
psu_subtraction1_70	3	n	0.3	2	y	0.2	4	y	2.3
psu_subtraction3_70	1	y	0.3	1	y	0.1	1	y	1.1

Categories for Summary of Results:

A: Number of times a partition resulted in an Ashenhurst decomposition.

B: Number of times a partitions did not result in an Ashenhurst decomposition.

C: Hit/Miss ratio defined as A/B.

Table 7.15: Summary of Results for tables 7.13 and 7.14

	CATEGORY								
Partitioning Method	A	B	C	D	E	F	G	H	I(sec)
RGP	31	41	0.76	82%	2.3	31	41	76%	0.2
DEMAIN	38	108	0.35	100%	3.8	32	40	80%	0.2
GUD(1-b)	38	172	0.22	100%	5.5	27	68	40%	3.1

D: Percentage of benchmarks that a partitioning approach yielded an Ashenhurst decomposition, defined as (A*100)/38(38 being the total number of benchmarks where an Ashenhurst decomposition exists).

E: Average number of partitions that must be tried to get an Ashenhurst decomposition(provided one exists), defined as (A+B)/A.

F: Same as category A except that only the first 4 partitions tried are considered.

G: Same as category B except that only the first 4 partitions tried are considered. This category is defined to be F/38.

H: Hit/Miss ratio defined as F/G.

I: Average execution time per partition selected(seconds).

From the results in Summary of Results Table 7.15, the following observations were made:

1) The approaches RGP and DEMAIN performed much better than GUD(1-b) in category H. This indicates that these heuristic approaches significantly improve the probability of finding Ashenhurst decompositions over random or pseudo random partitioning.

2) Another observation that is made is that for the given set of benchmarks, both RGP and DEMAIN approaches were able to find an Ashenhurst decomposition within the first four partitions tried in about 80% of the cases.

For more details on partitioning algorithm RGP, see paper by Gatlin[22]. For more

details on partitioning algorithm in program DEMAIN, see paper by Luba[28] and Selvaraj[58]. For more details on partitioning algorithms in program GUD, see documentation for program MULTIS/GUD[52].

7.5 Additional Comparisons Between Decomposition Programs

In this section, more recent results of program GUD(binary version) are compared with several other programs. With the exception of program GUD, all programs compared in Table 7.16 and Table 7.17 are separate programs which are not called from within program MULTIS. Results for programs MISII and DSGN174 were obtained from a publication by Steinbach[62]. Other results were obtained using PSU programs.

Table 7.16: Results for *DFC* on MCNC Benchmarks

PROGRAM				GUD	TRADE	MISII	DSGN174	GUD_MV
File	in	out	cubes	DFC	DFC	DFC	DFC	DFC
5xp1	7	10	143	296	496	384	292	202
9sym	9	1	158	176	640	984	400	112
b12	15	9	72	300	412	[1]	[1]	248
bw	5	28	97	752	1148	[1]	[1]	570
con1	7	2	18	68	80	68	60	76
duke2	22	29	406	[3]	6516	2428	2200	2970
ex5p	8	63	214	2172	[4]	3720	1560	2068
misex1	8	7	40	284	472	208	224	256
misex2	25	18	101	600	548	464	436	484
misex3c	14	14	837	[2]	19816	4204	3028	1588
rd53	5	3	63	84	120	96	84	56
rd73	7	3	274	164	320	352	256	120
rd84	8	4	515	260	508	672	320	176
sao2	10	4	133	792	1848	516	468	436
squar5	5	8	56	160	228	[1]	[1]	152
xor5	5	1	32	16	16	[1]	[1]	16
Z5xp1	7	10	141	320	540	[1]	[1]	[1]
9symml	9	1	159	176	644	908	796	[1]
adr2	4	3	24	32	36	[1]	[1]	[1]
c8	28	18	166	408	552	[1]	[1]	[1]
cc	21	20	96	284	256	[1]	[1]	[1]
f51m	8	8	154	268	372	392	240	[1]
root	8	5	256	572	1000	[1]	[1]	[1]

From the results shown in these tables, it is clear that GUD_MV outperforms all the other decomposers in terms of DFC. Times shown are user times. While the results for program GUD have improved since early testing in terms of DFC and time, the performance of GUD still falls short of all other decomposers in terms of execution times. In terms of DFC, GUD and DSGN174 are roughly equal among the decomposers compared, second only to GUD_MV. As noted in the introduc-

tion to program MULTIS/GUD, program GUD does not use any sophisticated approaches to partitioning and encoding. The results for GUD in Table 7.16 and Table 7.16 were obtained using partitioning and encoding approaches which are, for all practical purposes, pseudo random.

The following is relevant information corresponding to Table 7.16 and Table 7.17.

```
Benchmarks run on SPARC-10 workstations. Times listed are user times.

[1]-Results not available for this benchmark
[2]-Out of Memory
[3]-Program still running after 10,000 seconds
[4]-Program limited to 32 input and 32 output variables

GUD     - Program designed at PSU by the POLO group
TRADE   - Program designed at PSU by Wei Wan
MISII   - Program designed at UC Berkeley
DSGN174 - Program designed by B. Steinbach
GUD_MV  - Program designed at PSU by Stan Grygiel
```

Table 7.17: Results for user time on MCNC Benchmarks

Program				GUD	TRADE	MISII	DSGN174	GUD_MV
File	in	out	cubes	time [s]	time [s]	time [s]	time [s]	time [s]
5xp1	7	10	143	56.4	2.3	4.2	2.7	10.6
9sym	9	1	158	289.0	29.9	17.8	8.7	31.2
b12	15	9	72	53.2	3.6	[1]	[1]	10.9
bw	5	28	97	63.8	9.5	[1]	[1]	18.2
con1	7	2	18	5.0	0.8	0.3	0.4	2.5
duke2	22	29	406	[3]	301.8	36.9	113.2	3134.0
ex5p	8	63	214	780.7	[4]	18.1	4126.7	183.3
misex1	8	7	40	26.9	4.1	1.4	1.9	8.5
misex2	25	18	101	839.5	11.0	2.4	14.4	905.0
misex3c	14	14	837	[2]	7361.6	5499.7	196.6	1482.3
rd53	5	3	63	8.0	1.2	1.3	0.5	1.8
rd73	7	3	274	85.7	2.5	14.1	3.5	12.1
rd84	8	4	515	280.7	32.6	119.7	7.3	30.6
sao2	10	4	133	663.5	31.5	9.6	13.9	48.6
squar5	5	8	56	10.9	2.1	[1]	[1]	4.5
xor5	5	1	32	0.6	0.1	[1]	[1]	0.4
Z5xp1	7	10	141	57.6	2.4	[1]	[1]	[1]
9symml	9	1	159	300.6	29.5	9.5	19.0	[1]
adr2	4	3	24	0.4	0.2	[1]	[1]	[1]
c8	28	18	166	140.4	4.5	[1]	[1]	[1]
cc	21	20	96	35.6	3.0	[1]	[1]	[1]
f51m	8	8	154	49.0	3.2	4.4	1.9	[1]
root	8	5	256	242.2	21.7	[1]	[1]	[1]

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

Presented in this thesis were three algorithms designed to improve different phases in the decomposition process of an Ashenhurst-Curtis style functional decomposition program. These algorithms were designed for column compatibility checking, column minimization for Ashenhurst type decompositions, and column-based input/output encoding.

The primary objective of the **GCA** algorithm was targeted at improving efficiency of column compatibility checking. The **GCA** algorithm creates the same data as the classical approach(**PCA**), but more efficiently in terms of execution time. Results for the **GCA** algorithm demonstrated a reduction in the execution time by an impressive margin over the classical approach. In some cases, the execution time was reduced by more than two orders of magnitude(x100 or 99% reduction). In addition to reducing execution time, the **GCA** approach, unlike the **PCA** approach, is able to create the compatibility graph for large bound sets in about the same amount of time as for small bound sets. By allowing large bound sets to be checked very efficiently, the **GCA** approach allows a larger portion of the search space to be checked in a reasonable amount of time. It is not known

for sure whether this added capability will lead to higher quality decompositions. However, this added capability provides the potential for improvement in DFC. Future work should include testing the **GCA** approach to determine how often lower DFC values are obtained by using large bound sets as opposed to small bound sets. Knowing this information may lead to more effective general decomposition strategies.

In addition to improving the efficiency of column compatibility checking, the large data sets created in the process of column compatibility checking by the **GCA** approach may be used by a modified graph coloring algorithm to efficiently check for Ashenhurst decompositions. Though the modified graph coloring algorithm presented(**MGCA**) was not implemented or tested, there are indications that it has potential for significant savings in execution time when bound sets are larger than the free sets. There are two indicators of this potential. The first indicator comes from examples done by hand. The other indicator is the fact that, by using the large data sets(sets of columns) created by the **GCA** approach, the number of nodes and edges in the newly created **MIG** graph are much fewer than are found in the corresponding **CIG** graph. Future work should include testing to determine if and when there is a savings in execution time using the **MGCA** approach.

Yet another algorithm presented in this thesis was a high quality column-based input/output encoding approach(**DC_ENC**). This approach is a heuristic encoding approach, designed to provide high quality encoding and still be very fast.

Examples shown, illustrated that Don't Cares are introduced into the codes of the predecessor sub-function without changing the successor sub-function in a Curtis style decomposition. This in itself was a proof that using combined class codes can reduce the complexity of the predecessor sub-function in some cases. Stated another way, if you have a fully specified sub-function and you are able to change specified values to Don't Cares without changing anything else, then the resulting sub-function is *at least* as simple as the fully specified sub-function with no Don't Cares. Future work should include testing to determine how often lower DFC values are obtained by using the **DC_ENC** encoding approach and under what conditions. The primary conditions to consider are the overlap ratio and the size of bound sets relative to the size of the free sets. Knowing these conditions would give a better indication of when **DC_ENC** is effective and when it is not effective.

The overall goal of this thesis was to present algorithms which can be used to improve the efficiency and quality of multi-level decompositions. In doing so, it is hoped that a Curtis style functional decomposition program is a viable candidate for use in the synthesis of a variety of technologies. While Curtis style functional decomposition can be used in several applications, it has not yet met the requirements of industry and is therefore not considered seriously as an alternative to existing methods in the synthesis of circuits to various technologies. Circuit applications where it has been used, at least experimentally, are FPGAs and CPLDs. Up to and including the present, there has been little interest in Curtis style func-

tional decomposition programs because they have either produced poor quality solutions, been too slow, used too much memory, and/or were not easily modified to meet industry specifications. However, new advances in research are steadily chipping away at these obstacles. In fact, all but the last problem were improved to a level of acceptability for most cases tested and from results available for comparison. Unfortunately, application specific requirements vary widely and access to such requirements is not readily attainable unless working directly with or for a vendor. Also, there is the problem of different CAD development packages and development packages which may have been customized to meet particular vendor requirements. For instance, a vendor may choose to customize a development package, for in-house use, by providing additional feedback mechanisms in different stages of a design process. This would create additional compatibility issues to be resolved if a new method were to be practical to be introduced for one or more technologies. However, these kinds of issues are expected when new software is considered as an add-on or a for replacement of existing software modules. Regardless whether or not a synthesis approach using some variant of Curtis style functional decomposition is acceptable for any of the existing technologies, there may be emerging technologies ideally suited for its use.

Already, Curtis style functional decomposition is used experimentally by Wright Patterson Air Force Base and by AbTech corporation for the purpose of pattern recognition. Unfortunately, specific details of their application is not readily ob-

tained. More investigation into specific technologies is required to determine exactly what other technologies that functional decomposition are applicable for. Therefore, one of the highest priorities for future research is to determine a beneficial and cost effective application for a Curtis style functional decomposition program. This may involve analyzing numerous technologies and comparing actual results from currently existing methods of decomposition against the best Curtis style functional decomposition program available.

BIBLIOGRAPHY

[1] R. Almeria, "The Encoding Problem," *PSU report* May 1995.

[2] P. Ashar, S. Devadas, A. Newton, "Sequential Logic Synthesis," Kluwer Academic Publishers. 1992.

[3] R. L. Ashenhurst. "The Decomposition of Switching Functions." *Proc. of the International Conference on Computer-Aided Design*, pp. 84-87, November 1959.

[4] K.A. Bartlett, R.K. Brayton, G.D. Hachtel, R.M. Jacoby, C.R. Morrison, R.L. Rudell, A.L. Sangiovanni-Vincentelli, A.R. Wang, "Multilevel Logic Minimization Using Don't Cares," *IEEE Trans. on CAD,* Vol. 7, pp. 723-740, June 1988.

[5] D. Bochmann, F. Dresig, B. Steinbach, "A New Decomposition Method for Multilevel Circuit Design," *Proc. of the European Conference on Design Automation (EDAC),* Amsterdam, 1991.

[6] G. Boole, "An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities," *London*, 1854.

[7] R.K. Brayton, G.D. Hachtel, C. McMullen, A. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis," *Kluwer Academic Publishers,* Boston, 1984.

[8] R.K. Brayton, R. Rudell, A. Sangiovanni-Vncentelli, A.R. Wang, "MIS - a Multiple-Level Logic Minimization System," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 6. No. 6, pp. 1062-1081, 1987.

[9] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis," *Proceedings of the IEEE*, Vol. 78, No. 2, pp. 264-300, February 1990.

[10] M. A. Burns, "Decomposition of Boolean Functions with comparisons between Graph Coloring and Set Covering Approaches with emphasis on Solving Column Compatibility," *PSU report*, April 1995.

[11] M. A. Burns, "Input-Output Encoding combined with Column Minimization," *PSU report*, May 1995.

[12] M. Craven, J. Shavlik, "Machine Learning Approaches to Gene Recognition," *University of Wisconsin Report, unpublished,* March 1994.

[13] H.A. Curtis. "A New Approach to the Design of Switching Circuits," D. Van Nostrand Company, 1962.

[14] M. J. Ciesielski, S. Yang, and M. Perkowski, "Multiple-Valued Minimization Based on Graph Coloring," *Proc. ICCD'89,* pp. 262 - 265, October 1989.

[15] M. Ciesielski, J. Shen, "A Unified Approach to Input-Output Encoding for FSM State Assignment," *IEEE Design Automation Conf.*, pp. 176-181, 1991.

[16] S. Devadas, A.R. Wang, A.R. Newton, A. Sangiovanni-Vincentelli, "Boolean Decomposition in Multi-Level Logic Optimization," *Proc. IEEE International Conf. on Computer-Aided Design,* pp. 290-293, 1988.

[17] S. Devadas, H-K. T. MA, A.R. Newton, A. Sangiovanni-Vincentelli, "MUSTANG: State assignment of finite state machines targeting multi-level logic implementations," *Proc. IEEE International Conf. on Computer-Aided Design,* vol. 7, pp. 1290-1300, December 1988.

[18] X. Du, G. Hatchtel, B.Lin, and A.R. Newton,"MUSE: A MUltilevel Symbolic Encoding Algorithm for State assignment," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 10, pp. 28-38, 1991.

[19] B.J. Falkowski, and M. Perkowski, "Algorithm for the Generation of Disjoint Cubes for Completely and Incompletely Specified Boolean Functions," *Intern. Journal of Electronics,* Vol. 70, No. 3, pp. 533 - 538, March 1991.

[20] C. Files, "A New Approach to Partition Selection in Ashenhurst-Curtis Functional Decomposition," Masters thesis, University of Idaho, Moscow ID, August 1995.

[21] C. Files, "Using a Search Heuristic in an NP-Complete Problem in Ashebhurst-Curtis Decomposition," Final Report for Graduate Summer Research Program, Wright Laboratory, Sponsored by Air Force Office of Scientific Research, Bolling Air Force Base, DC and Wright Laboratory, August 1994.

[22] R. Gatlin, "Variable Partitioning Based on Table Compare," *PSU report*, December 1995.

[23] J. Goldman, "Pattern Theoretic Knowledge Discovery," *Wright-Laboratory Report, WL/AART-2, Wright Patterson Air Force Base,* August 1994.

[24] N. Iliev, "Constrained Input Encoding: Application to LUT-based Functional Decomposition," *PSU report* May 1995.

[25] R.M. Karp, F.E. McFarlin, J.P. Roth, J.R. Wilts, "A Computer Program for the Synthesis of Combinational Switching Circuits," *In Proc. AIEE Annual Symposium on Switching Circuits Theory,* pp. 182-194, 1961.

[26] R.M. Karp, "Functional Decomposition and Switching Circuit Design," *J. Soc. Industr. Appl. Math.*, Vol. 11, No. 2, pp. 291-335, June 1963.

[27] B. Lin, A.R. Newton. "Synthesis of Multiple-Level Logic From Symbolic High-Level Description Languages," *IFIP International Conf. on Very Large Scale Integration,* pp. 187-196, August 1989.

[28] T. Luba, J. Kalinowski, K. Jasinski, A. Krasniewski, "Combining Serial Decomposition with Topological Partitioning for Effective Multi-Level PLA Implementations", In *"Logic and Architecture Synthesis,"*, 1991. P. Michel and G. Saucier, (Editors), Elsevier Science Publisher B.V. (North Holland), pp. 243-252, 1991.

[29] T. Luba, H. Selvaraj, A. Krasniewski, "A New Approach to FPGA-based Logic Synthesis", *Workshop on Design Methodologies for Microelectronics and Signal Processing,* Gliwice-Cracow, 1993.

[30] T. Luba, M. Mochocki, J. Rybnik, "Decomposition of Information Systems Using Decision Tables," *Bulletin of the Polish Academy of Sciences, Technical Sciences,* Vol. 41, N o.3, 1993.

[31] T. Luba, R. Lasocki, "Decomposition of Multiple-valued Boolean Functions," *Applied Mathematics and Computer Science,* Vol.4, No.1, pp. 125-138, 1994.

[32] T. Luba, "Multiple-Valued and Multiple-Level Decomposition and Its Applications in Information Systems Analysis and Logic Synthesis," *Report on Window-on-Science Project WOS-95-2155*, June 1995.

[33] T. Luba, M. Nowicka, "Balanced Multi-Level Decomposition and Its Applications in FPGA-based Synthesis," *Report*, Warsaw University of Technology, 1995.

[34] Y.T. Lai, M. Pedram, S. Vrudhula, "BDD-based Logic Decomposition: Theory." *Technical Report,* Dept. of EE. Systems, University of Southern California, 1992.

[35] D. Mitchie, "Problem of Decomposition and the Learning of Skills," *Proceedings of the European Conference on Machine Learning, Lecture Notes in Artificial Intelligence,* v. 912, pp. 17-31, 1995.

[36] R. Murgai, N. Shenoy, R.K. Brayton, A. Sangiovanni-Vincentelli, "Improved Logic Synthesis Algorithm for Table Look Up Architectures," *Proc. IEEE Intern. Conf. on Computer Aided Design,* pp. 564-567, 1991.

[37] R. Murgai, R. Brayton, A. Sangiovanni-Vincentelli, "Optimum Functional Decomposition Using Encoding," *IEEE Design Automation Conference,* p. 408-414, 1994.

[38] M. Perkowski, J. Brown, "A Unified Approach to Designs with Multiplexers and to the Decomposition of Boolean Functions," *Proc. ASEE Annual Conference,* pp.1610-1619, 1988.

[39] M. Perkowski, "Synthesis of Multioutput Three Level NAND Networks," *Proc. of the Seminar on Computer Aided Design,* IFAC - Intern. Federation of Control, pp. 238 - 265, Budapest, Hungary, 3-5 November 1976.

[40] M. A. Perkowski, I. Schaefer, A. Sarabi, and M. Chrzanowska-Jeske, "Multi-Level Logic Synthesis Based on Kronecker Decision Diagrams and Boolean Ternary Decision Diagrams for Incompletely Specified Functions," *Journal on VLSI Design,* November 1995.

[41] M.A. Perkowski, Ph. Ho, "Free Kronecker Decision Diagrams and their Application to Atmel 6000 Series FPGA Mapping," *Proc. Euro-DAC'94*, Grenoble, Sept. 19-23, 1994, France.

[42] M. Perkowski, N. Nguyen, "Minimization of Finite State Machines in System SuperPeg," *Proc. of the Midwest Symposium on Circuits and Systems,* pp. 139 - 147, Luisville, Kentucky, 22-24 August 1985.

[43] M. Perkowski, H. Uong, "Generalized Decomposition of Incompletely Specified Multioutput, Multi-Valued Boolean Functions," *Unpublished manuscript, Department of Electrical Engineering,* PSU 1987.

[44] M.A. Perkowski, P. Wu, and K.A. Pirkl, "KUAI-EXACT: A New Approach for Multi-Valued Logic Minimization in VLSI Synthesis," *Proc. 1989 IEEE International Symposium on Circuits and Systems*, pp. 401– 404, 1989.

[45] M.A. Perkowski, P. Dysko, and B.J. Falkowski, "Two Learning Methods for a Tree-Search Combinatorial Optimizer," *Proc. of IEEE Int. Conf. on Comput. and Comm.,* pp. 606-613, Scottsdale, Arizona, 1990.

[46] M.A. Perkowski, J. Liu, J.E. Brown, "Rapid Software Prototyping: CAD Design of Digital CAD Algorithms," In G. W. Zobrist (ed), *Progress in Computer-Aided VLSI Design,* Vol. 1, pp. 353-401, 1989.

[47] M. Perkowski, L. Csanky, A. Sarabi, I. Schaefer, "Fast Minimization of Mixed-Polarity AND/XOR Canonical Forms", *Proc. of the IEEE Intern. Conf. on Comp. Design,* ICCD'92, Boston, October 11-13, 1992, pp. 32-36.

[48] M. A. Perkowski, T. Ross, D. Gadd, J.A. Goldman, and N. Song, "Application of ESOP Minimization in Machine Learning and Knowledge Discovery," *Proc. of the Second Workshop on Applications of Reed-Muller Expansion in Circuit Design*, Chiba, Japan, pp. 102-109, August 1995.

[49] M. Perkowski, T. Luba , S. Grygiel, P. Burkey, M. Burns, N. Iliev, M. Kolsteren, R. Lisanke, R. Malvi, Z. Wang, H. Wu, F. Yang, S. Zhou, and J. Zhang, "Unified Approach to Functional Decompositions of Switching Functions," *Report submitted to AbTech Corporation*, unpublished, Version IV, December 1995.

[50] Marek Perkowski, Michael Burns, Ralph Almeria, Nick Iliev, "Approaches to the Input-Output Encoding Problem in Boolean Decomposition", *Report submitted to AbTech Corporation*, unpublished, January 1996.

[51] M. Perkowski, M. Burns, T. Luba, S. Grygiel, C. Stanley, R. Price, Z. Wang, J. Lu, P. Burkey, D. Manoharan, and Sanof Mohammad, "Development of Search Strategies for MULTIS," *Report submitted to AbTech Corporation*, unpublished, December 1995.

[52] Marek Perkowski, Michael Burns, Paul Burkey, Rahul Malvi, Stanislaw Grygiel, Roger Shipman, Nick Iliev, Jin Zhang, Zhi Wang, Jing Lu, Robert Gatlin "Documentation of MULTI-Strategy decomposer(MULTIS)," *Report submitted to AbTech Corporation*, unpublished, Version IV, January 1996.

[53] M. Perkowski, M. Marek-Sadowska, T. Luba, S. Grygiel, P. Burkey, R. Malvi, Z. Wang, J.S. Zhang, and C. Stanley, "Cube Diagram Bundles: A New Representation of Multi-Valued Relations," *PSU Report 1996.*

[54] M. Perkowski, M. Marek-Sadowska, T. Luba, S. Grygiel, P. Burkey, R. Malvi, Z. Wang, J.S. Zhang, and C. Stanley, "Fundamental Operations on Strongly Unspecified Multi-Valued Functions and Relations," *PSU Report 1996.*

[55] M. Perkowski, M. Marek-Sadowska, T. Luba, S. Grygiel, P. Burkey, R. Malvi, Z. Wang, J.S. Zhang, and C. Stanley, "GUD-MV: Multi-Level Decomposition of Multi-Valued Functions and Relations," *PSU Report 1996.*

[56] J.P. Roth, "Minimization over Boolean Trees," *IBM Journal.* 4, 5, pp. 543 -555, 1960.

[57] A. Saldanha, T. Villa, R. Brayton, A. Vincentelli, "A Framework for Satisfying Input and Output Encoding Constraints," *IEEE Design Automation Conference,* p. 170-175, 1991.

[58] H.S. Selvaraj, Ph.D. Dissertation, "FPGA-Based Logic Synthesis," Warsaw University of Technology, 1994.

[59] R.C. Shipman, "Generalized Functional Decomposition Using Graph Coloring: A comparison of an exact with a heuristic algorithm," Honors Thesis, 1995.

[60] B. Steinbach, "XBOOLE - A Toolbox for Modeling, Simulation and Analysis of Large Digital Systems," *System Analysis and Modelling, Gordon and Breach Science Publishers,* Vol. 9, No. 4, pp. 297-312, 1992.

[61] B. Steinbach, M. Stoeckert, "Design of Fully Testable Circuits by Functional Decomposition and Implicit Test Pattern Generation," *Proc. 1994 IEEE Test Conference,* 1994.

[62] B. Steinbach, A. Wereszynski, "Synthesis of Multi-Level Circuits Using EXOR-Gates," *Proceedings IFIP, WG 10.5, Workshop on Applications of the Reed-Muller Expansion in Circuit Reed-Muller* Chiba, Japan, August 1995.

[63] J.P. Roth, R.M. Karp, "Minimization Over Boolean Graphs," *IBM Journal Res. and Develop.* No.4, pp. 227-238 (543-558), April 1962.

[64] C. Vogt, "The Combinatorics of Function Decomposition and Applications of Learning Theory," *UC Sandiego Report, unpublished,* 1992.

[65] W. Wan, Masters thesis, "A New Approach to the Decomposition of Incompletely Specified Functions Based on Graph Coloring and Local Transformations and Its Application to FPGA Mapping," Portland State University. 1992

[66] W. Wan, M.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. of the IEEE EURO-DAC '92, European Design Automation Conference,* Sept. 7-10, Hamburg, 1992, pp. 230 - 235.