11-9-1993

Investigation of Solution Space of Trees and DAGs for Realization of Combinational Logic in AT 6000 series FPGAs

Philip Ho
Portland State University

THESIS APPROVAL

The abstract and thesis of Philip Ho for the Master of Science in Electrical and Computer Engineering were presented November 9, 1993, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS: [redacted]
Marek A. Perkowski, Chair

[redacted]
Malgorzata Chrzanowska-Jeske

[redacted]
Maria E. Balogh
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL: [redacted]
Rolf Schaumann, Chair
Department of Electrical Engineering

* *

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by [redacted] on 17 June 1994

ABSTRACT

An Abstract of the thesis of Philip Ho for the Master of Science in Electrical and Computer Engineering presented November 5, 1993.

Title: Investigation of Solution Space of Trees and DAGs for Realization of Combinational Logic in AT 6000 series FPGAs.

Various tree and Directed Acyclic Graph structures have been used for representation and manipulation of switching functions. Among these structures the Binary Decision Diagrams have been the most widely used in logic synthesis. A BDD is a binary tree graph that represents the recursive execution of Shannon's expansion. A FDD is a directed function graph that represents the recursive execution of Reed Muller expansion.

A family of decision diagrams for representation of Boolean function is introduced in this thesis. This family of Kronecker Functional Decision Diagrams (KFDD) includes the Binary Decision Diagrams (BDD) and Functional Decision Diagrams (FDD) as subsets. Due to this property, KFDDs can provide a more compact representation of the functions than either of the two above-mentioned decision diagrams.

The new notion of permuted KFDD is introduced to generate a compact circuit in FPGAs to represent a switching function. A permuted tree search is a free search method which is not limited by the order of variable and the expansion tree as in the cases of KFDD, BDD and FDD.

A family of decision diagrams and the theory developed for them are presented in

this thesis. The family of permuted Kronecker Functional Decision Diagrams includes BDDs and FDDs as subsets is incorporated into program RESPER. Due to this property, permuted KFDD can provide a more compact circuit realization in the multi-level circuit. The circuit obtained can be realized directly with FPGAs like AT 6000 series from Atmel. This algorithm is implemented on several MCNC benchmarks, the results compared with previous programs, TECHMAP and REMIT, are very encouraging.

The main achievement of this thesis is the creation of the algorithm which applies a permuted tree search method combined with Davio Expansion and generates Directed Acyclic Graph which is next mapped to a compact circuit realization.

INVESTIGATION OF SOLUTION SPACE OF TREES AND DAGS FOR REALIZATION OF COMBINATIONAL LOGIC IN AT 6000 SERIES FPGAs

by

PHILIP HO

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

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1993

ACKNOWLEDGEMENTS

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

I would like to thank Marek A. Perkowski, Chair of the Committee, for his initial inspiration and professional guidance throughout the research. He trained me to work independently, to be a better programmer and also to be a better researcher.

I am deeply appreciative of the patience, and encouragement of my father and mother. Without their financial support, love, and faith I would not be able to complete this project.

I am deeply appreciative of the patience, encouragement and companionship of my fiance, Melisa. Without her love and understanding, I would give up on this study.

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

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Digital logic designers have the opportunity to design a circuit from Standard SSI/MSI devices, Standard LSI/VLSI devices, Gate array devices, Standard-cell devices, Full-custom devices and Programmable Logic Devices. Among all these, PLDs are the most commonly used. It is a digital IC capable of being programmed to provide a variety of different logical functions. Its turn around time is very short, ranging from a few minutes to a few hours, which significantly reduces the time of a product to market. Because of these, the PLD technology receives a great deal of popularity in the electronic industry.

PLDs were first introduced by Harris in 1970. The type introduced were the Programmable Read-Only Memory, which had a structure of a fixed AND array followed by a programmable OR array. This type of AND/OR structures began to dominate the PLD technology. About the late 70s and early 80s, the Programmable Array Logic device (PAL) based on PLD architecture was invented at Monolithic Memories. It consists of a programmable AND array followed by a fixed OR array. In 1985, Xilinx introduced its Logic Cell Array, now called a Field Programmable Gate Array (FPGA), which had a very different achitecture from the AND/OR structure. It consisted of a matrix of Configurable Logic Blocks (CLB) surrounded by a ring of Input/Output interface Blocks, and an interconnect network for connecting the CLB blocks. Each CLB is capable of implementing an arbitrary Boolean function of its input variables. This new design style started to create a new wave of the PLD industry. Actel later introduced multiplexer-based cell in the basic logic block instead of the Lookup Table approach of Xilinx. Recently, Concurrent Logic (now part of Atmel) introduced another kind of FPGA structure, which contains an array of small-size cells. Each cell contains the most commonly used simple logic and wiring functions. Synthesis for this type of structure~~s~~ is the subject of this thesis.

FPGAs can be used in almost all of the applications that currently use PLDs and small scale integration (SSI) logic chips. FPGAs are a completely general medium for implementing digital logic. They are particularly suited for implementation of ASICs, such as an IBM PS/2 micro channel interface, a 1 megabit FIFO controller, etc. Random logic circuitry is usually implemented using PALs. If the speed of the circuit is not of ~~the~~ critical concern, such circuitry can be implemented advantageously with FPGAs as well. FPGAs are almost ideally suited for prototyping applications. The low cost of implementation and the short time needed to physically realize a given design, provide enormous advantages over more traditional approaches to build prototype hardware. A whole new class of computers has been made possible with the advent of in-circuit reprogrammable FPGAs. These machines consist of a board of such FPGAs, usually with the pins of neighboring chips connected. The idea is that a software program can be "compiled" into hardware rather than software. This hardware is then implemented by programming the board of FPGAs. Algotronix Ltd. sells a small add-on board for IBM PCs that can perform this function. At the research level, the Digital Equipment Corporation in Paris[29] has achieved performance ranging from 25 billion operations per second up to 264 billion operation per second on applications such as RSA cryptography, the discrete cosine transform, Ziv-Lempel encoding and 2D convolution. FPGAs are also attractive when it is desirable to change the structure of a given machine that is already in operation.

Since there have been a tremendous efforts to develop different types of digital

ICs, sophisticated logic design tools are required to allow for their fast prototyping. Most of the well-known logic design tools like ESPRESSO[26] and PALMINI[27] concentrated on fast two level AND/OR logic minimization. However, multi-level circuits have a smaller and often faster realizations for most logic functions, synthesis tools started to emerge for minimization of the circuit area in multi-level realization. Synthesis tools like MISII[28] are now the cores in the industrial Computer Aided logic design systems.

The logic synthesis methods developed for FPGAs have been based on algebraic decomposition (factorization) methods[25]. However, it is known that logic synthesis methods based on Boolean decomposition methods can produce better results[24]. Moreover, those core CAD tools have been based on the "unate paradigm". The "unate paradigm" is the assumption that most of the logic functions occuring in logic design are unate or nearly unate. The meaning of "unate" or "nearly unate" for logic minimization purposes is, that the circuit realization of a nearly unate function with AND and OR gates is smaller in terms of the numbers of gates than that of a circuit using the AND and EXOR gates. On the other hand, the meaning of "linear" or "nearly linear" for logic minimization purposes is that the circuit realization of a nearly linear function with AND and EXOR gates is smaller in terms of the numbers of gates than that of a circuit using the AND and OR gates. Arithmetic function like counters, adders, multipliers, signal processing functions and error correcting logic belong to the class of nearly linear functions. Thus those functions will have a smaller circuit realization if the EXOR gate is incorporated into the design.

The synthesis incorporating the EXOR gate has been neglected because the EXOR gate was perceived to be slower and having a larger circuit area. However, those upcoming FPGAs from Xilinx, Actel, and Atmel allow the implementation of the EXOR gate without any speed or circuit size penalty in comparison to the AND and OR gates. Since the Atmel cells can realize the set of functions used in Functional Decision Diagrams and Reed Muller trees, these expansions can be easily applied to this type of FPGAs. The basic cell of the AT 6000 series can be programmed to one-bit multiplexer and the three-input AND/EXOR cell. This suggests using these cells for a tree-like expansions such as GRM trees and RM trees. Papers [6], [3], and [7] use EXOR gates to minimize the multi-level circuits.

The initial phase of many logic synthesis systems, such as MISII and BOLD[30], restructures the original network to reduce a cost function that is calculated directly from the network itself. The intention is to improve the final circuit by reducing the complexity of the network. In this phase, the method does not consider the type of element that will be used for the final circuit. After the initial phase which produces the optimized network, the technology mapping stage transforms this network into the final circuit. This is done by selecting pieces of the network that can be implemented by one of the available circuit elements and specifying how these elements are to be interconnected. The circuit is optimized to reduce a cost function that typically incorporates measures of both the area and delay.

This thesis introduces an algorithm to produce a more compact circuit realization as a multi-level circuit. To accomplish this goal, a Functional Decision Diagram which combines permuted tree search method with Davio Expansion using Directed Acyclic Graph is introduced. The obtained decision diagram is mapped into AT 6000 FPGA series resources. This algorithm is implemented in RESPER and the results are very encouraging. First, this thesis will look into the architecture of the AT 6000 FPGA series. It then introduces the family of multi-level expansions in Chapter 5. In Chapter 6, the full description of RESPER and step-by-step circuit realization with one of the MCNC benchmarks are given. Directed Acyclic Graph with negated edges is introduced in Chapter 6. Chapter 7 compares the results with several algorithms [3, 18] based on Davio Expansion which tend to map into Atmel FPGAs. Since not much research had been done related to mapping ESOP function and the internal operation of RESPER can accept ESOP function with some modification, a paper design will be presented using the algorithm from this thesis in chapter 8.

CHAPTER II

FIELD PROGRAMMABLE GATE ARRAY (FPGA)

Gate Arrays are an important branch of custom VLSI (Very Large Scale Integration). By 1990, it is estimated that more than half of all semiconductors sold will be semi custom designs of which gate arrays are a major part. Today, gate arrays outsell standard cell ICs by 4 to 1 margin. Over 70 vendors currently offer gate arrays and this number is constantly increasing. Gate arrays are semicustom digital integrated circuits, which are mostly made ahead of time and which are customized to the users' need by defining one or more layers of metal (via the appropriate mask) on the die itself.

Each of these techniques, full custom approach and semi-custom approach, requires an extensive manufacturing effort, taking several months from beginning to end. This results in a high cost for each unit unless large volumes are produced.

In the electronics industry, it is vital to reach the market with new products in the shortest possible times, so the reduction of the development and production times is essential. It is also important that the financial risks of developing a new product can be limited so that more new ideas can be prototyped. Field Programmable Gate Arrays (FPGAs) are a solution to these time-to-market and financial risk problems because they provide instant manufacturing and very low cost prototypes. A field programmable device is a device in which the final logic structure can be directly configured by the end user without the use of an IC fabrication facility.

II.1. EVOLUTION OF PROGRAMMABLE DEVICE

Programmable devices have long played a key role in the design of digital hardware. They are general purpose chips that can be configured for a wide variety of applications. There are several types of programmable devices such as fuse-link programmable, electrically programmable and software programmable. Examples of fuse-link programmable devices are: programmable read-only memories (PROMs), programmable array logics (PALs), and field programmable logic arrays (FPGAs). Examples of electrically programmable device are electrically erasable programmable read-only memories (EEPROMs) and EPROMs. An example of a device that can be software programmable to perform a logic function is a microprocessor.

PROMs are a viable alternative for realizing simple logic circuits, and its structure is well suited for implementing computer memories. Another type of device for implementing a logic circuit is PLD. It comprises an array of AND gates connected to an array of OR gates. The logic circuit implemented in a PLD is a Sum-Of-Products form. The most basic version of PLD is the PAL. It is a collection of pre-made logic functions on a chip. It consists of a programmable-AND followed by a fixed-OR plane. On a PAL, the interconnect lines among the logic functions are fixed in place. The users' only choice is to break or not break (via fusible link) a given line. A more flexible version of PAL is the PLA. PLAs also comprise an AND plane followed by an OR plane, but both planes are programmable. They are available in both mask programmable and field-programmable options. For mask programmable devices, it is made to the point where the metallization will define its function. For field programmable devices, its connections always involve some sort of programmable switch (such as a fuse). Although both types of PLDs allow high speed performance implementation of logic circuits, they can only implement small logic circuits that can be represented with a modest number of product terms.

The most general type of programmable devices consists of an array of uncommitted elements that can be interconnected according to a user's specifications. One of the classes of devices is known as the Mask Programmable Gate Arrays (MPGAs). It consists of rows of transistors that can be interconnected to implement a desired logic circuit. User specified connections are available both within the rows (to implement basic logic gates) and between the rows (to connect the basic gates). In MPGAs, all the mask layers that define the circuitry of the chip are pre-defined by the manufacturer except those that specify the final metal layers. These metal layers are customized to connect the transistors in the array to implement the desired circuit. The main advantage of MPGAs over PLDs is that they can implement a much larger circuit. The other class of devices is known as Field Programmable Gate Array (FPGA), which combines the programmablility of a PLD and the scalable interconnection structure of an MPGA. Like MPGAs, the FPGAs consist of an array of uncommitted elements that can be interconnected in a general way. Like PALs, the interconnections between the elements are user programmable. FPGAs were first introduced by Xilinx in 1985. Since then many different FPGAs have been developed by a number of companies: Actel, Algotronix, Altera, Atmel, among others.

II.2. IMPLEMENTATION PROCESS

A designer who wants to make good use of FPGAs must have access to an efficient CAD system. Figure 1 shows the steps involved in a typical CAD system for implementing a circuit in an FPGA.

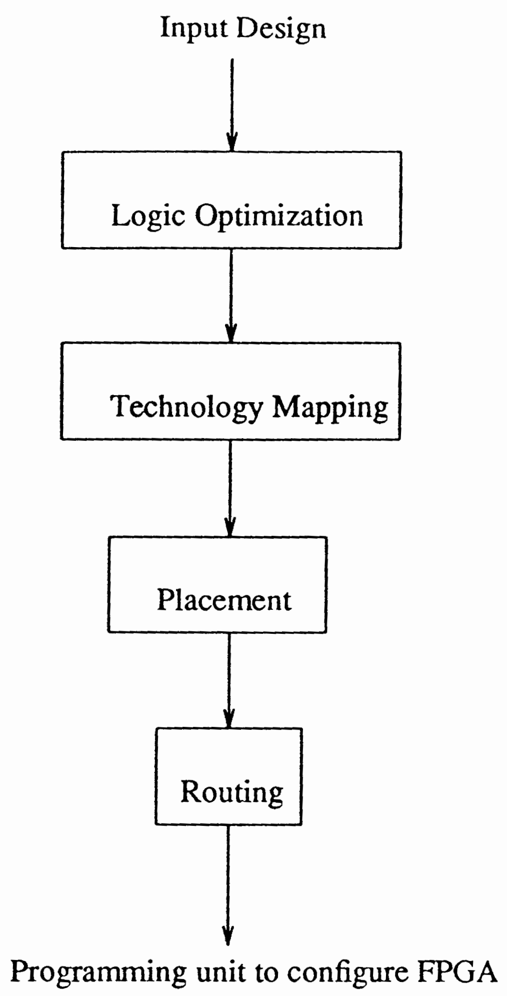


Figure 1. CAD system for FPGAs.

The starting point for the design process is the initial logic entry of the circuit that is to be implemented. The circuit description used in this thesis is disjoint ON cubes written in PLA format. This set of disjoint ON cubes is then processed by a logic optimizations tool. The goal is to optimize the area and speed of the final circuit. The optimized disjoint ON cubes are next transformed into a circuit using FPGA logic blocks. This is done by the technology mapping stage. Having mapped the circuit into logic blocks, it is necessary to decide where to place each block in the FPGA array. A placement program is used to solve this problem. The final step in the CAD system is performed by the routing software, which assigns the FPGA wire segments and chooses programmable switches to establish the required connections among the logic blocks. Upon the successful completion of the placement and routing, the CAD system's output is fed to a programming unit to configure the final FPGA chip.

This thesis involves only the logic optimization and technology mapping. The Davio Expansion is applied to decompose and optimize the boolean function, and transform it into a circuit of Atmel's FPGA logic blocks.

CHAPTER III

ARCHITECTURE OF AT 6000 SERIES

The AT 6000 series is a new generation of Field Programmable Gate Arrays introduced by Atmel. Its general architecture is based on an array of logic cells. In contrast to other FPGAs, like the Actel's Multiplexer Based or Xilinx's Table Look Up based approaches, the logic cells in the AT 6000 series can realize functions of only up to three input variables. Therefore, the architecture is also called "Fine Grain Cellular Array FPGAs". Because this thesis introduces the synthesis methods that are especially suited for the AT 6000 series, the basic features of this architecture will be reviewed in this chapter.

The AT 6000 series employs a patented, symmetrical architecture consisting of many small yet powerful logic cells connected to a flexible bussing network and surrounded by a programmable I/O. The Atmel's architecture was developed to provide the highest levels of performance, functional density and design flexiblility in an FPGA. The cells' small size allows for the realization of arrays with a large number of cells. For example, the AT 6000 has 6400 logic cells while the largest Xilinx chip has only 900 cells, so that the lower cell complexity is traded off for the larger number of cells. A simple, high speed bussing network offers fast, efficient communication over medium and long distances. Thus, the AT 6000 series provides the density and performance of custom gate arrays without the prototyping and debugging delays necessary for mask-programmed devices.

III.1. BUSING NETWORK

There are two kinds of buses: local and express. Local buses are the link between the array of cells and the bussing network. There are two local buses North-South 1 and North-South 2 for every column of cells, and two local buses East-West 1 and East-West 2 for every row of cells. Each local bus is connected to every cell in its column or row, thus providing every cell in the array with read/write access to two North-South and two East-West buses.

Each cell, in addition, provides the ability to make a 90 degree turn between either of the two North-South buses and either of the two East-West buses. Express buses are not connected directly to cells and, thus, provide the highest speeds. Each express bus is paired with a local bus. There is a connective unit, a repeater, spaced every eight cells, which serves to allow interchanges between local and express buses.

III.2. CELL STRUCTURE

The Atmel cell (Fig 4) is simple and small and yet can be programmed to perform all the logic and wiring functions needed to implement any digital circuit. Because its four sides are functionally identical, each cell is completely symmetrical.

In addition to the four local bus connections, a cell receives eight inputs and provides two outputs to its North, South, East, and West neighbors. These ten inputs and outputs are divided into two classes: A and B. There is an A input and a B input for each neighboring cell and a single A output and single B output driving all four neighbors. Between cells, an A output is always connected to an A input and a B output to a B input.

Within the cell, the four A inputs and the four B inputs enter two separate, independently configurable multiplexers. Cell flexibility is enhanced by allowing each multiplexer to select also the logical constant 1. The two multiplexer outputs enter the two upstream AND gates. The write access to the four local buses are controlled by the tri-state buffer.

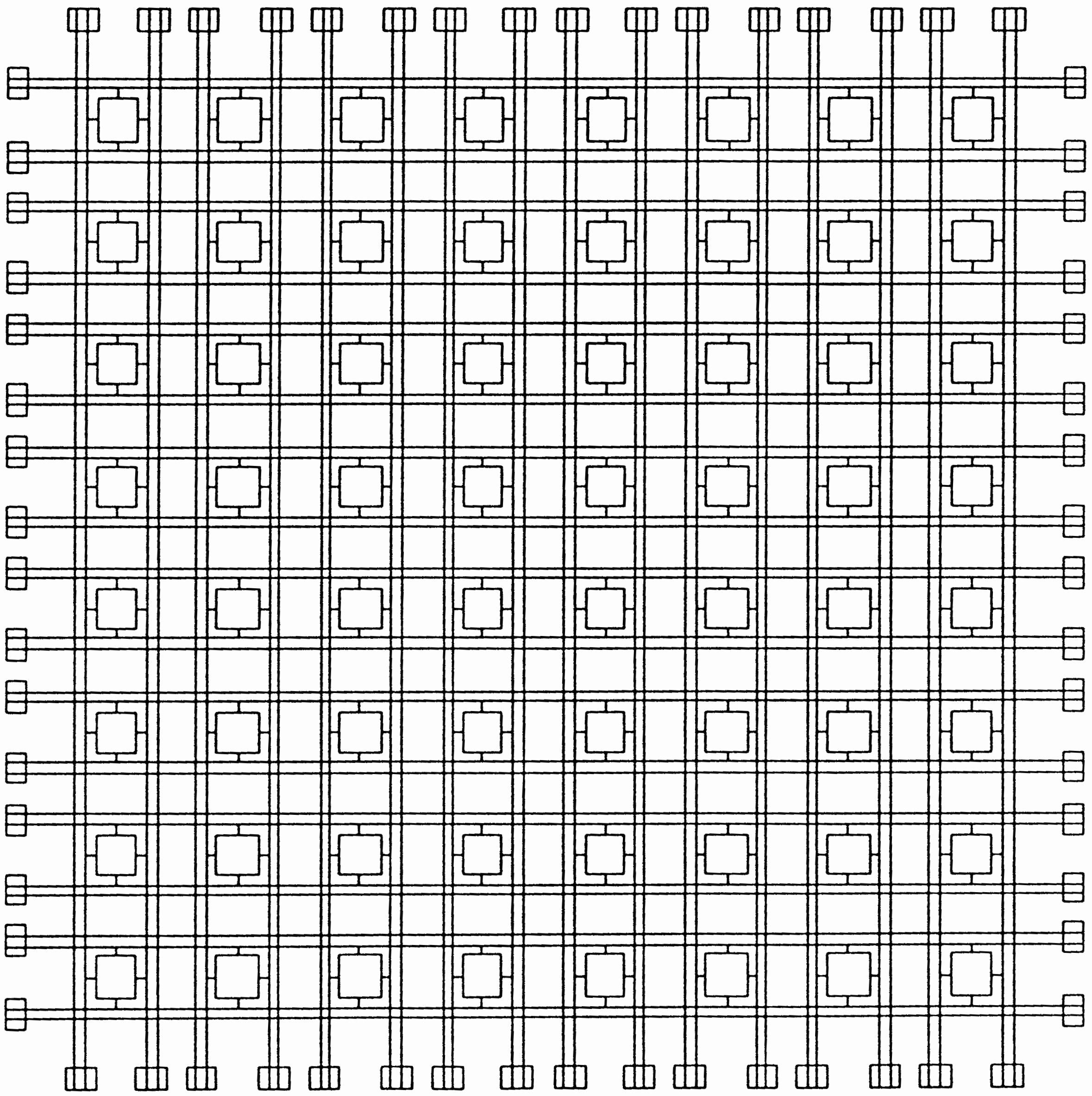


Figure 2. Busing Network.

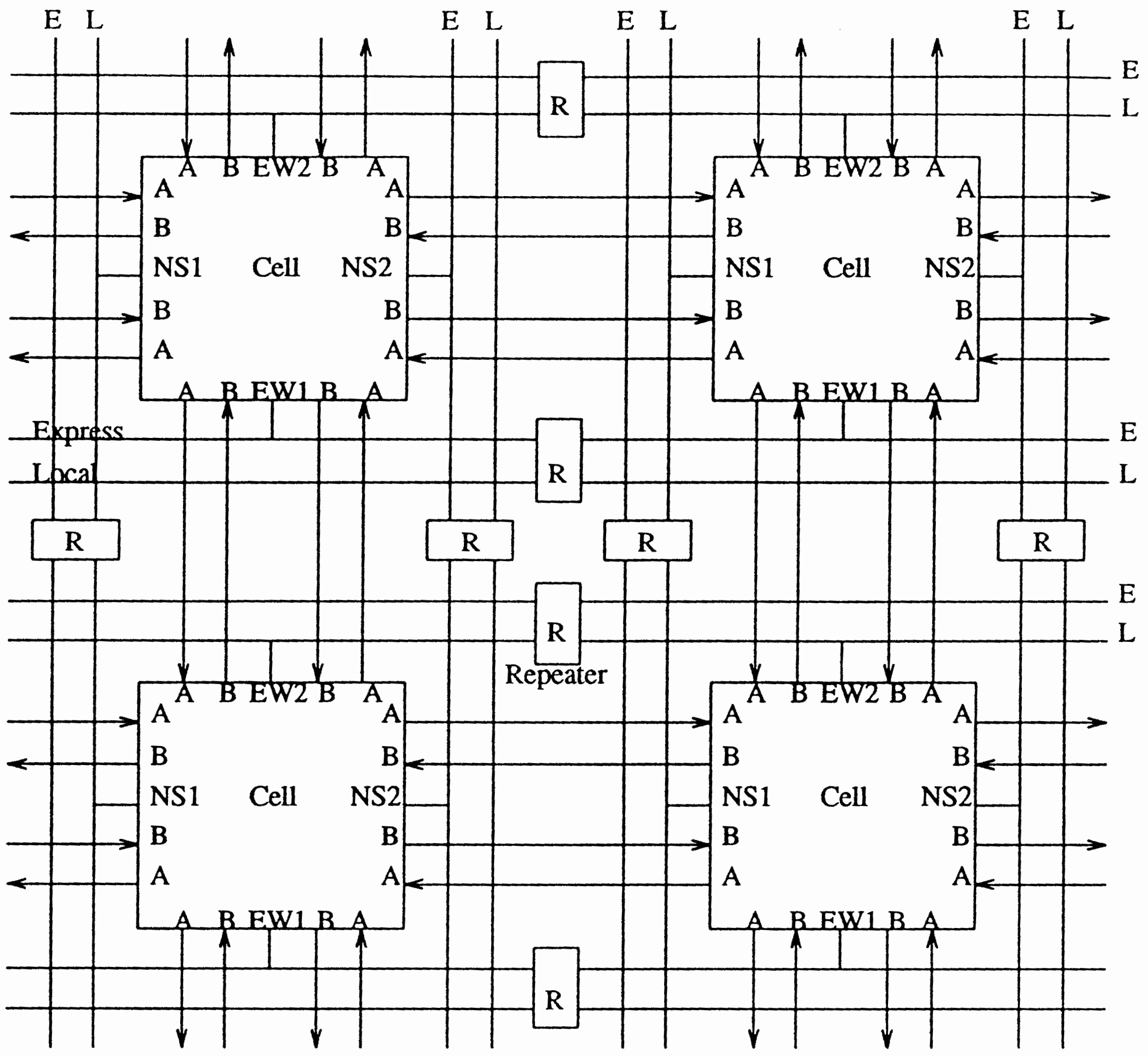


Figure 3. Cell to Cell and Cell to Bus Connections.

III.3. LOGIC STATES

The Atmel cell implements a rich and powerful set of logic functions, stemming from forty-four cell states. Some states use both A and B inputs. Other states are created by selecting the 1 input on either or both of the input to the multiplexer. There are twenty-four purely combinational states with a range of functions, including AND, OR, NAND, NOR and one-bit multiplexer. Five constant states that produce all combinations of constant values at two cells outputs as shown in Figure 5.

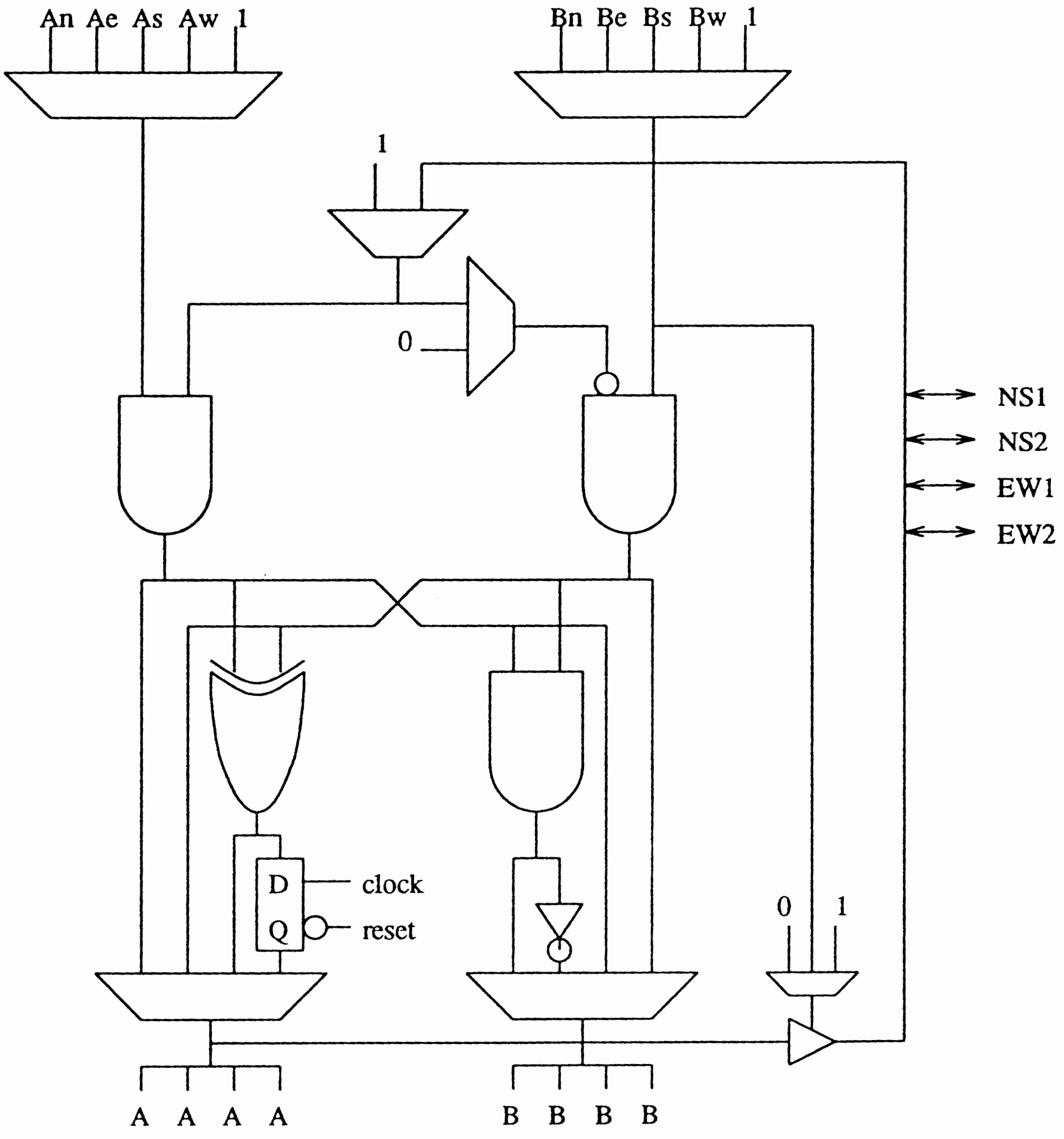
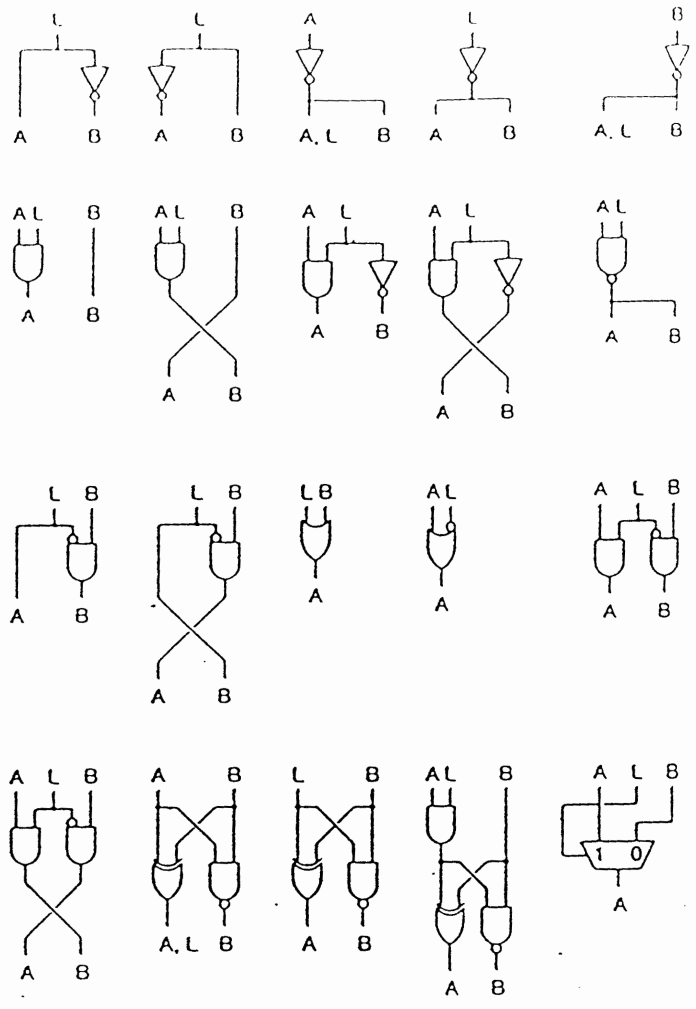
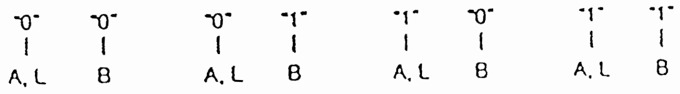


Figure 4. Cell Structure.


Combinatorial States



Constant States

Figure 5. Logic States.

CHAPTER IV

CURRENT RESEARCH VERSUS OUR APPROACH

Recently, each of these programs [3, 15, 23, 18] introduced some new multilevel formalization. In this thesis, the researcher puts together, compares and generalizes these formalisms. The program, RESPER, was developed and several comparisons were done.

IV.1. CURRENT RESEARCH

The REMIT program[3] starts from a completely specified Boolean function in the form of an array of ON disjoint cubes, and generates a permuted tree using Reed-Muller Expansion. The variable selection rules select the variable that occurs most often in disjoint cubes, one at a time.

RMS program[15] uses Reed-Muller Expansion to create a new efficient representation called Functional Decision Diagrams (FDDs). It starts with a two level SOP to calculate an order of variables in the FDD according to the most often used variables. The isomorphic subtrees are next reduced.

ASYL program[23] applies Shannon Expansion to build the BDD of each function. It also uses the Reduced Order Binary Decision Diagrams (ROBDDs) approach to minimize the area. Its target is on ACTEL's multiplexer-based Field Programmable Gate Arrays (FPGAs). Its heuristics to select the variable are the following:

1 Select a variable that appears in all product terms under the same polarity.

2 If a product term is restricted to a simple literal then select this literal.

3 If all the variables appear only once in a function then select the smallest product term.

4 Select the set of variables of maximum occurance.

TECHMAP program[18] uses the concepts of BDDs and FDDs applied to ACTEL and Atmel FPGAs by generating the Shared Reduced Ordered Kronecker Decision Diagrams (SROKDDs). It adapted a breadth-first top-down algorithm for the SROKDD generation. If the input function is a multiple output function, it decomposes a single one-output function at a time. Based on the variable and expansion selection of the first single-output function, it decomposes the other single-output functions and generates the Kronecker Decision Diagram (KDD). During the decomposition, it combines all those isomorphic trees in order to generate a SROKDD. Its heuristics to select the variable are based on the following three conditions. All these conditions can determine that the next level node is redundant.

Condition 1:

$$f_i = 0$$

$$f_i = 1$$

$$f_i = x_j$$

$$f_i = \overline{x_j}$$

It states that the data input function f_i is either a constant value, a single variable, or a negation of a variable.

Condition 2:

$$f_i = f_j$$

It states that data input function f_i is identical to input function f_j in the same level of the tree.

Condition 3:

$$f_i = \overline{f_j}$$

It states that data input function f_i is the complement of data input function f_j in the same level of the tree.

Its heuristics to select the expansion are divided into three following modes.

-C1 The expansion of a node is selected based on the two functions out of f_{s_i}, $f_{\overline{s_i}}$, and $f_{s_i} \oplus f_{\overline{s_i}}$ having the highest fan-out. In case of a tie, the heuristic C3 is applied.

-C2 If the variable occurs mostly in a positive form in the output function, Davio expansion 2 is selected. If the variable occurs mostly in a negative form, Davio expansion 3 is selected. If there is a tie, the Shannon expansion is chosen.

-C3 The expansion of a node is selected based on the two functions out of f_{s_i}, $f_{\overline{s_i}}$, and $f_{s_i} \oplus f_{\overline{s_i}}$ having the least number of product terms.

The main objective of the above FPGA technology mapping approaches was to minimize the area.

IV.2. OUR APPROACH

We developed the concept of the Reduced Shared Permuted Kronecker Decision Diagram (RSPKDD), and we applied most of the heuristics from the above researches to mapping as one of the possible applications of the RSPKDD. Our FPGA mapping techniques try to construct the network in such a way that:

- the decomposed network is technology-feasible for the Atmel devices.

- the number of nodes in the network is as small as possible.
- the path from the input to output is as short as possible.
- the selected variable and expansion can vary in every level of the tree.

The presented method has the following assets:

- The decomposition methods are specifically adapted to the FPGAs whose general architectures are based on logic cells which can take up to three input variables.
- It applies a set of rules to select a good variable and an appropriate expansion for each node.
- It uses the shared reduced order approach to reduce the number of nodes and levels.

CHAPTER V

THE DAVIO EXPANSION

The general objective of decomposition methods in logic synthesis is to decompose a given set of functions into smaller subfunctions that can be realized by certain gate structures so that the final circuit realization is optimized for speed and area. Usually, a large logic function is difficult to analyze and to find a small circuit realization for it. One way to solve the problem is to decompose the initial logic function into smaller blocks which are easier to implement. There are two basic approaches to decomposition: the algebraic factorization and the Boolean decomposition. The algebraic decomposition methods are based on the factoring and extraction of common functions. They operate not on Boolean functions, but on certain expressions that describe these functions. Boolean decomposition methods take advantage of the structure of the function itself to be decomposed. Because they operate on the whole functions they are computationally more expensive than the algebraic methods. Therefore, the multilevel synthesis tools like RENO and MISII make use of algebraic methods to find a local minimum and then try to apply Boolean decomposition methods to find a better local minimum.

One of the most fundamental concepts for the decomposition of logic functions is the Shannon expansion. The Shannon expansion can always be applied to a logic function in contrast to other types of Boolean decomposition like the Ashenhurst or Curtis decompositions. These decompositions can be only applied to logic functions belonging to certain classes, like the class of disjoint decomposable functions.

Therefore, this chapter reviews the concepts of the Davio expansions over the Galois Field (2) and shows its circuit realizations. It will be shown that the Davio expansion is ideally suited to the decomposition of logic functions to subfunctions that can be realized with the AT 6000 series.

V.1. DAVIO EXPANSIONS AND DECOMPOSITION

The well-known Davio expansion is given by

$$f\,(x_1,\ldots x_i,\ldots x_n) = x_i \cdot f\,(x_1,\ldots x_i{=}1,\ldots, x_n) \oplus \overline{x_i} \cdot f\,(x_1,\ldots x_i{=}0,\ldots, x_n) \qquad (1)$$

By applying the rules $\overline{a} = 1 \oplus a$ and $a = 1 \oplus \overline{a}$ one obtains the two Davio expansions:

$$\begin{gathered} f\,(x_1,\ldots x_i,\ldots x_n) = \\ f\,(x_1,\ldots x_i{=}0,\ldots, x_n) \oplus [x_i \cdot [f\,(x_1,\ldots x_i{=}0,\ldots, x_n) \oplus f\,(x_1,\ldots x_i{=}1,\ldots, x_n)]] \end{gathered} \qquad (2)$$

$$\begin{gathered} f\,(x_1,\ldots x_i,\ldots x_n) = \\ f\,(x_1,\ldots x_i{=}1,\ldots, x_n) \oplus [\overline{x_i} \cdot [f\,(x_1,\ldots x_i{=}0,\ldots, x_n) \oplus f\,(x_1,\ldots x_i{=}1,\ldots, x_n)]] \end{gathered} \qquad (3)$$

in short form:

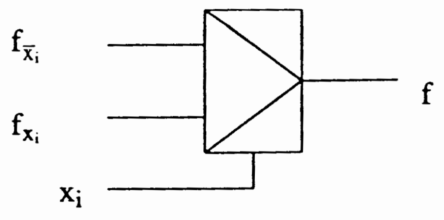
$$f = x_i \cdot f_{x_i} \oplus \overline{x_i} \cdot f_{\overline{x_i}} \qquad (4)$$

$$f = f_{\overline{x_i}} \oplus x_i \cdot [f_{x_i} \oplus f_{\overline{x_i}}] = f_{\overline{x_i}} \oplus x_i \cdot g \qquad (5)$$

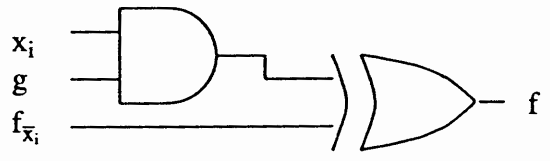
$$f = f_{x_i} \oplus \overline{x_i} \cdot [f_{x_i} \oplus f_{\overline{x_i}}] = f_{x_i} \oplus \overline{x_i} \cdot g \qquad (6)$$

The circuit realization of Equation (4) is given by a multiplexer gate while Equation (5) and Equation (6) describe an AND/EXOR gate structure.

a. Circuit Realization of Equation (4):



b. Circuit Realization of Equation (5):



c. Circuit Realization of Equation (6):

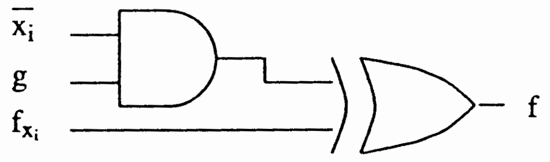


Figure 6. Circuit realizations of the Davio expansions.

It can be observed from Figure 6, the circuit realization of the three expansions correspond to the realizable functions of a macrocell of the Atmel AT 6000 series. Therefore, the Davio expansions are ideally suited for the decomposition of Boolean functions with respect to the realization with the AT 6000 series. It is shown in Figure 7 and Figure 8.

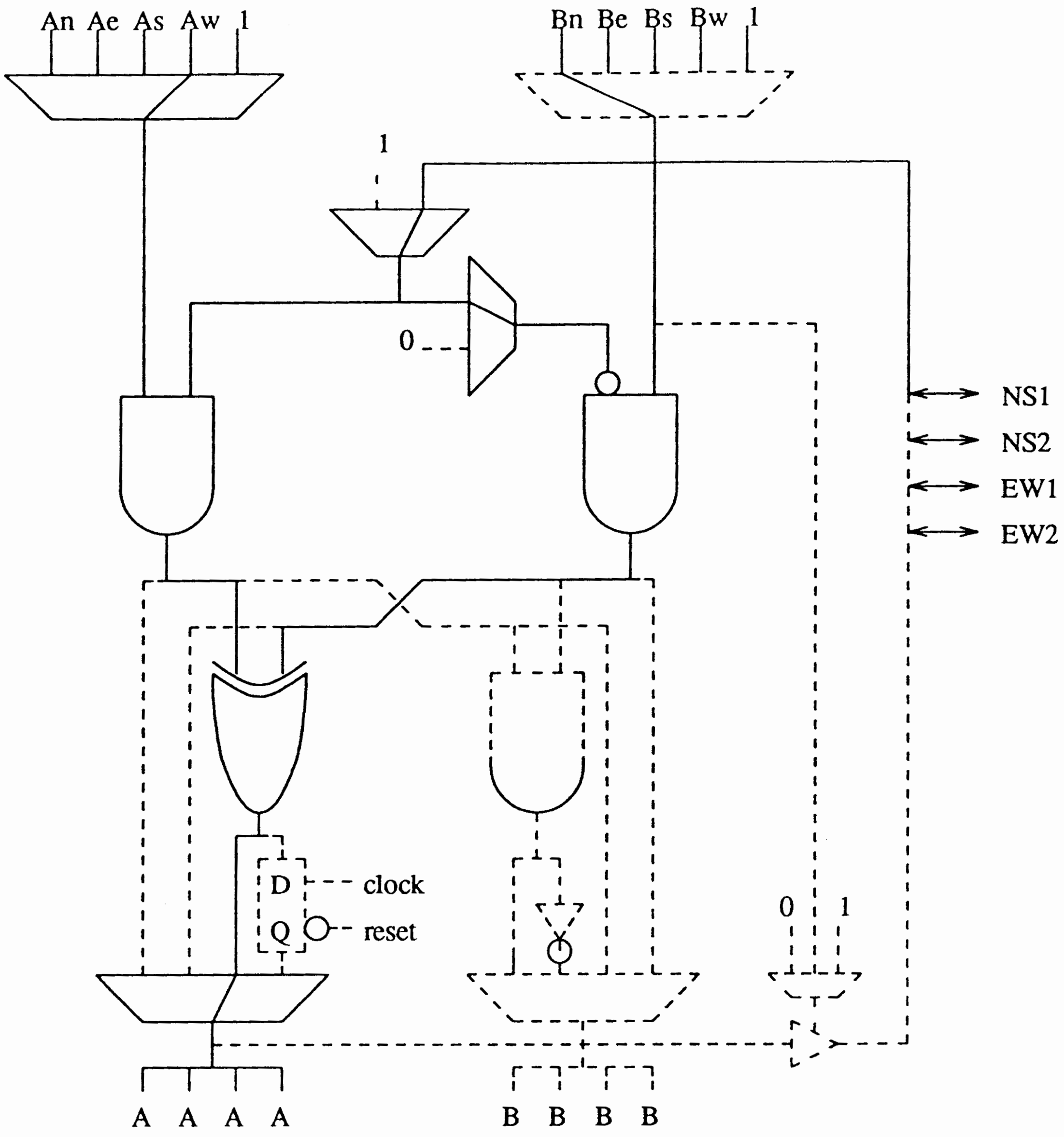


Figure 7. AT 6000 cell configuration- (A*L)XOR(B*Lnot).

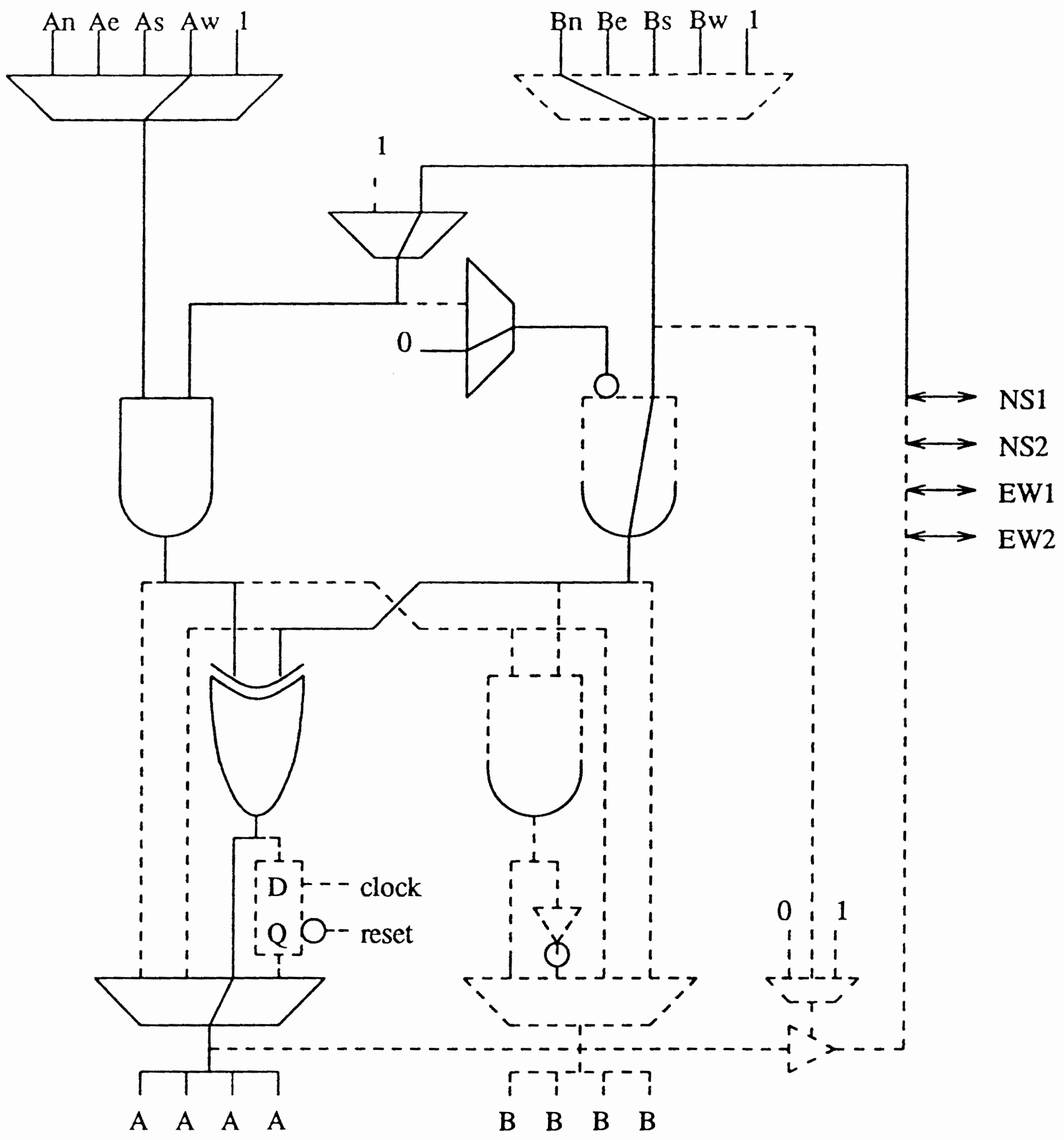


Figure 8. AT 6000 cell configuration- B XOR (A*L).

The application of (4) to all variables of a function leads to the construction of a BDD. BDDs are a graph representation of Boolean functions proposed by Akers[22] and developed by Bryant[10]. Multiple BDDs can be joined into a single graph which consists of the BDDs sharing their sub-graphs. Minato[4] called them Shared BDDs.

The application of (5) and (6) to each variable generates adaptive logic trees [15,16,17]. The FDDs are obtained by applying the reduction procedures used for BDDs[10] to the adaptive logic trees. If only equation (5) is used repeatedly for some fixed order of expansion variables, the Reed-Muller Trees are created. If for every variable one uses (5) and (6), the Generalized RM Trees are created. If the trees are based on equation (2) but with different orders of variables in subtrees, the Permuted Reed Muller tree is obtained. If all three expansions are applied with a fixed order of variables, the Kronecker Reed-Muller trees are obtained. Applying the expansion in a tree for a fixed order of expansion variables, but selecting various variable polarities in different subtrees, the Pseudo Kronecker Reed-Muller tree is obtained [19,20]. The Ordered Kronecker Decision Diagram (OKDD) is the decision tree obtained by applying any of the three expansions with fixed order of variables [18]. The Reduced OKDD (ROKDD) is obtained from the OKDD by removing isomorphic subtrees. The Shared ROKDD (SROKDD) for multioutput function is obtained similar to [14,23], connecting the isomorphic subtrees with positive or negative (output inverter) edges. The Permuted Kronecker Decision Diagram (PKDD) is the decision tree obtained by applying any of the three expansions with a different order of variables in subtrees. The Reduced Shared PKDD is the decision tree obtained by applying any of the three expansions with different order of variables in subtrees, and all isomorphic subtrees will be connected with either positive or negative edges. The REduced Shared PERmuted (RESPER) is the program which implements the Reduced Shared PKDD to decompose boolean functions and map to Atmel FPGAs. This program will be introduced in a later chapter.

V.2. CIRCUIT REALIZATIONS OBTAINED BY THE DAVIO EXPANSION

The relations between Reed-Muller Tree, Generalized Reed-Muller Tree, Permuted Reed-Muller Tree, and Kronecker Reed-Muller Tree introduced in recent papers are illustrated here. Different type of trees generated from Davio Expansion form the solution space investigated in this thesis, and the investigation of this space has been one of the major motives of this thesis. In this section some basic terms and theories will be defined, and based on those theories a space is created.

Definition 1. *Literal*

The *literal* of a variable x_i can be in either positive (x_i) or negative ($\overline{x}_i$) form.

Definition 2. *Polarity*

The *polarity* of a variable is "1" for a positive literal and "0" for a negative literal.

Definition 3. *Decomposition*

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

Definition 4. *Terminal Vertex*

A *teminal vertex* has an attribute a value value(v) $\in$ 0, 1.

Definition 5. *Non-terminal Vertex*

A *non-terminal vertex* has an attribute an argument index index(v) $\in$ 1,..., n and two children low(v), high(v) $\in$ V.

Definition 6. *Directed Graph*

A *directed graph* is a finite nonempty set *V* together with an irreflexive relation *R* on *V*. As with graphs, the elements of *V* are called vertices. Each ordered pair in *R* is referred to as a directed edge.

Definition 7. *Cycle*

A *u*-*v* trail in which *u*=*v* and which contains at least three edges is called a circuit. A cicuit is a graph *G* in which no vertices are repeated (except the first and last) is called a cycle of *G*.

Definition 8. *Acyclic Graph*

A graph has no cycle is *acyclic graph.*

Definition 9. *Directed Acyclic Graph*

A *directed acyclic graph* (DAG) is a graph which is directed and acyclic. An acyclic digraph has at least one point of outdegree or indegree.

Definition 10. *Tree*

A *tree* is a connected graph with no cycles. Let *u* and *v* be any vertices of a tree *T*. Then there is a unique path in T from *u* to *v*.

Definition 11. *Binary Decision Diagram*

Binary Decision Diagram (BDD) (Fig. 9) is a Directed Acyclic Graph having root vertex v denoting a function f_v denoted recursively as

1. If v is a terminal vertex:

    a) If value(v) = 1, then $f_v = 1$.

    b) If value(v) = 0, then $f_v = 0$.

2. If v is a nonterminal vertex with index(v)=i, then f_v is the function

$$f_{v(x_{1},}\ldots, x_n) = \bar{x}\cdot f_{low(v)}(x_1, \ldots, x_n) + x\cdot f_{high(v)}(x_1, \ldots, x_n)$$

where the cofactors are defined as $f_{low(v)}(x_1, \ldots, x_n) = (x_1, \ldots, x_{i-1}, 0, \ldots, x_n)$ and $f_{high(v)}(x_1, \ldots, x_n) = (x_1, \ldots, x_{i-1}, 1, \ldots, x_n)$.

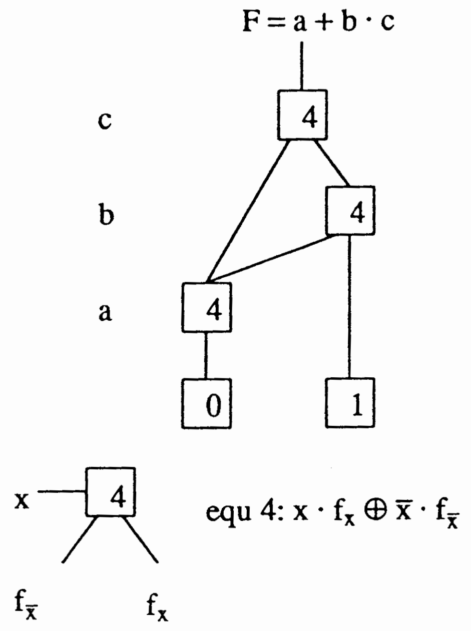


Figure 9. Binary Decision Diagram.

Definition 12. *Ordered Function Graph*

An *ordered function graph* is a function graph such that for any non terminal vertex v, if *low(v)* is also non terminal, then *index(v) < index(low(v))*. Similary, if *high(v)* is nonterminal, then *index(v) <index(high(v))*.

Definition 13. *Ordered Binary Decision Diagram*

An ordered BDD is an *Ordered Binary Decision Diagram (OBDD)*.

Definition 14. *Reduced Ordered Function Graph*

An *ordered function graph* is *reduced* if it contains no vertex v with *low(v)* = *high(v)*, nor does it contain distinct vertices v and v′ such that the subgraphs rooted by v and v′ are isomorphic.

Definition 15. *Reduced Ordered Binary Decision Diagram*

An reduced OBDD is an *Reduced Binary Decision Diagram (ROBDD).*

Definition 16. *Shared Reduced Ordered BDD*

Shared Reduced Ordered BDD (SBDD) is a DAG which contains multiple BDDs sharing their subgraphs. (Fig 10)

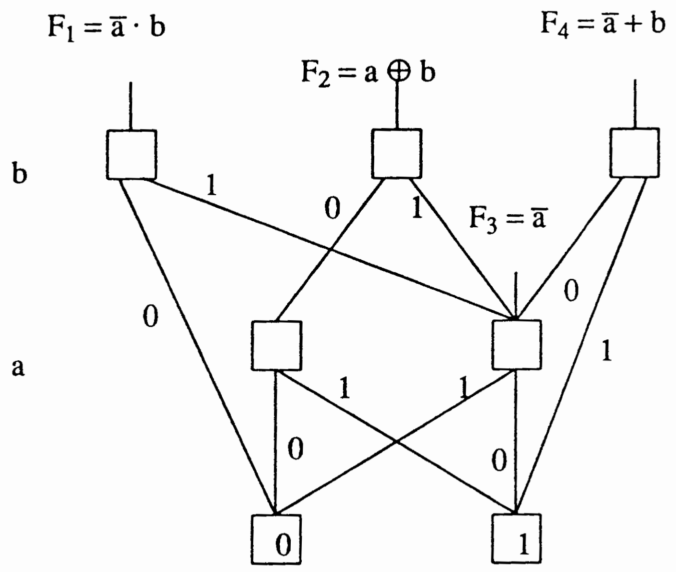


Figure 10. Shared BDD.

Definition 17. *Adaptive Logic Tree*

An *Adaptive Logic Tree*[16] is a function graph having root vertex v denoting a function f_v denoted recursively as

1. If v is a terminal vertex:

   a) If value(v) = 1, then $f_v = 1$.

   b) If value(v) = 0, then $f_v = 0$.

2. If v is a nonterminal vertex with index(v)=i, then f_v is the function

$$f_{v(x_1, \ldots, x_n)} = \overline{x} \cdot f_{low(v)}(x_1, \ldots, x_n) + x \cdot f_{high(v)}(x_1, \ldots, x_n)$$

Example 1: Let F_n represent a general Boolean function of n independent variables x_i, $1 < i < n$. We can then express each F_n as a canonical sum of products. Thus the general structure of F_n may be developed as follows:

$$F_0 = k_0 \quad \cdots \tag{7}$$

$$F_1 = x'_1 \cdot k_0 + x_1 \cdot k_1 \quad \cdots \tag{8}$$

$$F_2 = x'_2 \cdot x'_1 \cdot k_0 + x'_2 \cdot x_1 \cdot k_1 + x_2 \cdot x'_1 \cdot k_2 + x_2 \cdot x_1 \cdot k_3 \quad \cdots \tag{9}$$

where x'_i means "not x_i", + is the OR function, · is the AND function and k_i = 0 or 1.

Thus the k_i is the coefficients in the minterm expansion of the function. With a suitable choice of k_i, F_n can take on any of the available functions of n variables.

Further, we see that, from the general form,

$$F_n = x'_n \cdot F_{n-1} + x_n \cdot F^*_{n-1} \quad \cdots \tag{10}$$

where F^*_j is F_j with different set of k_i. This leads to an implementation of equation 10 with the form of that of Figure 11. We can build up a general adaptive logic tree for the n variables in terms of the circuit of Figure 11 and a mechanisation of the recursion given by equation 10.

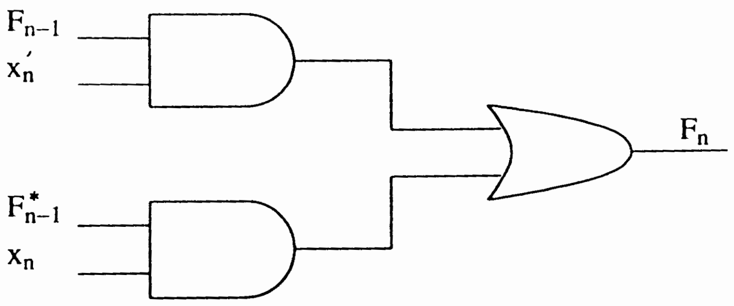


Figure 11. Circuit of the general form.

Definition 18. *Functional Decision Diagram*

A *Functional Decision Diagram (FDD)* is a DAG having root vertex v denoting a function f_v denoted recursively as

1. If v is a terminal vertex:

   a) If value(v) = 1, then $f_v = 1$.

   b) If value(v) = 0, then $f_v = 0$.

2. If v is a nonterminal vertex with index(v)=i, then f_v is the function of $f_v(x_i)$.

Based on the Shannon Expansion of the two-level fixed-polarity Reed-Muller Expansion, there exists a recursively defined multilevel representation[16]:

$$\begin{aligned} F_n &= F_n(x_1, x_2, \ldots, x_n) \\ &= F_n(x_n = 0) \cdot \overline{x_n} \oplus F_n(x_n = 1) \cdot x_n \\ &= F_n(x_n = 0) \oplus [F_n(x_n = 0) \oplus F_n(x_n = 1)] \cdot x_n \\ &= F_{n-1} \oplus F^*_{n-1} \cdot x_n \end{aligned}$$

where F^*_{n-1} is a function of $n-1$ variables with identical structure but different coefficients than F_{n-1}. As a prerequisite, the input cubes must be disjoint. Graphically this could be illustrated as in Figure 12:

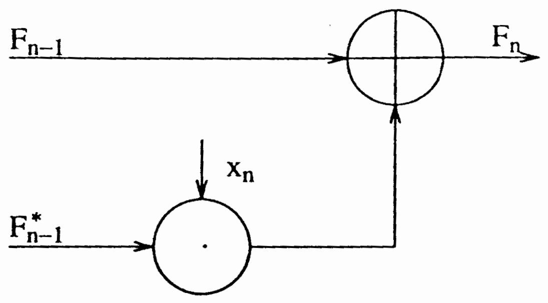


Figure 12. Function graph of F_n.

Using this recursive, multi-level representation, the function of four variables $F = \bar{a} \cdot \bar{b} \cdot d + b \cdot c \cdot \bar{d} + \bar{b} \cdot \bar{c} \cdot d$ becomes an adaptive logic tree[16,17] shown in Figure 11.

This logic tree could be easily transformed into a binary tree of Figure 14, by combining the AND and EXOR nodes from Figure 12 into a single node. Note that this is a multi-level representation of the canonical fixed-polarity RME with all of its coefficients.

Using the same reduction procedures used for BDDs [10]:

- combine all isomorphic subtrees.
- eliminate all nodes with isomorphic children.

and obtains a reduced representation of this logic tree.

The result is a canonical representation of the functional domain (see Fig 15). It is called Functional Decision Diagram since each node of the FDD decides whether the product term belongs to the function or not.

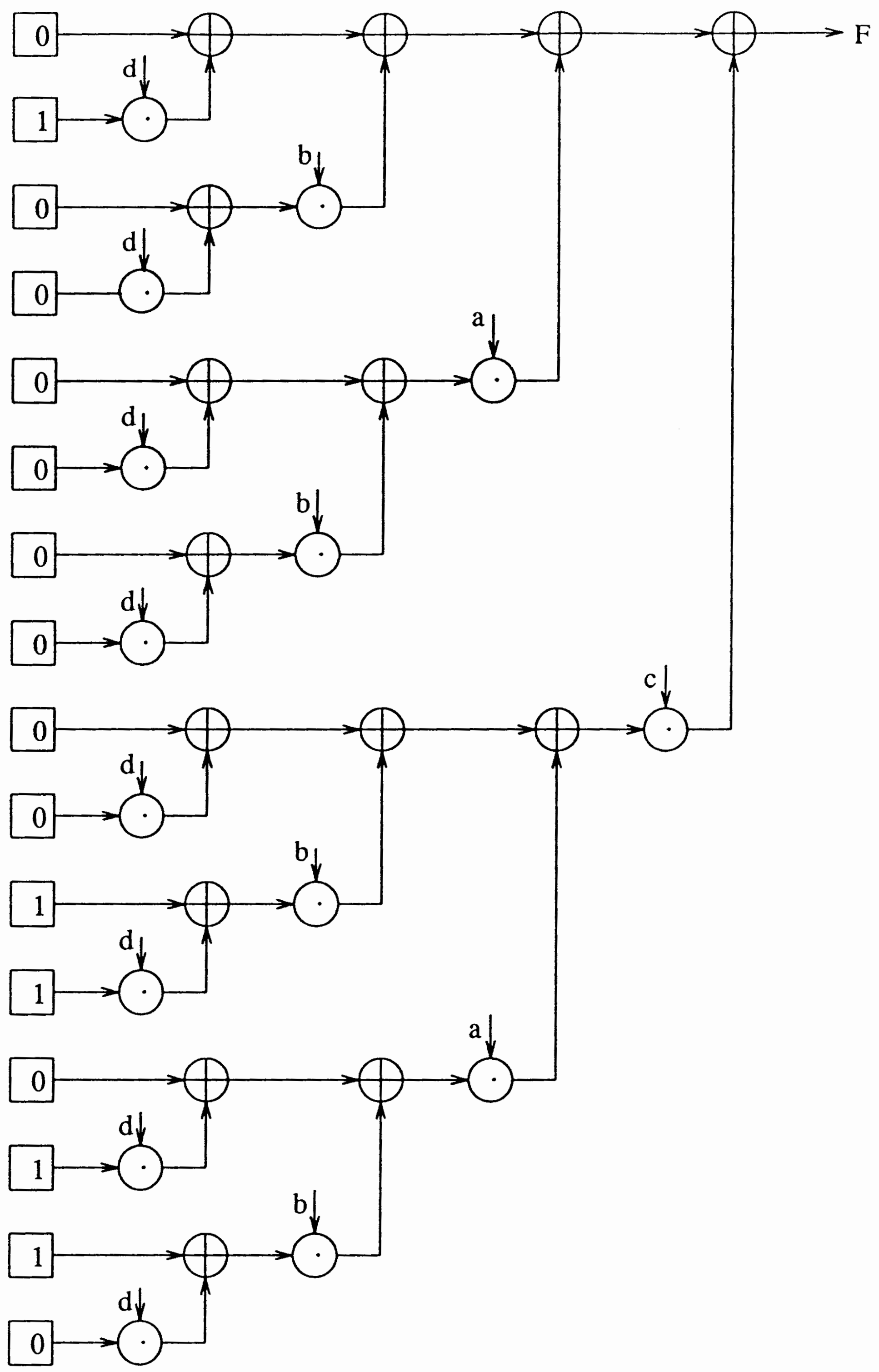


Figure 13. Adaptive Logic-Tree.

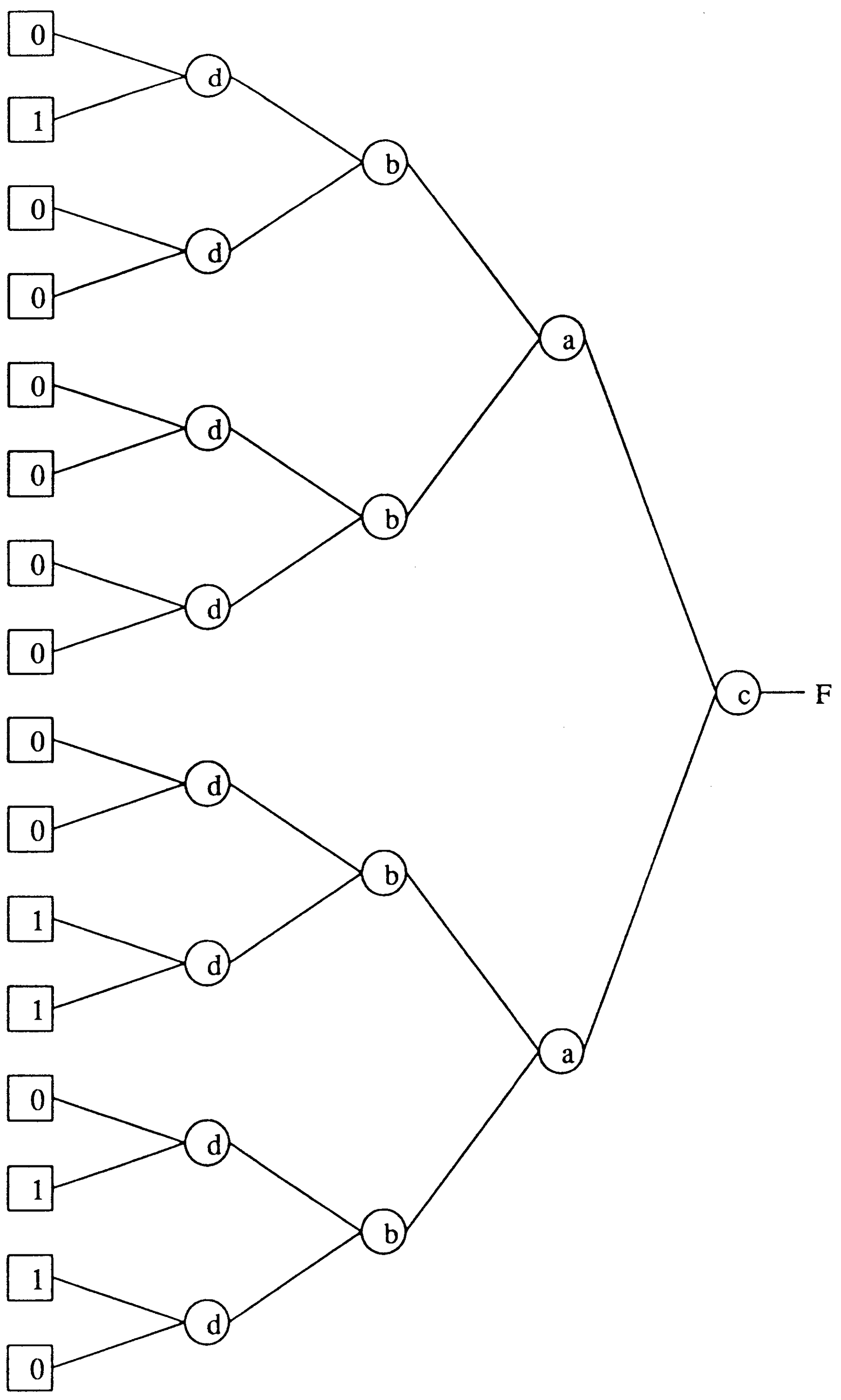


Figure 14. Logic-Tree as Binary Tree.

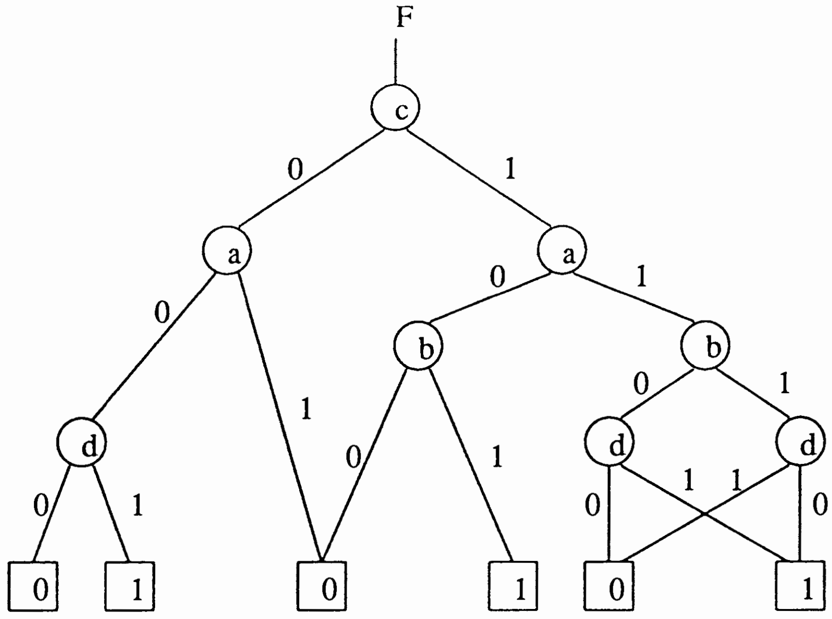


Figure 15. Functional Decision Diagram.

Applying the theory of Functional Decision Diagrams, we can use different combinations of Davio expansions to generate different types of trees.

Definition 19. *Reed-Muller Tree*

A Reed-Muller Tree (Fig. 16) is a function graph having root vertex v denoting a function f_v denoted recursively as:

1. If v is a terminal vertex:

   a. If value(v) = 1, then $f_v = 1$.

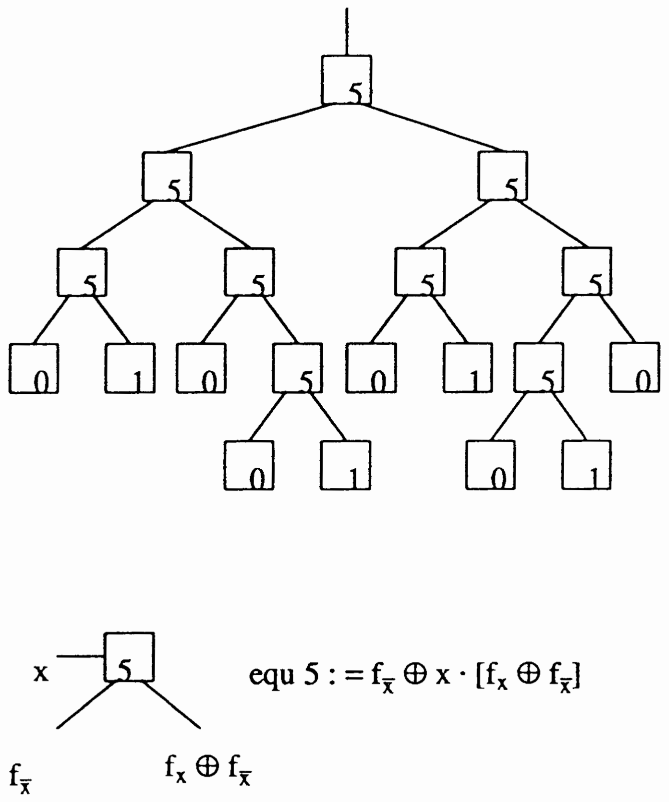
   b. If value(v) = 0, then $f_v = 0$.

2. If v is a non-terminal vertex with index(v) = i, the f_v is one and only of the functions:

   a. $f_v(x_1, \ldots, x_n) = f_{low(v)}(x_1,\ldots,x_n) \oplus x \cdot [f_{high(v)}(x_1,\ldots,x_n) \oplus f_{low(v)}(x_1,\ldots,x_n)]$.

Any path from the root to the terminal vertices will traverse the same order of variables.

$$F = \bar{a}bcd + ab\bar{c}d + ac$$



$$\text{equ } 5 := f_{\bar{x}} \oplus x \cdot [f_x \oplus f_{\bar{x}}]$$

Figure 16. Reed-Muller Tree.

Definition 20. *Permuted Reed-Muller Tree*

A Permuted Reed-Muller Tree (Fig. 17) is a function graph having root vertex v denoting a function f_v denoted recursively as:

1. If v is a terminal vertex:

   a. If value(v) = 1, then $f_v = 1$.

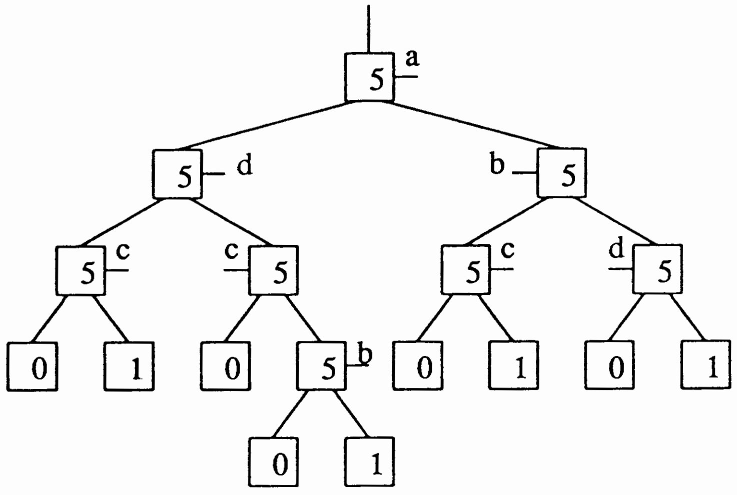
   b. If value(v) = 0, then $f_v = 0$.

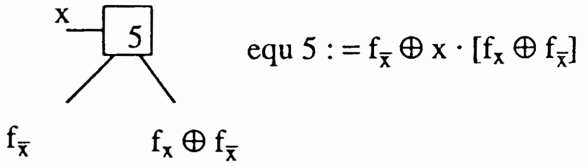
2. If v is a non-terminal vertex with index(v) = i, the f_v is one and only of the functions:

   a. $f_v(x_1, \ldots, x_n) = f_{low(v)}(x_1,\ldots,x_n) \oplus x \cdot [f_{high(v)}(x_1,\ldots,x_n) \oplus f_{low(v)}(x_1,\ldots,x_n)]$.

Any path from the root to the terminal vertices will traverse a different order of variables.

$F = \bar{a}bcd + ab\bar{c}d + ac$





x
5
$equ\ 5 := f_{\bar{x}} \oplus x \cdot [f_x \oplus f_{\bar{x}}]$
$f_{\bar{x}}$
$f_x \oplus f_{\bar{x}}$

Figure 17. Permuted Reed-Muller Tree.

Definition 21. *Generalized Reed-Muller Tree*

A Generalized Reed-Muller Tree (Fig. 18) is a function graph having root vertex v denoting a function f_v denoted recursively as:

1. If v is a terminal vertex:

   a. If value(v) = 1, then $f_v = 1$.

   b. If value(v) = 0, then $f_v = 0$.

2. If v is a non-terminal vertex with index(v) = i, the f_v is a one and only of the functions:

   a. $f_v(x_1, \ldots, x_n) = f_{low(v)}(x_1,\ldots,x_n) \oplus x \cdot [f_{high(v)}(x_1,\ldots,x_n) \oplus f_{low(v)}(x_1,\ldots,x_n)]$.

   b. $f_v(x_1, \ldots, x_n) = f_{high(v)}(x_1,\ldots,x_n) \oplus \overline{x} \cdot [f_{high(v)}(x_1,\ldots,x_n) \oplus f_{low(v)}(x_1,\ldots,x_n)]$.

Any path from the root to the terminal vertices will traverse the same order of variables.

Definition 22. *Kronecker Reed-Muller Tree*

A Kronecker Reed-Muller Tree (Fig. 19) is a function graph having root vertex v denoting a function f_v denoted recursively as:

1. If v is a terminal vertex:

   a. If value(v) = 1, then $f_v = 1$.

   b. If value(v) = 0, then $f_v = 0$.

2. If v is a non-terminal vertex with index(v) = i, the f_v is a one and only of the functions:

   a. $f_v(x_1, \ldots, x_n) = f_{low(v)}(x_1,\ldots,x_n) \oplus x \cdot [f_{high(v)}(x_1,\ldots,x_n) \oplus f_{low(v)}(x_1,\ldots,x_n)]$.

   b. $f_v(x_1, \ldots, x_n) = f_{high(v)}(x_1,\ldots,x_n) \oplus \overline{x} \cdot [f_{high(v)}(x_1,\ldots,x_n) \oplus f_{low(v)}(x_1,\ldots,x_n)]$.

   c. $f_v(x_1, \ldots, x_n) = \overline{x} \cdot [f_{low(v)}(x_1,\ldots,x_n) \oplus x \cdot f_{high(v)}(x_1,\ldots,x_n)]$.

Any path from the root to the terminal vertices will traverse the same order of variables.

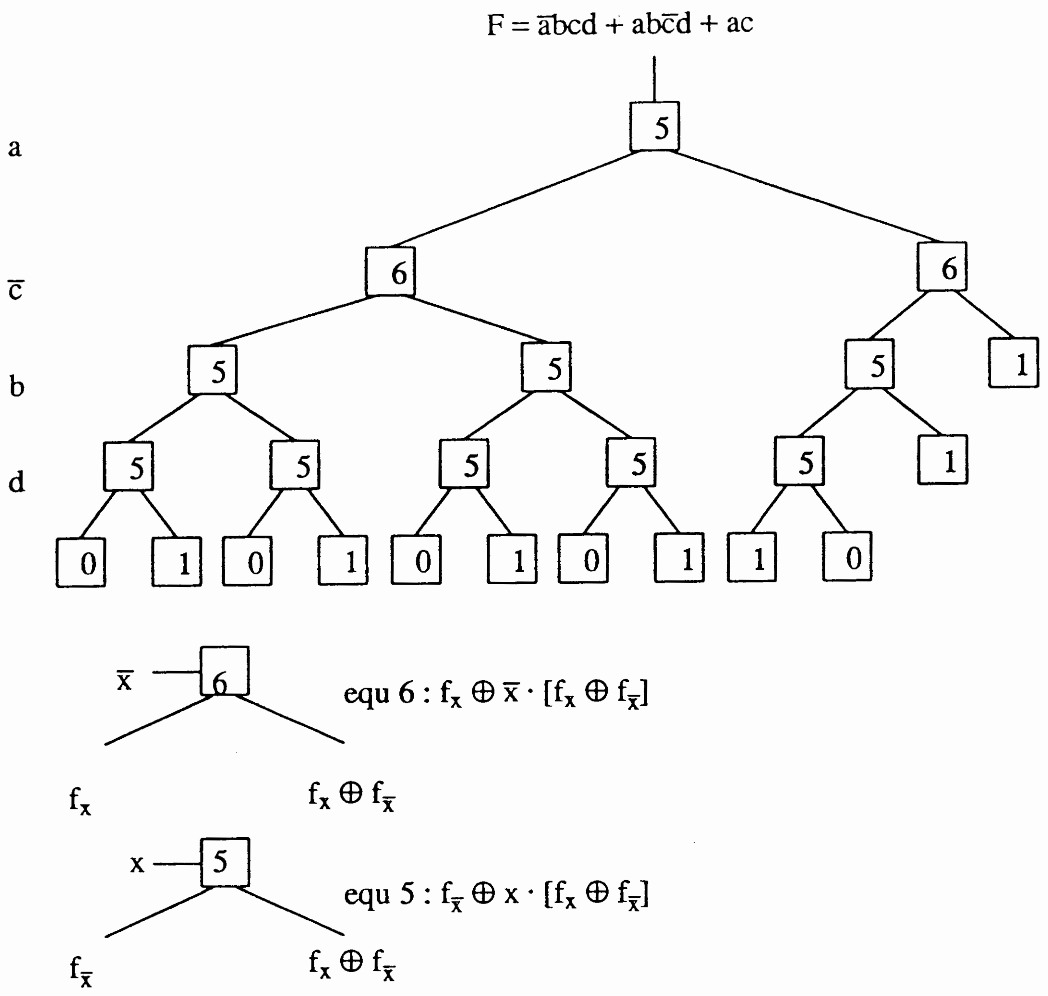

$F = \bar{a}bcd + ab\bar{c}d + ac$
a
$\bar{c}$
b
d
5
6
6
5
5
5
1
5
5
5
5
5
1
0
1
0
1
0
1
0
1
1
0
$\bar{x}$
6
equ 6 : $f_x \oplus \bar{x} \cdot [f_x \oplus f_{\bar{x}}]$
f_x
$f_x \oplus f_{\bar{x}}$
x
5
equ 5 : $f_{\bar{x}} \oplus x \cdot [f_x \oplus f_{\bar{x}}]$
$f_{\bar{x}}$
$f_x \oplus f_{\bar{x}}$

Figure 18. Generalized Reed-Muller Tree.

Definition 23. *Pseudo-Kronecker Reed-Muller Tree*

A Pseudo-Kronecker Reed-Muller Tree (Fig. 20) is a function graph having root vertex v denoting a function f_v denoted recursively as:

1. If v is a terminal vertex:

   a. If value(v) = 1, then $f_v = 1$.

   b. If value(v) = 0, then $f_v = 0$.

2. If v is a non-terminal vertex with index(v) = i, the f_v is a one and only of the functions:

   a. $f_v(x_1, \ldots, x_n) = f_{low(v)}(x_1,\ldots,x_n) \oplus x \cdot [f_{high(v)}(x_1,\ldots,x_n) \oplus f_{low(v)}(x_1,\ldots,x_n)]$.

   b. $f_v(x_1, \ldots, x_n) = f_{high(v)}(x_1,\ldots,x_n) \oplus \bar{x} \cdot [f_{high(v)}(x_1,\ldots,x_n) \oplus f_{low(v)}(x_1,\ldots,x_n)]$.

   c. $f_v(x_1, \ldots, x_n) = \bar{x} \cdot [f_{low(v)}(x_1,\ldots,x_n) \oplus x \cdot f_{high(v)}(x_1,\ldots,x_n)]$.

Any path from the root to the terminal vertices will traverse the same order of variables.

Definition 24. *Permuted Kronecker Reed Muller Tree*

A Permuted-Kronecker Reed-Muller Tree (Fig. 21) is a function graph having root vertex v denoting a function f_v denoted recursively as:

1. If v is a terminal vertex:

   a. If value(v) = 1, then $f_v = 1$.

   b. If value(v) = 0, then $f_v = 0$.

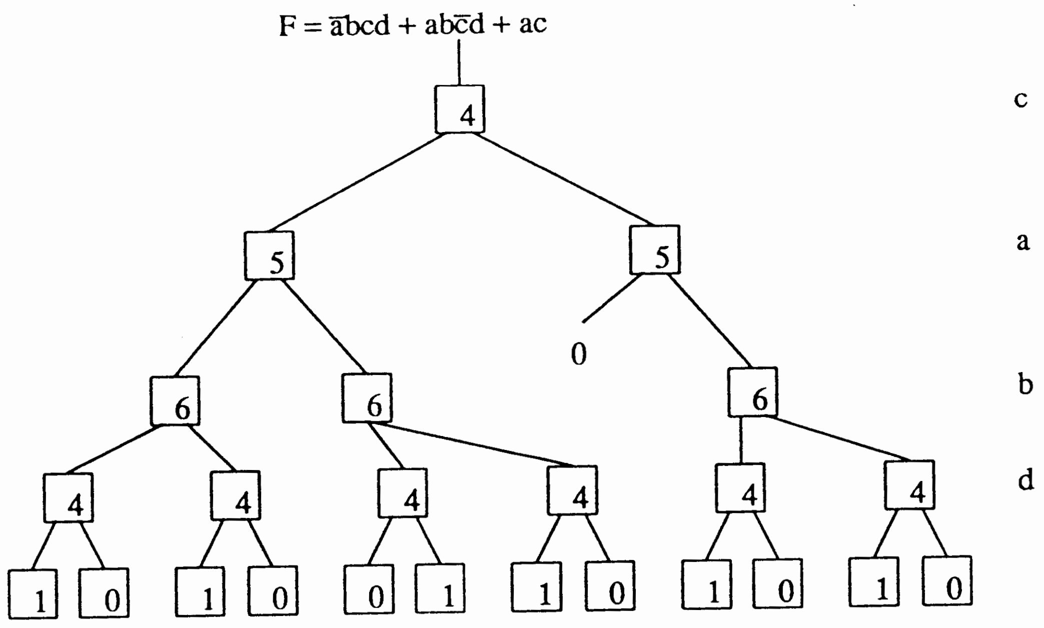
2. If v is a non-terminal vertex with index(v) = i, the f_v is a one and only of the functions:

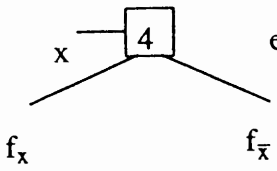
   a. $f_v(x_1, \ldots, x_n) = f_{low(v)}(x_1,\ldots,x_n) \oplus x \cdot [f_{high(v)}(x_1,\ldots,x_n) \oplus f_{low(v)}(x_1,\ldots,x_n)]$.

   b. $f_v(x_1, \ldots, x_n) = f_{high(v)}(x_1,\ldots,x_n) \oplus \bar{x} \cdot [f_{high(v)}(x_1,\ldots,x_n) \oplus f_{low(v)}(x_1,\ldots,x_n)]$.

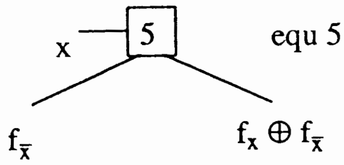
   c. $f_v(x_1, \ldots, x_n) = \bar{x} \cdot [f_{low(v)}(x_1,\ldots,x_n) \oplus x \cdot f_{high(v)}(x_1,\ldots,x_n)]$.

Any path from the root to the terminal vertices will traverse different order of variables.

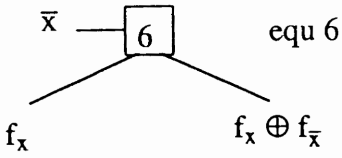




equ 4 : $x \cdot f_x \oplus \bar{x} \cdot f_{\bar{x}}$



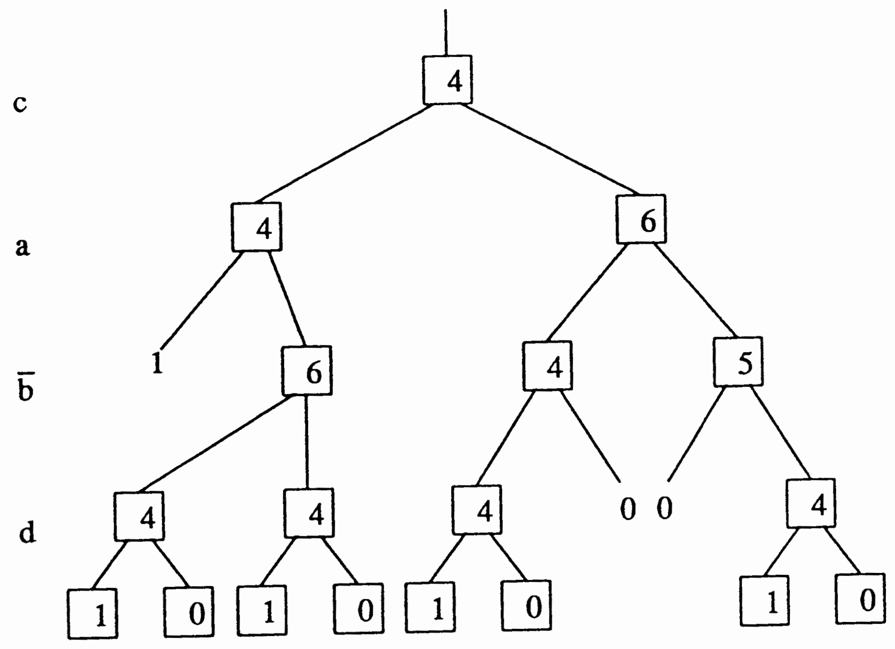
equ 5 : $f_{\bar{x}} \oplus x \cdot [f_x \oplus f_{\bar{x}}]$

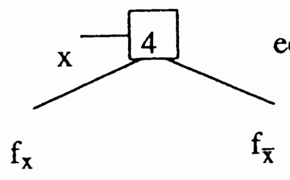


equ 6 : $f_x \oplus \bar{x} \cdot [f_x \oplus f_{\bar{x}}]$

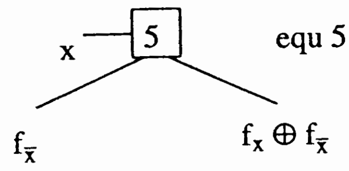
Figure 19. Kronecker Reed-Muller Tree.

$F = \bar{a}bcd + ab\bar{c}d + ac$

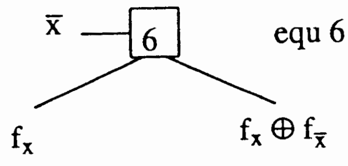




equ 4 : $x \cdot f_x \oplus \bar{x} \cdot f_{\bar{x}}$

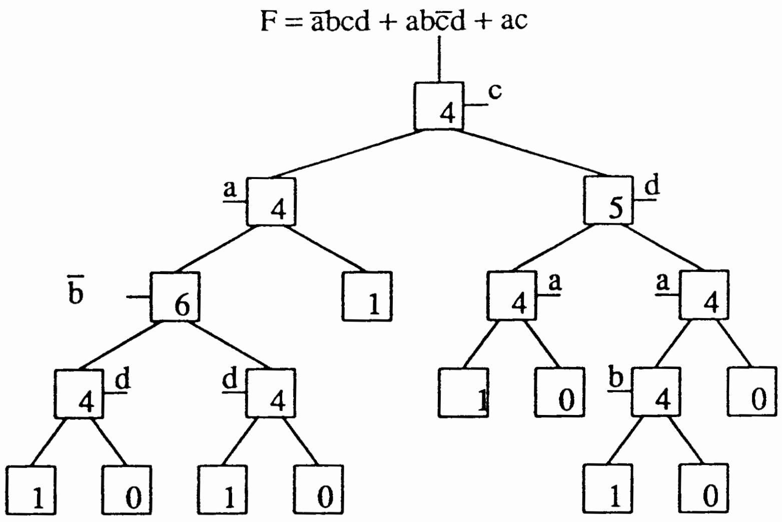


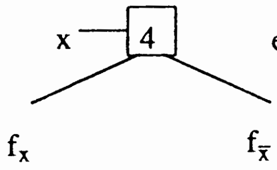
equ 5 : $f_{\bar{x}} \oplus x \cdot [f_x \oplus f_{\bar{x}}]$



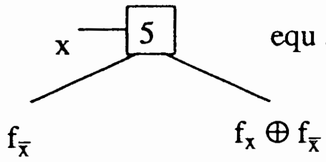
equ 6 : $f_x \oplus \bar{x} \cdot [f_x \oplus f_{\bar{x}}]$

Figure 20. Pseudo-Kronecker Reed-Muller Tree.

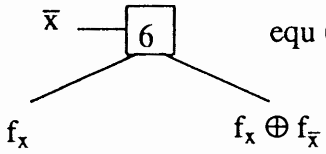




equ 4 : $x \cdot f_x \oplus \bar{x} \cdot f_{\bar{x}}$



equ 5 : $f_{\bar{x}} \oplus x \cdot [f_x \oplus f_{\bar{x}}]$



equ 6 : $f_x \oplus \bar{x} \cdot [f_x \oplus f_{\bar{x}}]$

Figure 21. Permuted Kronecker Reed Muller Tree.

V.3. TWO LEVEL CIRCUIT REALIZATIONS

The expansion formulas applied to various variables and the resulting subfunctions f_j generate different multi-level tree circuits as described in the previous section.

The obtained tree circuits can be flattened to a two level form which can be realized by an AND-EXOR circuit.

Definition 25. *Flattening*

This is one of the basic operations in logic synthesis, which is the inverse operation of "substitution". If G is a fan-in function of F, flattening G into F re-expresses F without G.

Example 2. If $F = Ga + b$ and $G = c + d$ then flattening G into F results in $F = ac + ad + b$.

Definition 26. *Reed-Muller form*

Reed-Muller (RM) form is an ESOP obtained by flattening of an RM Tree.

Example 3. Assuming a four variables input function (Fig. 16), the RM form expression is of the following form:

$$F(a, b, c, d) = c \oplus bcd \oplus ac \oplus abd$$

Definition 27. *Generalized Reed-Muller (GRM) form*

Generalized Reed-Muller (GRM) form is an ESOP obtained by flattening of a GRM Tree.

Example 4. Assuming a four variables input function (Fig. 18), the GRM form expression is of the following form:

$$F(a, b, c, d) = d \oplus \bar{c}d \oplus a \oplus ad \oplus ab \oplus a\bar{c}$$

Definition 28. *Kronecker Reed-Muller (KRM) form*

Kronecker Reed-Muller (KRM) form is an ESOP obtained by flattening of a KRM Tree.

Example 5 Assuming a four variables input function (Fig. 19), the KRM form expression is of the following form:

$$F(a, b, c, d) = cd \oplus \bar{b}cd \oplus ac \oplus a\bar{b}cd \oplus a\bar{b}d \oplus a\bar{b}cd \oplus ad$$

Definition 29. *Pseudo-Kronecker Reed-Muller (PKRM) form*

Pseudo-Kronecker Reed-Muller (PKRM) form is an ESOP obtained by flattening of a PKRM Tree.

Example 6 Assuming a four variables input function (Fig. 20), the KRM form expression is of the following form:

$$F(a, b, c, d) = ac \oplus cd \oplus \bar{b}cd \oplus a\bar{b}d \oplus ad$$

V.4. SOLUTION SPACE

From [3] and [18], they show that permuted RM synthesis and KRM synthesis can generate compact circuits. We looked into several different types of trees and DAGs, and those diagram can create a space for different synthesis methods for multi-level circuits. Table I shows all those forms of diagram with different search methods, and there exists some empty space for researchers to further explore. Comparing KRMs with RMs, KRMs have an edge since they use three equations rather than one. Comparing the permuted tree search with the non-permuted tree search, permuted tree search will provide more flexibility to select a splitting variable. Because of these two advantages, the researcher chose KRM tree combined with permuted tree search and DAG as our approach to create a compact circuit.

TABLE I

SOLUTION SPACE

	RM	GRM	KRM
non-permuted Tree	REMIT		Techmap KDD
Permuted Tree	REMIT		RESPER
Non-Permuted DAG			Techmap KDD
Permuted DAG			RESPER

CHAPTER VI

SEARCHING METHODS FOR TREES

In many problems, a systematic order to search all the vertices in a graph is a must. In tree search, it always start from the root vertex. Although there are many possible orders for visiting vertices of the graph, two methods are of particular importance.

VI.1. DEPTH FIRST SEARCH

Depth first traversal of a graph is roughly analogous to preorder traversal of an ordered tree. It traverses a singlepath of the graph as far as it can go (that is, until it visits a node with no successors or a node all of whose successors have already been visited). It then resumes at the last node on the path just traversed that has an unvisited successor and begins traversing a new path emanating from that node. Suppose that the traversal has just visited a vertex v, and let $w_1,w_2,...,w_k$ be the vertices adjacent to v. Then we shall next vist w_1 and keep $w_2,...,w_k$ waiting. After visiting w_1 we traverse all the vertices to which it is adjacent before returning to traverse $w_2,...,w_k$.

Depth-first traversal is naturally formulated as a recursive algorithm. Its action, when it reaches a vertex v, is

```
DepthFirst(graph)
{
for all vertex in graph do
```

```
          visited[vertex] = FALSE;
     for all vertex in graph do
          if not visited[vertex] then
               Traverse(vertex);
     }
```

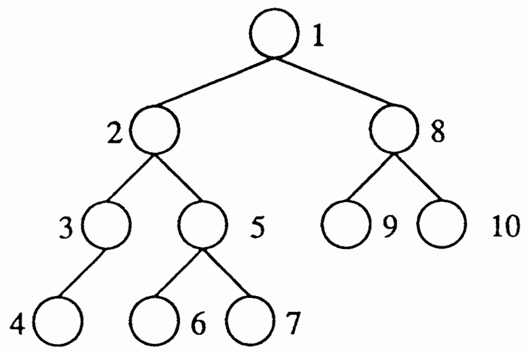


Figure 22. Depth first search.

The recusion is performed in the following procedure, to be declared within the previous one.

```
     Traverse(vertex)
     vertexW : vertices adjacent to the visit node
     {
     visited[vertex] = TRUE;
     Visit(vertex);
     for all vertexW adjacent to vertex do
          if not visited[vertexW] then
               Traverse(vertexW);
     }
```

VI.2. BREADTH FIRST SEARCH

Breadth first traversal of a graph is roughly analogous to level-by-level traversal of an ordered tree. It visits all successors of a visited node before visiting any successors of any of those successors. This is contradistinction to depth-first traversal, which visits the successors of a visited node before visiting any of its "brothers." If the traversal has just visited a vertex v, then it next visits all the vertices adjacent to v, putting other vertices adjacent to these in a queue to be traversed after all vertices adjacent to v have been visited.

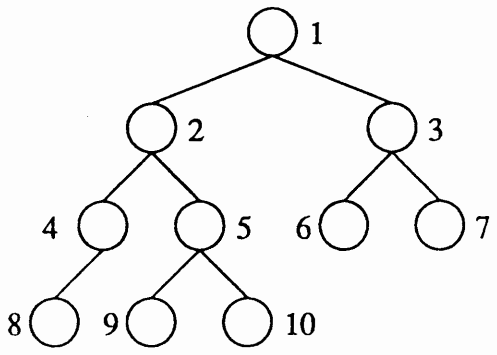


Figure 23. Breadth first search.

Since using recursion and programming with stacks are essentially equivalent, depth first traversal can be formulated with using stack, pushing all unvisited vertices adjacent to the one being visited onto the stack and popping the stack to find the next vertex to visit. The algorithm for breadth-first traversal is quite similar to the resulting algorithm for depth first traversal, except that a queue is needed instead of a stack. Its outline follows.

```
BreadthFirst(Graph)
vertexW : vertices adjacent to vertex
{
for all vertex in graph do
    if not visited(vertex) then
    begin
        AddQueue(vertex, Queue)
        Repeat
            DeleteQueue(vertex, Queue)
            visited[vertex] = true
            Visit(vertex)
            for all vertexW adjacent to vertex do
                if not visited[vertexW] then
                    AddQueue(w)
        Until Empty(Queue)
}
```

CHAPTER VII

DESCRIPTION OF RESPER

REduced Shared PERmuted (RESPER) Kronecker Decision Diagram is the synthesis algorithm for the calculation of the Permuted Kronecker Reed-Muller Tree for a given completely specified Boolean function having the minimal number of AND/EXOR gates and multiplexers. RESPER consists of three parts: trivial function, expansion selection and decomposition. The trivial function is used for the realization of Boolean functions as cascade circuits, where only one next level module is allowed or no module at all. Expansion selection option determines an appropriate expansion to be chosen for that module in that level. RESPER applies breadth first search alogrithm to do the expansion and variable selection. Finally, the RESPER is developed to provide a Reduced Permuted Pseudo Kronecker Reed-Muller Tree that is especially suited for the technology mapping to the AT 6000 series of Atmel.

VII.1. TRIVIAL FUNCTION REALIZATION

The basic principle of the level by level minimization algorithm from [9,7] is to find the minimal number of next level modules for a given level. This approach will be adopted here. A similar principle is used for the realization of Boolean functions as cascade circuits where only one next level module is allowed or no module at all.

There exist six basic conditions for which a next module is redundant.

Condition 1: $f_{x_i} = 0$

If this condition is applied to equation 6, one will get $f = \overline{x_i} \cdot f_{\overline{x_i}}$.

Proof:

$$\begin{aligned} f &= f_{x_i} \oplus \overline{x_i} \cdot [f_{x_i} \oplus f_{\overline{x_i}}] \\ &= 0 \oplus \overline{x_i} \cdot [0 \oplus f_{\overline{x_i}}] \\ &= \overline{x_i} \cdot f_{\overline{x_i}} \end{aligned}$$

We can use an AND gate with one negated input to implement this function, instead of using AND/EXOR gate which has the longest delay in the AT 6000 series. The AT 6000 series does not provide "0" as one of its inputs.

Condition 2: $f_{\overline{x_i}} = 0$

If this condition is applied to equation 5, it will get $f = x_i \cdot f_{x_i}$.

Proof:

$$\begin{aligned} f &= f_{\overline{x_i}} \oplus x_i \cdot [f_{x_i} \oplus f_{\overline{x_i}}] \\ &= 0 \oplus x_i \cdot [0 \oplus f_{\overline{x_i}}] \\ &= x_i \cdot f_{\overline{x_i}} \end{aligned}$$

We can use an AND gate with one negated input to implement this function, instead of using AND/EXOR gate which has the longest delay in AT 6000 series. And the AT 6000 series does not provide "0" as one of its inputs.

Condition 3: $f_{x_i} = 1$

If this condition is applied to equation 4, it will get $f = x_i \cdot 1 \oplus \overline{x_i} \cdot f_{\overline{x_i}}$.

Proof:

$$f = x_i \cdot f_{x_i} \oplus \overline{x_i} \cdot f_{\overline{x_i}}$$

$$= x_i \cdot 1 \oplus \overline{x_i} \cdot f_{\overline{x_i}}$$

We are able to use one wire less for the inputs to the multiplexer, since the AT 6000 series allows us to select "1" for one of the inputs.

<u>Condition 4:</u> $f_{\overline{x_i}} = 1$

If this condition is applied to equation 4, it will get $f = x_i \cdot f_{x_i} \oplus \overline{x_i} \cdot 1$.

Proof:

$$f = x_i \cdot f_{x_i} \oplus \overline{x_i} \cdot f_{\overline{x_i}}$$

$$= x_i \cdot f_{x_i} \oplus \overline{x_i} \cdot 1$$

We are able to use one wire less for the inputs to multiplexer, since the AT 6000 series allows us to select "1" for one of the input.

<u>Condition 5:</u> a data-input function is identical to another data-input function to a multiplexer in the same level of the tree circuit

$$f_{x_i} = f_{\overline{x_j}}$$

If this condition is applied to the equation 5, it gives $f_{\overline{x_i}}$ and f_{x_i}.

Proof:

$$f = f_{\overline{x_i}} \oplus x_i \cdot [f_{x_i} \oplus f_{\overline{x_i}}]$$

$$= f_{\overline{x_i}} \oplus x_i \cdot [f_{\overline{x_i}} \oplus f_{\overline{x_i}}]$$

$$= f_{\overline{x_i}} \oplus x_i \cdot 0$$

$$= f_{\overline{x_i}} = f_{x_i}$$

If this condition is applied to the equation 6, it gives $f_{\overline{x_i}}$ and f_{x_i}.

Proof:

$$f = f_{x_i} \oplus \overline{x_i} \cdot [f_{x_i} \oplus f_{\overline{x_i}}]$$

$$= f_{\overline{x_i}} \oplus \overline{x_i} \cdot [f_{\overline{x_i}} \oplus f_{\overline{x_i}}]$$

$$= f_{\overline{x_i}} \oplus \overline{x_i} \cdot 0$$

$$= f_{\overline{x_i}} = f_{x_i}$$

<u>Condition 6:</u> a data-input function is the complement of another data-input function to a multiplexer in the same level of the tree circuit

$$f_{x_i} = \overline{f_{\overline{x_j}}}$$

If this condition is applied to equation 5, the resultant function will be $f = f_{\overline{x_i}} \oplus x_i$.

Proof:

$$f = f_{\overline{x_i}} \oplus x_i \cdot [f_{x_i} \oplus f_{\overline{x_i}}]$$

$$= f_{\overline{x_i}} \oplus x_i \cdot [\overline{f_{\overline{x_i}}} \oplus f_{\overline{x_i}}]$$

$$= f_{\overline{x_i}} \oplus x_i \cdot 1$$

$$= f_{\overline{x_i}} \oplus x_i$$

If this condition is applied to equation 6, the result will be $f = f_{x_i} \oplus \overline{x_i}$.

Proof:

$$f = f_{x_i} \oplus \overline{x_i} \cdot [f_{x_i} \oplus f_{\overline{x_i}}]$$

$$= f_{x_i} \oplus \overline{x_i} \cdot [\overline{f_{\overline{x_i}}} \oplus f_{\overline{x_i}}]$$

$$= f_{x_i} \oplus \overline{x_i} \cdot 1$$

$$= f_{x_i} \oplus \overline{x_i}$$

As we see, it will give less wire connections and less modules for the next level.

In most algorithms only the first five conditions are taken into consideration to decrease the number of next level modules. The case of a data input function being the complement of another data input function has not been taken into consideration in any synthesis algorithm. The advantage of the presented method is, that it also verifies Condition 6. The complement function can be easily realized by an inverter logic cell as shown in Figure 25. The pseudo code of trivial realization is described in Fig. 24.

VII.2. VARIABLE AND EXPANSION SELECTION

The size of the BDD of a function is sensitive to the ordering of the input variables. A human with some understanding of the problem domain can generally choose an appropriate ordering without great difficulty for a small function from the Karnaugh map. It seems quite likely that using a small set of heuristics, a computer program itself could select an adequate ordering most of the time. Heuristic is any rule that directs the search. The construction of BDD starts from a minimized SOP expression of each function and performs successive Shannon decompositions according to the order of splitting variables. Some heuristics have been proposed to find a good order. They are based either on the analysis of an existing multilevel netlist [2] or on the number of occurances of the variables [3].

There has been a tremendous effort for determining a good variable ordering[12,13]. The researcher adopted the synthesis algorithm form [9] for the RESPER. To reduce the solution space for a large function to a space that is computationally feasible, the heuristic searching algorithm allows all three decomposition choices. The heuristic for the variable selection is to select the variable that will obtain less modules in the next level. In order to obtain the result which is as close as possible to the exact solution, the program starts to check all possible variables at each node in each level. Selecting the variable is determined by the set of conditions defined in the previous section.

```
// module 1     : one bit multiplexer
// module 4     : two-inputs XOR
// module 6     : two-inputs AND
// module 7     : two-inputs AND with one of the input negated
// temp_module : a sub-routine to store the result function in a temperaory place
                until that level will be computed.
//trivial      : a flag to indicate any of these conditions is met

Input:function f,
        function fnot,
        function g.  /* $g = f_x \oplus f_{\bar{x}}$ */

Output: trivial,
        select_module.

check_trivial(f, fnot, g, trivial, select_module)
{
        if(g == 0)
        /* fx == fxnot */
                create temp_module();
        if(g == 1)
        /* fx == fxnot */
                select_module 4;
        if(fx == 0)
        /* select equation 6    */
                select_module 7;
        if(fx == 1)
        /* select equation 4    */
                select_module 1;
        if(fxnot == 0)
        /* select equation 5    */
                select_module 6;
        if(fxnot == 1)
        /* select equation 4    */
                select_module 1;
}
```

Figure 24. Pseudo-code of condition checking.

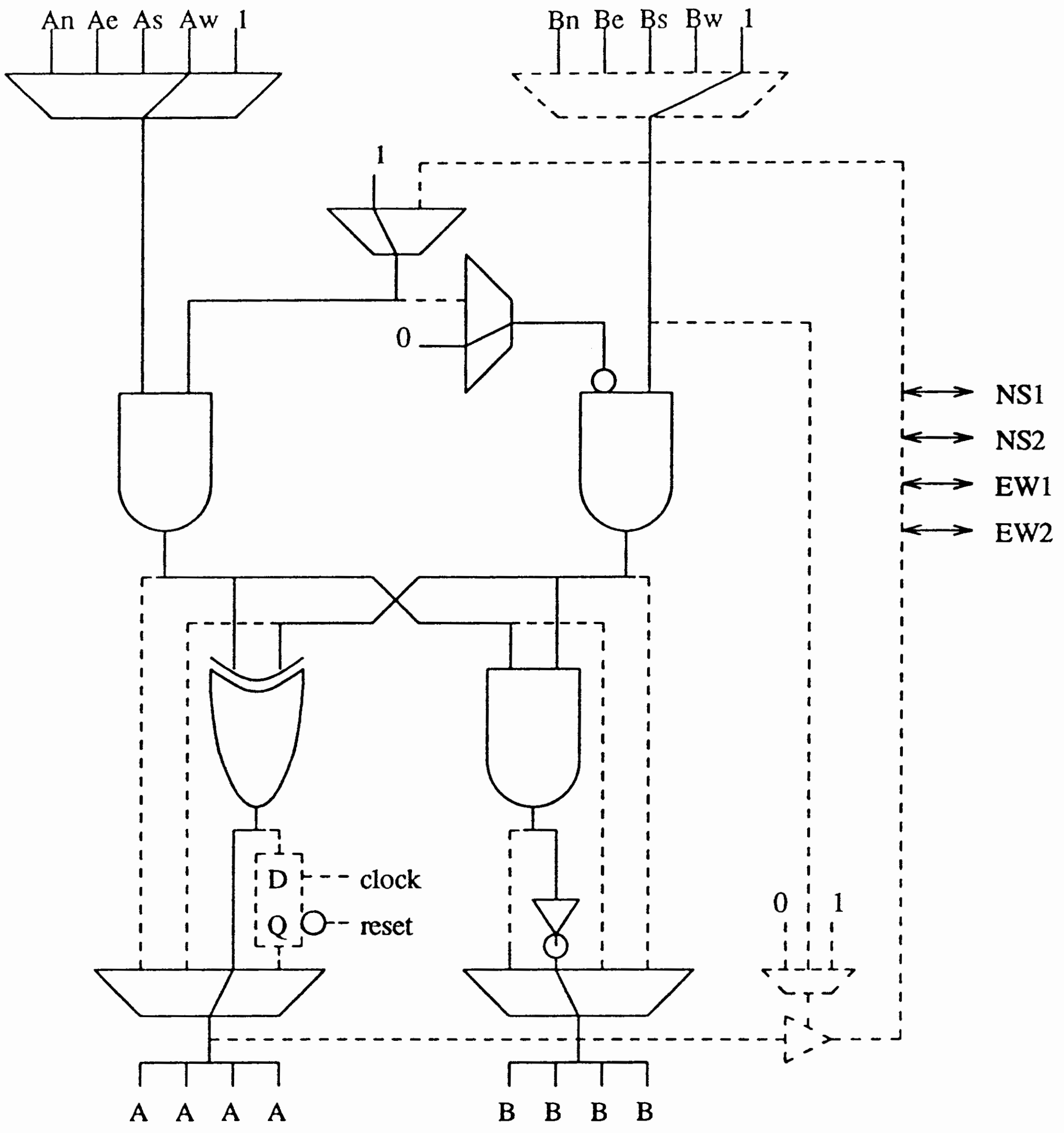


Figure 25. Cell Configuration: inverter.

The expansion selection is limited by the modules availability and their delay times. In AT 6000 series two modules are provided which fit two of the Davio Expansions. One of the modules is a two input multiplexer which is good for equation 4 and the other is the AND/XOR which is good for equation 5. If equation 6 is used we need to add an inverter to the AND/XOR. The time delay is also an important factor for the choice. If there is a tie for the Equation 5 and Equation 6, we need to choose equation 5 since an inverter needs to be added. An added inverter will increase the number of levels of the tree.

The selection of an appropriate variable also creates the backbone for selecting an appropriate expansion. The expansion selections also are using the same set of conditions as the variable selection is. If condition 3 or condition 4 is met, then expansion 4 will be selected. If condition 2 or condition 6 is met, then expansion 5 will be chosen. If condition 1 is met, then equation 6 is chosen. If condition 5 is met, either equation 5 or 6 will create the same result. For condition 6, the program will select either equation 5 or equation 6. Since the objective for this program is to minimize the delay and area, equation 5 will be chosen if condition 6 is met. If condition 5 is met during the expansion and variable selection, the program will stop searching and will select equation 5 for that module. The reason is that this function is independent of the chosen variable, and it does not need a module to represent this function at this level. However, if none of those conditions had been detected, the cost of each expression for each selected variable is calculated. Whichever combination of expansion type and variables provides the least number of cubes will be chosen. This evaluation is performed for each input variable for every of the output functions.

The pseudo code of variable and expansion selection is illustrated in Fig. 24.

VII.3. SHARED FUNCTIONAL DECISION DIAGRAM

FDD is a canonical representation of the functional domain (Fig.15). Each node of the FDD decides whether the product term belongs to the function or not. Each FDD has the following operation:

1. Deleting a node whose two edges direct to the same node.

2. Sharing isomorphic sub-graph.

```
// trivial         : a flag from the check trivial which declares one of the condition
                     has been met in this function from check_trivial
// selected_module  : a module has been selected during this current evaluation
// chosen_module  : the chosen module after the evaluation
// situation       : a flag to state whelther a condition has been met or not during
                     the evaluation
// selected_literal : the select literal for the decomposition
// k              : one of the input literal

Input:      trivial
       selected_module

Output:     selected_literal
       chosen_module

select_expansion(select_literal, trivial, select module)
{
       if (trivial == true)
         if (situation == trivial)
           if (the size of previous chosen module is greater than the select module){
              chosen_module = selected_module;
              size of chosen module = size of the select module;
              selected_literal = k;
           }
         else{
           chosen_module = selected_module;
           size of chosen module = size of the select module;
           selected_literal = k;
           situation = trivial;
         }
       else if (situation == trivial)
         if (the size of previous chosen module is greater than the select module){
           chosen_module = selected_module;
           size of chosen module = size of the select module;
           selected_literal = k;
         }
}
```

Figure 26. Pseudo-code of variable and expansion selection.

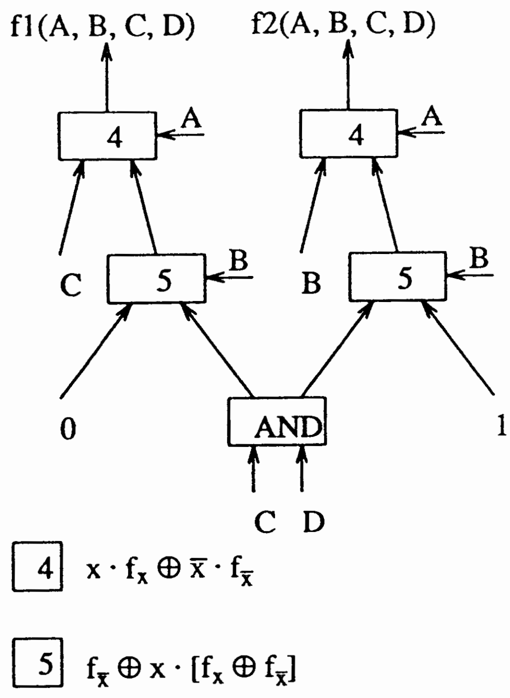

f1(A, B, C, D)
f2(A, B, C, D)
4
A
4
A
C
5
B
B
5
B
0
AND
1
C D
4 $x \cdot f_x \oplus \bar{x} \cdot f_{\bar{x}}$
5 $f_{\bar{x}} \oplus x \cdot [f_x \oplus f_{\bar{x}}]$

Figure 27. Shared Functional Decision Diagram.

Multiple FDDs can be joined into a single SFDD which consists of the FDDs sharing their subgraphs (Fig 27). In other words, two isomorphic subgraphs do not coexist in the SFDD. In SFDD, there is an input inverter added (Fig 28). Its purpose is to swap a positive edge and negative edge at the next node. By using this input inverter, SFDD will not only reduce those isomorphic subgraphs but also those subgraphs which are inverses of the others. This constraint brings about the following advantages to manipulate a completely specifed Boolean functions.

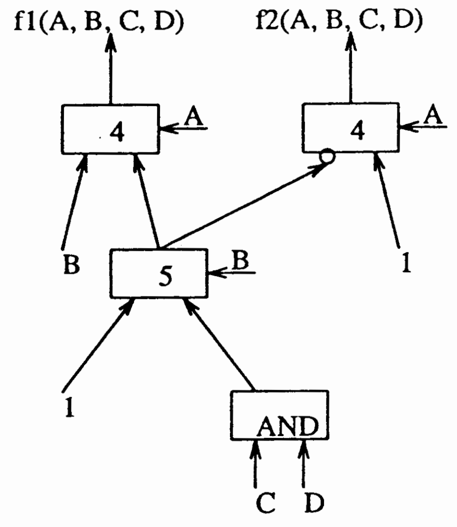
1. The equivalence between two functions can be checked by

$$F_n \oplus F_m = 0 \qquad n \neq m$$

2. The inversion between two functions can be checked by

$$F_n \oplus F_m = 1 \qquad n \neq m$$

3. By sharing sub-graph we can compactly represent many functions together.



$\boxed{4}$ $x \cdot f_x \oplus \bar{x} \cdot f_{\bar{x}}$

$\boxed{5}$ $f_{\bar{x}} \oplus x \cdot [f_x \oplus f_{\bar{x}}$

Figure 28. input inverter.

The pseudo code of finding isomorphic nodes is described in Fig. 29.

VII.4. RESPER IMPLEMENTATION

The algorithms described above form the core of the decomposition. The task of these subroutines is to assist the whole program to choose an optimal variable order combined with a suitable expansion, to create a graph without coexisting subgraphs. The program reads in the disjoint ON cubes written in PLA format. If the input data include n output functions, the program divides an n-output functions into n single-output functions. Each output function is stored in n different modules. Starting from module[0], the program computes all input literals and searches for the best selected variable based on those six conditions. After the selected variable is chosen, it generates a module[n+1] and checks for an isomorphic module. If the module[n+1] is isomorphic with other module, the module[n+1] will be eliminated. The program will go on to module[1], module[2] until there isn't any isomorphic module left. The pseudo-code of the whole program is presented in Fig.30-36, and it is followed by an example which realizes one of the selected MCNC benchmark functions. It shows only the evaluation of the first level of the permuted KDD.

```
//end:           total number of modules in that level
//module:        module
//total_module:  total number of modules in the tree.
//match:         the modules which match

Input:    Module[]
          total_module
          end

Output:   match  //flag match, inverse, no match

find_isomorphic(module, total_module, match, end)
{
    for i = 0 to end
    {
        if (module[total_module] xor module[i] is equal to 0){
        /* module[total] is matched with module[i] */
            match = i;
            return 1;
        }else{
        if (module[total_module] xor module[i] is equal to 1)
        /* module[total] is matched with the inverse of module[i] */
            match = i;
            return 0;
        }else
            return 2;
    }
}
```

Figure 29. Pseudo-code of finding isomorphic node.

```
//i            : input literal counter
//j            : output literal counter
//list         : a list of input cubes
//total_module : total modules of present level
//compute_f    : a subroutine to generate $f_x$
//compute_fnot : a subroutine to generate $f_{\bar{x}}$
//compute_g    : a subroutine to do f exor fnot
//match        : the number of that module match with
//flag         : an indicator to indicate there is a matched function
//end          : total modules of previous level

main()
{
 Get the input function

 for i = 0 to (outLiterals - 1)
     create_module(module, i, list);

 while (done == FALSE){
     for i = 0 to end{
         for j = 0 to inLiterals{
                 f = compute_f(module[i], j);
                 fnot = compute_fnot(module[i], j);
                 g = compute_g(f, fnot);
                 check_trivial(f, fnot, g, trivial, select_module);
                 select_expansion(variable, trivial, select_module);
         } /* end of j loop */

         f = compute_f(module[i], variable);
         fnot = compute_fnot(module[i], variable);
         g = compute_g(f, fnot);

         switch (select_module){
                 case 1:  pseudo-code in Fig. 27
                 case 2:  pseudo-code in Fig. 28
                 case 3:  pseudo-code in Fig. 29
                 case 4:  pseudo-code in Fig. 30
                 case 6:  pseudo-code in Fig. 31
                 case 7:  pseudo-code in Fig. 32
     }
 }
     if (total_module == 0)
         end = total_module;
     else
         done = TRUE;
}
```

Figure 30. Pseudo-code of RESPER.

```
case 1: /* select equation 4 */

        create(module, total_module, f);
        flag = find_isomorphic(module, total_module, match, end);

        if (flag == 0)
        /* module[match] is inversed of module[total_module] */
                connect to the module[match] with negated input;
        else if (flag == 1)
        /* module[match] is matched with module[total_module] */
                connect to the module[match];
        else
        /* there isn't any match */
                total_module++;

        create(module, total_module, fnot);
        flag = find_isomorphic(module, total_module, match, end);

        if (flag == 0)
        /* module[match] is inversed of module[total_module] */
                connect to the module[match] with negated input;
        else if (flag == 1)
        /* module[match] is matched with module[total_module] */
                connect to the module[match];
        else
        /* there isn't any match */
                total_module++;
```

Figure 31. Pseudo-code of case 1.

```
case 2: /* select equation 6 */

        create(module, total_module, fnot);
        flag = find_isomorphic(module, total_module, match, end);

        if (flag == 0)
        /* module[match] is inversed of module[total_module] */
                connect to the module[match] with negated input;
        else if (flag == 1)
        /* module[match] is matched with module[total_module] */
                connect to the module[match];
        else
        /* there isn't any match */
                total_module++;

        create(module, total_module, g);
        flag = find_isomorphic(module, total_module, match, end);

        if (flag == 0)
        /* module[match] is inversed of module[total_module] */
                connect to the module[match] with negated input;
        else if (flag == 1)
        /* module[match] is matched with module[total_module] */
                connect to the module[match];
        else
        /* there isn't any match */
                total_module++;
```

Figure 32. Pseudo-code of case 2.

```
case 3: /* select equation 5 */

        create(module, total_module, f);
        flag = find_isomorphic(module, total_module, match, end);

        if (flag == 0)
        /* module[match] is inversed of module[total_module] */
                connect to the module[match] with negated input;
        else if (flag == 1)
        /* module[match] is matched with module[total_module] */
                connect to the module[match];
        else
        /* there isn't any match */
                total_module++;

        create(module, total_module, g);
        flag = find_isomorphic(module, total_module, match, end);

        if (flag == 0)
        /* module[match] is inversed of module[total_module] */
                connect to the module[match] with negated input;
        else if (flag == 1)
        /* module[match] is matched with module[total_module] */
                connect to the module[match];
        else
        /* there isn't any match */
                total_module++;
```

Figure 33. Pseudo-code of case 3.

```
case 4: /* select equation 5 with g = 1*/

        create(module, total_module, fnot);
        flag = find_isomorphic(module, total_module, match, end);

        if (flag == 0)
        /* module[match] is inversed of module[total_module] */
                connect to the module[match] with negated input;
        else if (flag == 1)
        /* module[match] is matched with module[total_module] */
                connect to the module[match];
        else
        /* there isn't any match */
                total_module++;
```

Figure 34. Pseudo-code of case 4.

```
case 6: /* select equation 1 with f = 0 */

        create(module, total_module, fnot);
        flag = find_isomorphic(module, total_module, match, end);

        if (flag == 0)
        /* module[match] is inversed of module[total_module] */
                connect to the module[match] with negated input;
        else if (flag == 1)
        /* module[match] is matched with module[total_module] */
                connect to the module[match];
        else
        /* there isn't any match */
                total_module++;
```

Figure 35. Pseudo-code of case 6.

```
case 7: /* select equation 1 with fnot = 0 */

        create(module, total_module, f);
        flag = find_isomorphic(module, total_module, match, end);

        if (flag == 0)
        /* module[match] is inversed of module[total_module] */
                connect to the module[match] with negated input;
        else if (flag == 1)
        /* module[match] is matched with module[total_module] */
                connect to the module[match];
        else
        /* there isn't any match */
                total_module++;
```

Figure 36. Pseudo-code of case 7.

Example6:

An example of the execution of the algorithm is shown below. The selected input function is one of the MCNC benchmarks, "adr2", which contains 4 inputs and 3 outputs. This function is a completely specified disjoint function.

Step 1: The input function

input literal = 4

output literal = 3

.inputs v0 v1 v2 v3

.outputs v4.0 v4.1 v4.2

001x 100

0101 100

0110 100

1100 100

1111 100

100x 100

x0x1 010

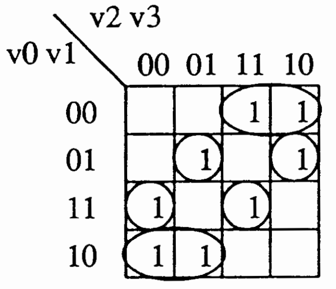
x1x0 010

0111 001

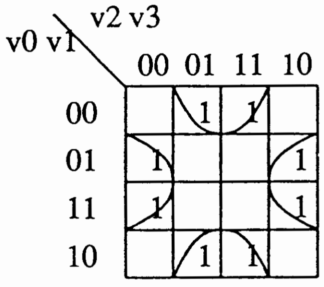
1101 001

1x1x 001

Step 2: Create output modules from the list



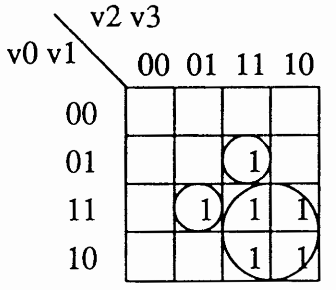
$$\begin{bmatrix} 001x \\ 0101 \\ 0110 \\ 1100 \\ 1111 \\ 100x \end{bmatrix}$$



$$\begin{bmatrix} x0x1 \\ x1x0 \end{bmatrix}$$

This is module[0] – v4.0

This is module[1] – v4.1



$$\begin{bmatrix} 0111 \\ 1101 \\ 1x1x \end{bmatrix}$$

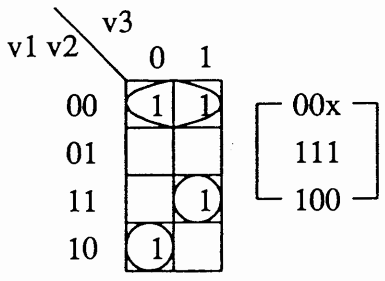
This is module[2] – v4.2

total_module = 3;

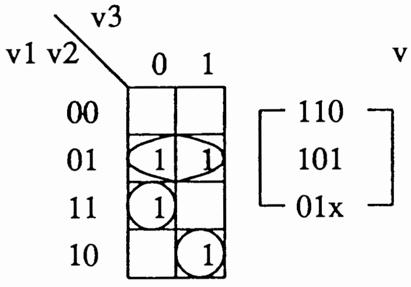
Step 3: Loop

i = 0; j = 0;

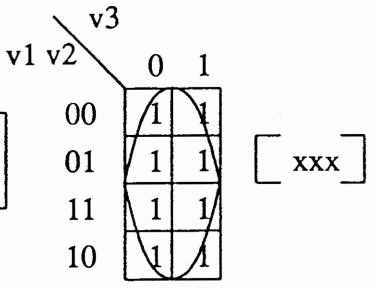
/* doing module[0], literal 0 (v0) */



function f_{v0}



function $f_{\overline{v0}}$



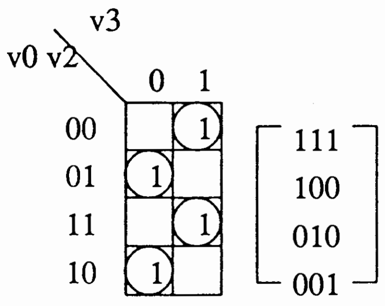
function g

Step 3a: check_trivial
trivial = TRUE;
selected_module = 4;

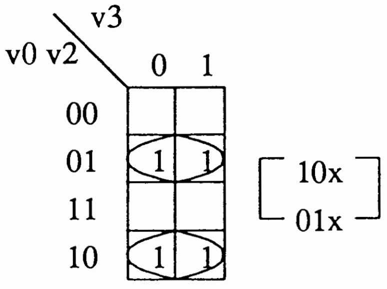
Step 3b: select_expansion
chosen_module = 4;
size of chosen module = 3;
selected_literal = 0;
situation = TRUE;

i = 0; j = 1;

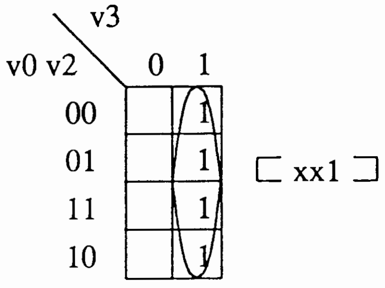
/* doing module[0], literal 1 (v1) */



function f_{v1}



function $f_{\overline{v1}}$



function g

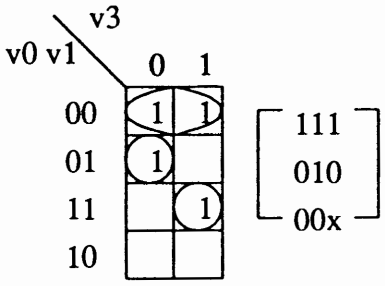
Step 3a: check_trivial

trivial = FALSE;

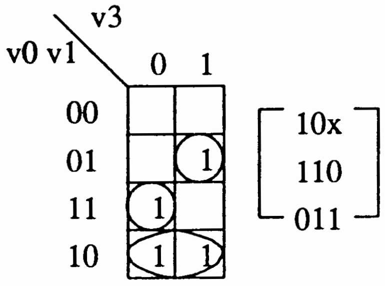
Step 3b: select_expansion
chosen_module = 4;
size of the chosen module = 3;
selected_literal = 0;
situation = TRUE;

i = 0; j = 2;

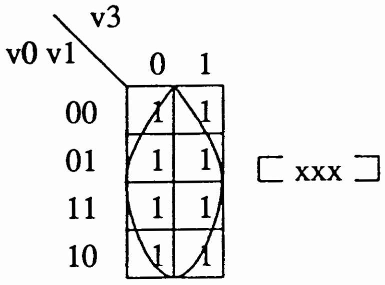
/* doing module[0], literal 2 (v2) */



function f_{v2}



function $f_{\overline{v2}}$

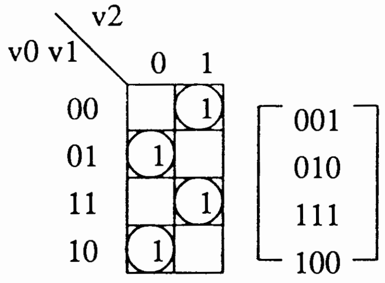


function g

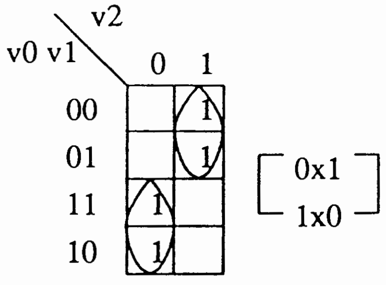
Step 3a: check_trivial
trivial = TRUE;
selected_module = 4;

Step 3b: select_expansion
chosen_module = 4;
size of the chosen module = 3;
selected_literal = 0;
situation = TRUE;

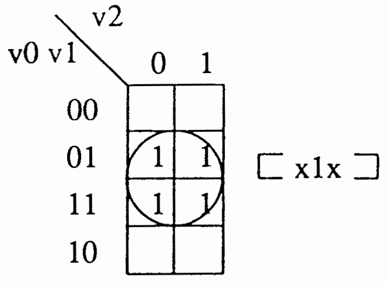
i = 0; j = 3; /* doing module[0], literal 3 (v3) */



function f_{v3}



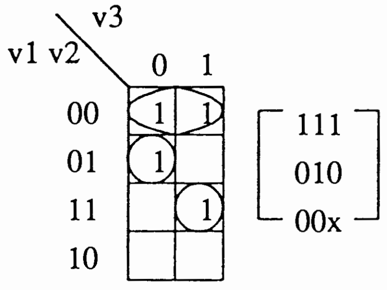
function $f_{\overline{v3}}$



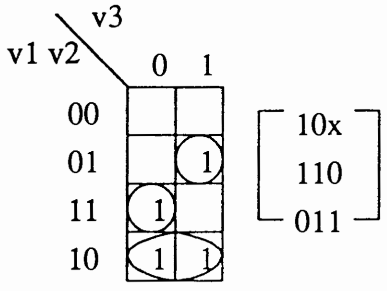
function g

Step 3a: check_trivial
trivial = FALSE;

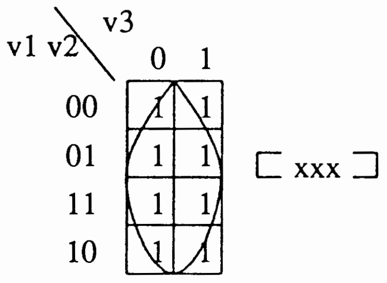
Step 3b: select_expansion
chosen_module = 4;
size of the chosen module = 3;
selected_literal = 0;
situation = TRUE;



f = compute_f(module[0], 0)



fnot = compute_fnot(module[0], 0)

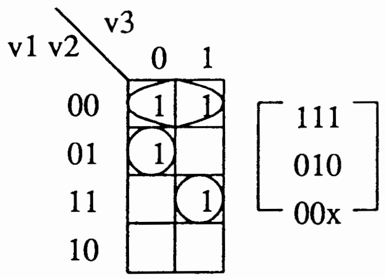


g = compute_g(f, fnot)

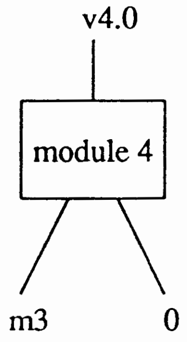
switch (4)
create_module(module, 3, fnot);

flag = 2;

module[3]:

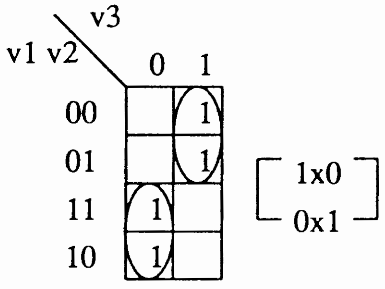


```
total_module = 4;
/*print output*/
 .names v0 m3 v4.0
 10 1
 01 1
```

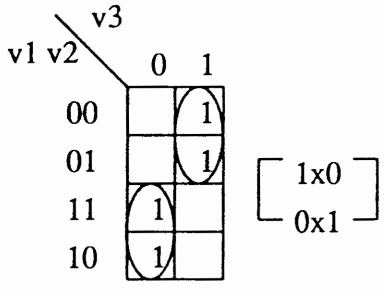


i = 1; j = 0;

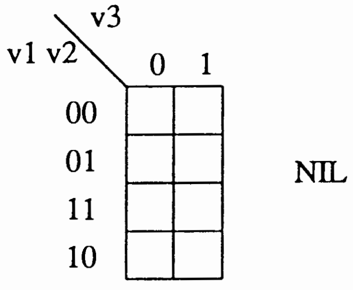
/* doing module[1], literal 0 (v0) */



function f_{v0}



function $f_{\overline{v0}}$



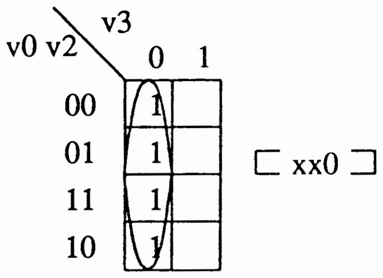
function g

Step 3a: check_trivial
trivial = TRUE;
selected_module = 5;

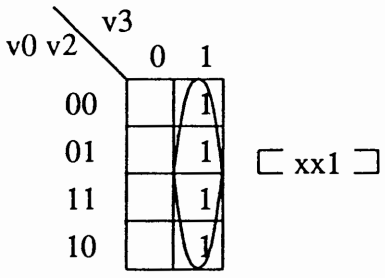
Step 3b: select_expansion
chosen_module = 5;
size of the chosen module = 2;
selected_literal = 0;
situation = TRUE;

i = 1; j = 1;

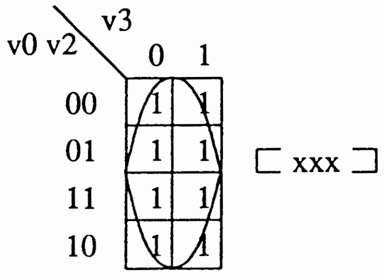
/* doing module[1], literal 1 (v1) */



function f_{v1}



function $f_{\overline{v1}}$



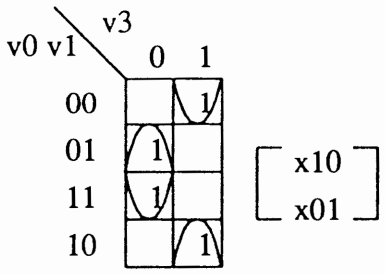
function g

Step 3a: check_trivial
trivial = TRUE;
selected_module = 4;

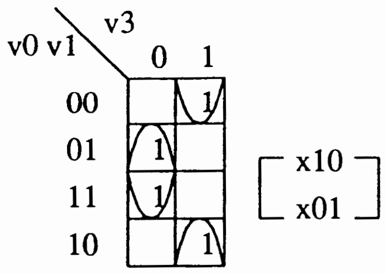
Step 3b: select_expansion
chosen_module = 5;
size of the chosen module = 2;
selected_literal = 0;
situation = TRUE;

i = 1; j = 2;

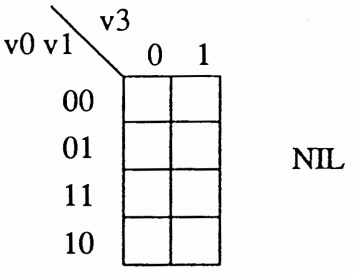
/* doing module[1], literal 2 (v2) */



function f_{v2}



function $f_{\overline{v2}}$



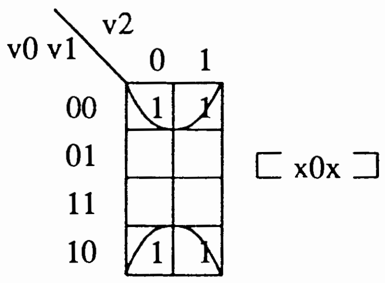
function g

Step 3a: check_trivial
trivial = TRUE;
selected_module = 5;

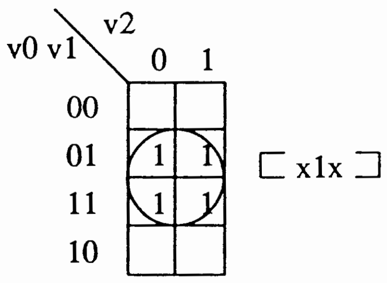
Step 3b: select_expansion
chosen_module = 5;
size of the chosen module = 2;
selected_literal = 0;
situation = TRUE;

i = 1; j = 3;

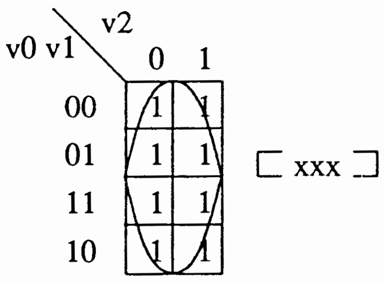
/* doing module[1], literal 3 (v3) */



function f_{v3}



function $f_{\overline{v3}}$

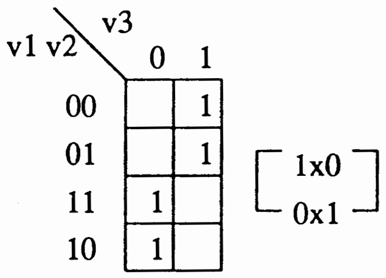


function g

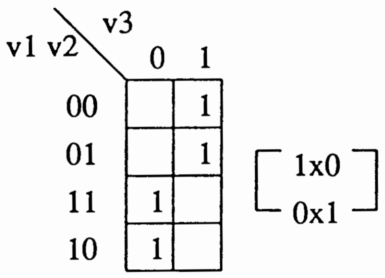
Step 3a: check_trivial
trivial = TRUE;
selected_module = 5;

Step 3b: select_expansion
chosen_module = 5;

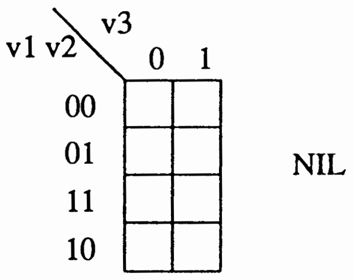
size of the chosen module = 2;
selected_literal = 0;
situation = TRUE;



f= compute_f(module[1], 0)

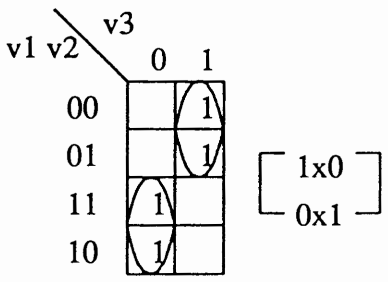


fnot = compute_fnot(module[1], 0)



g = compute_g(f, fnot)

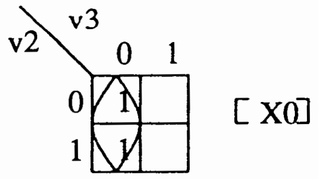
switch(5)
create_module(temp_module, 0, f)
temp_module[0]:



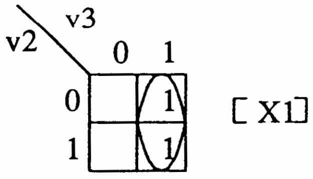
temp_module[0]

i = 0; j = 0;

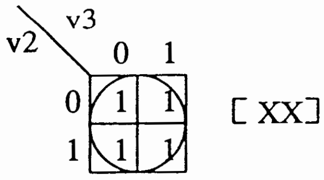
/* doing temp_module[0], literal 0 (v0) */



function f_{v1}



function $f_{\overline{v1}}$



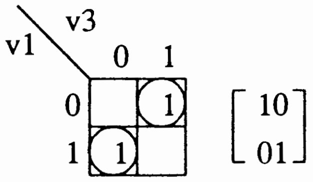
function g

Check_trivial:
trivial = TRUE;
selected_module = 4;

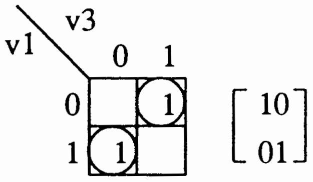
Select_expansion:
chosen_module = 4;
size of the chosen module = 1;
selected_literal = 0;
situation = TRUE;

i = 0; j = 1;

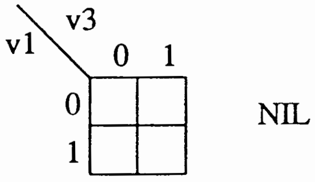
/* doing temp_module[0], literal 1 (v1) */



function f_{v2}



function $f_{\overline{v2}}$



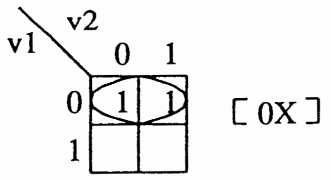
function g

Check_trivial:
trivial = TRUE;
selected_module = 5;

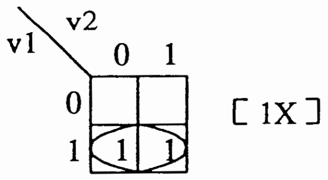
Select_expansion:
chosen_module = 5;
size of the chosen module = 2;
selected_literal = 1;
situation = TRUE;

i = 0; j = 2;

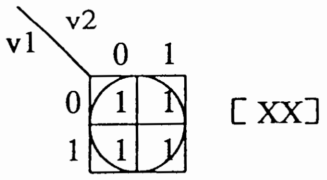
/* doing temp_module[0], literal 2 (v2) */



function f_{v3}



function $f_{\overline{v3}}$



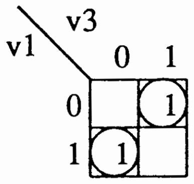
function g

Check_trivial:
trivial = TRUE;
selected_module = 5;

Select_expansion:
chosen_module = 5;
size of the chosen module = 2;
selected_literal = 1;
situation = TRUE;

switch (5)

create_module(temp_module, 1, f)
temp_module[1]



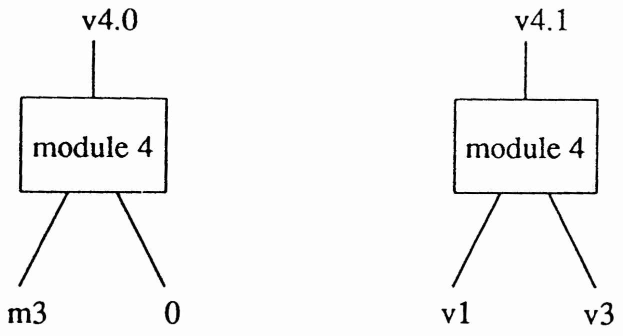
temp_module[1]

i = 1; j = 0;

/* doing temp_module[1], literal 0 (v0) */
f:
0 1
fnot:
1 1
g:
x 1

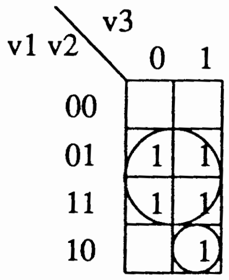
Check_trivial:

```
        trivial = TRUE;
        selected_module = 4;

        Select_expansion:
        chosen_module = 4;
        size of the chosen module = 1;
        selected_literal = 0;
        situation = TRUE;

i = 1; j = 1;

/* doing temp_module[1], literal 1 (v1) */
f:
0 1
fnot:
1 1
g:
x 1
        Check_trivial:
        trivial = TRUE;
        selected_module = 4;

        Select_expansion:
        chosen_module = 4;
        size of the chosen module = 1;
        selected_literal = 0;
        situation = TRUE;

f = compute_f(temp_module[1], 0);
   0 1
fnot = compute_fnot(temp_module[1], 0);
   1 1
g = compute_g(f, fnot);
   x 1

switch (4)
            create_module(module, 4, fnot);
            /*module has only 1 variable and this module is done;*/
            total_module = 4;
            /*print output*/
             .names v1 v3 v4.1
             10 1
             01 1
```



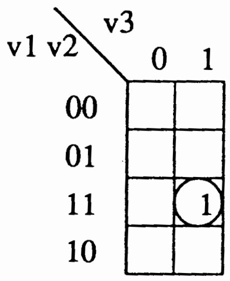
i = 2; j = 0;

/* doing module[2], literal 0 (v0) */



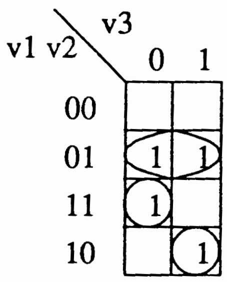
$\begin{bmatrix} x1x \\ 101 \end{bmatrix}$

function f_{v0}



$[\,111\,]$

function $f_{\overline{v0}}$



$\begin{bmatrix} 01x \\ 110 \\ 101 \end{bmatrix}$

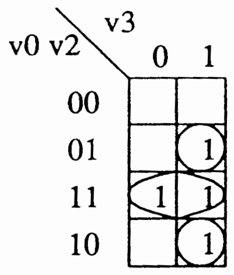
function g

Step 3a: check_trivial
trivial = FALSE;

Step 3b: select_expansion
chosen_module = 1;
size of chosen module = 3;
selected_literal = 0;
situation = FALSE;

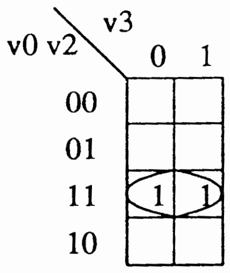
i = 2; j = 1;

/* doing module[2], literal 1 (v1) */



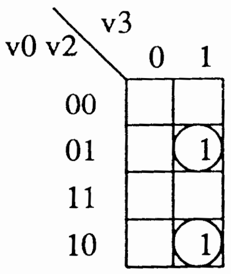
$$\begin{bmatrix} 11x \\ 011 \\ 101 \end{bmatrix}$$

function f_{v1}



$$\begin{bmatrix} 11x \end{bmatrix}$$

function $f_{\overline{v1}}$



$$\begin{bmatrix} 011 \\ 101 \end{bmatrix}$$

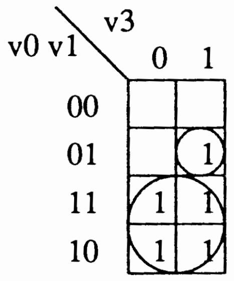
function g

Step 3a: check_trivial
trivial = FALSE;

Step 3b: select_expansion
chosen_module = 1;
size of chosen module = 3;
selected_literal = 0;
situation = FALSE;

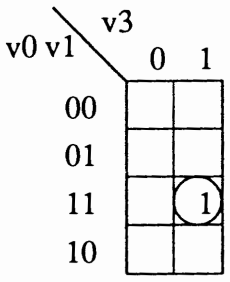
i = 2; j = 2;

/* doing module[2], literal 2 (v2) */



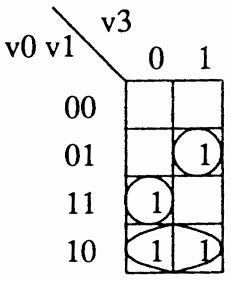
$$\begin{bmatrix} 1xx \\ 011 \end{bmatrix}$$

function f_{v2}



$$\begin{bmatrix} 111 \end{bmatrix}$$

function $f_{\overline{v2}}$



$$\begin{bmatrix} 10x \\ 110 \\ 011 \end{bmatrix}$$

function g

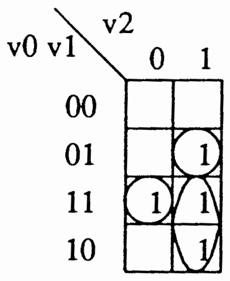
Step 3a: check_trivial
trivial = FALSE;

Step 3b: select_expansion
chosen_module = 1;
size of chosen module = 3;
selected_literal = 0;

situation = FALSE;

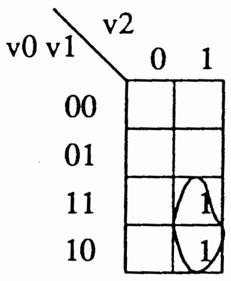
i = 2; j = 3;

/* doing module[2], literal 3 (v3) */



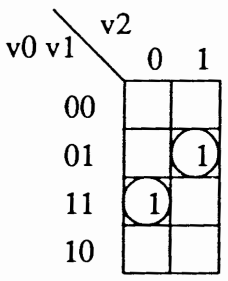
$$\begin{bmatrix} 011 \\ 110 \\ 1x1 \end{bmatrix}$$

function f_{v3}



$$\begin{bmatrix} 1x1 \end{bmatrix}$$

function $f_{\overline{v3}}$

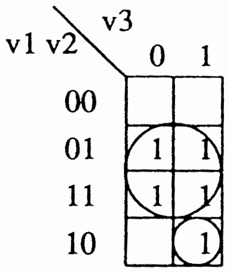


$$\begin{bmatrix} 011 \\ 110 \end{bmatrix}$$

function g

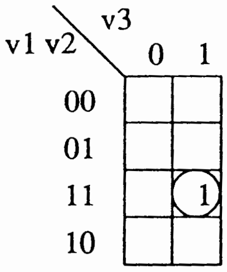
Step 3a: check_trivial
trivial = FALSE;

Step 3b: select_expansion
chosen_module = 1;
size of chosen module = 3;
selected_literal = 0;
situation = FALSE;



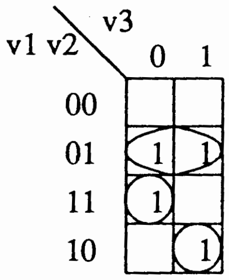
$\begin{bmatrix} x1x \\ 101 \end{bmatrix}$

f = compute_f(module[2], 0)



$[\,111\,]$

fnot = compute_fnot(module[2], 0)

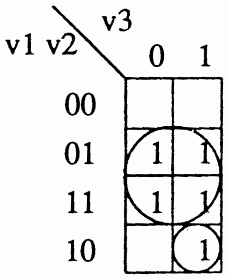


$\begin{bmatrix} 01x \\ 110 \\ 101 \end{bmatrix}$

g = compute_g(f, fnot)

switch (1)

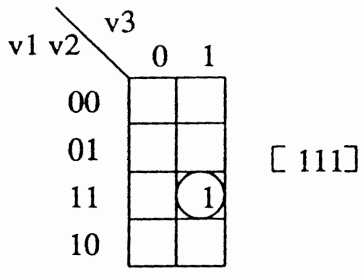
```
    create(module, 4, f);
    flag = 2;
       module[4]
```



$\begin{bmatrix} x1x \\ 101 \end{bmatrix}$

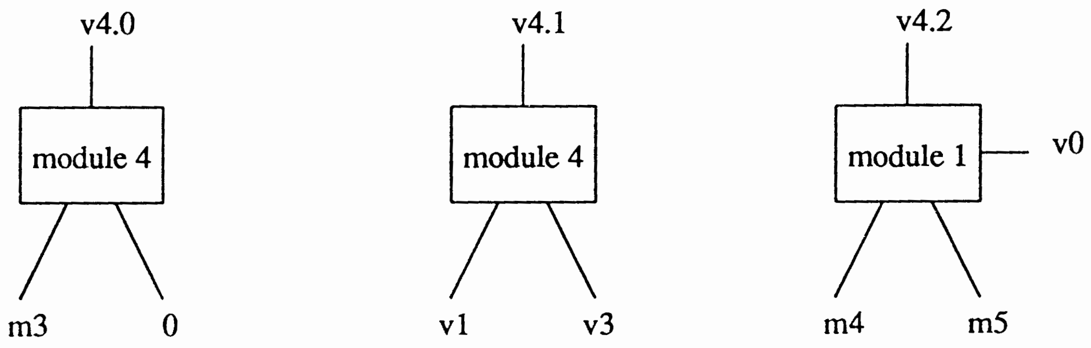
module [4]

```
    total_module = 5;

    create(module, 5, fnot);
    flag = 2;
       module[5]
```



module [5]

```
total_module = 6;
/*print output*/
 .names m4 m5 v4.2
 11- 1
 1-1 1
```



Step 4:

```
    begin = 3;
    end = 5;
    done = FALSE;
```

GoTo Step 3;

Repeat the same steps until there are no more modules created.

CHAPTER VIII

EVALUATION OF THE RESULTS

The researcher ran RESPER on a networked SUN 4/670MP workstation. The results are listed in Table II and III. All results are verified by the "verify" command of the *MIS-II* system. The procedure for verifying is:

```
resper  MCNC_file > output_file   \*  execute RESPER program
                                  MCNC_file: The name of the example file from
                                  the MCNC benchmark;
                                  output_file: The name of the output file which
                                  user assigned.

misII                             \*  enter misII program  *\
read_pla   MCNC_file              \*  read in the MCNC benchmark example file  *\
write_blif   MCNC_file.blif       \*  write the MCNC benchmark example file in
                                  blif format  *\
verify output_file MCNC_file.blif \*  verify  *\
```

The results listed under *TECHMAP* and *REMIT* are from [18] and [3] respectively. It also ran on a networked SUN 4/670MP workstation.

In Table II and III, *E/N* is the name of the example. *I/N* is the number of input variables. *O/N* is the number of output functions. *Modules* is the number of modules in the final mapped circuit. Level is the longest path that a signal must go from the primary input to the primary output in the circuit. *Time* is the running time which the program takes to generate the output. *C1, C2* and *C3* in Table II are the heuristics used by TECHMAP to select the expansion. It was mentioned in Chapter 3. From Table III, we observe that permuted tree search method alone in REMIT[3] will not create good results compared with TECHMAP[18] which was using heuristic C3. Based on Table II, the heuristic used should be the best heuristic among those three in TECHMAP. Based on the comparison from Table III, we observe that RESPER can generate relatively good result as TECHMAP for single output function. However, from Table II we observe that RESPER generates a better result on multi-output than TECHMAP. As it was mentioned in Chapter IV, the poor results were generated from TECHMAP is because of the method of chosen variable and expansion. TECHMAP based on the variable and expansion selection of the first single output function, and decomposed the other single output function. On the other hand, RESPER can generate good results is because it selects the best variable and expansion of each node. Although RESPER generates better result than TECHMAP, it has its weakness in minimizing function. We can observe from benchmark 5xpl, which is the case where TECHMAP beats RESPER because of its weakness. In RESPER, the size of the function is one of the criteria to select appropriate variable and expansion. Overall speaking, the permuted tree search method combined with shared reduced order approach will produce a good result in general cases.

TABLE II

COMPARISONS BETWEEN *RESPER*, AND *TECHMAP*

E/N	I/N	O/N	*RESPER*			*TECHMAP*				
			mod.	Level	time	C1	C2	C3	Level	time
adr2	4	3	4	3	0.1	6	6	6	3	0.1
b12	15	9	50	8	33.9	130	147	125	12	11.4
cc	21	20	42	6	49.4	117	117	117	15	24.7
conl	7	2	10	4	0.4	16	24	15	5	0.2
cu	14	11	27	9	11.7	73	73	78	10	1.5
inc	7	9	61	6	3.1	81	89	76	6	2.0
squar5	5	8	18	4	0.5	29	35	29	4	0.4
f51m	8	8	35	7	8.4	52	53	45	7	3.3
xor5	5	1	3	3	0.2	3	3	3	3	0.1
5xp1	7	10	55	6	5.3	51	54	44	6	2.6
rd53	5	3	12	4	0.7	13	14	12	4	0.3
bw	5	28	100	4	3.0	102	104	102	4	2.8
misex1	8	7	32	5	8.3	60	39	56	7	1.2
misex2	25	18	107	11	79.6	282	271	275	23	15.8
TOTAL			560			1015	1029	983		

TABLE III

COMPARISONS BETWEEN *RESPER*, *REMIT*, AND *TECHMAP*

E/N	I/N	O/N	*RESPER*		*REMIT*		*TECHMAP*	
			modules	Level	modules	Level	modules	Level
misex52.tt	6	1	10	5	19	5	10	5
5x1.tt	7	1	10	4	28	4	9	4
5x10.tt	7	1	6	4	16	4	6	4
5x2.tt	7	1	20	6	51	6	19	4
5x5.tt	7	1	6	6	10	6	7	6
5x6.tt	7	1	3	6	6	6	3	6
5x7.tt	7	1	2	6	6	6	2	6
con11.tt	7	1	5	6	17	6	8	6
con12.tt	7	1	5	6	11	6	7	6
misex21.tt	5	1	12	5	24	5	11	5
misex22.tt	5	1	10	5	19	5	9	5
misex23.tt	5	1	8	5	18	5	9	5
misex25	5	1	9	5	19	5	11	5
misex26	5	1	7	5	13	5	7	5
misex48	*6*	*1*	*14*	*5*	*24*	*5*	*12*	*5*
misex49	*6*	*1*	*9*	*5*	*19*	*5*	*10*	*5*
misex50	*6*	*1*	*8*	*5*	*20*	*5*	*8*	*5*
misex52	*6*	*1*	*10*	*5*	*19*	*5*	*10*	*5*
misex53	*6*	*1*	*7*	*5*	*14*	*5*	*7*	*5*
misex54	*6*	*1*	*16*	*5*	*23*	*5*	*12*	*5*
misex55	*6*	*1*	*9*	*5*	*20*	*5*	*8*	*5*
misex56	*6*	*1*	*8*	*5*	*20*	*5*	*9*	*5*
misex57	*6*	*1*	*9*	*5*	*19*	*5*	*11*	*5*
misex58	*6*	*1*	*7*	*5*	*12*	*5*	*7*	*5*
misex59	*11*	*1*	*23*	*11*	*244*	*11*	*20*	*11*
misex60	*11*	*1*	*11*	*11*	*67*	*11*	*11*	*11*
misex61	*11*	*1*	*13*	*11*	*69*	*11*	*13*	*11*
TOTAL			*257*		*827*		*257*	

CHAPTER IX

FUTURE WORK

There has been lots of research on Multi-level decomposition. Those have been mainly targeted at AND/OR function. The same theory algorithm was used in this thesis to decompose ESOP function[19, 21]. The only modification needed is the procedure to minimize the ESOP function, and the algorithm from [21] is used to make the program work for ESOP function. The following is a step by step example following the pseudo code from RESPER.

Step 1: input function

```
0x00 1
00xx 1
xxx1 1
0x1x 1
1101 1
```

Step 2: create output module

```
module[0]:
  0x00 1
  00xx 1
  xxx1 1
  0x1x 1
  1101 1
```

Step 3: Loop

```
do module[0];
i = 0; j = 0;
  f:
   xx1  1
   101  1
   fnot:
   1xx  1
   x11  1
   g:
   001  1
   1xx  1

Step 3a: check trivial
   trivial = FALSE;

Step 3b: select expansion
   select literal = 0
   select module = 1
   size of the module = 4

i = 0; j = 1;

   f:
   x11  1
   0xx  1
   fnot:
   001  1
   xx1  1
   g:
   101  1
   0xx  1

Step 3a: check trivial
   trivial = FALSE;

Step 3b : select expansion
   select literal = 0
   select module = 1
   size of the module = 4

i = 0; j = 2;

   f:
   01x  1
   xx1  1
   fnot:
   101  1
```

```
  01x  1
  g:
  101  1
  xx1  1

Step 3a: check trivial
  trivial = FALSE;

Step 3b: select expansion
  select literal = 0
  select module = 1
  size of the module = 4

i = 0; j = 3;

  f:
  111  1
  0x0  1
  x0x  1
  fnot:
  01x  1
  g:
  100  1
  xx1  1

Step 3a: check trivial
  trivial = FALSE;

Step 3b: select expansion
  select literal = 0
  select module = 1
  size of the module = 4

f = compute_f(module[0], 0);
fnot = compute_fnot(module[0], 0);
g = compute_g(f, fnot);

switch(1)
  create_module(module, 1, f);
    module[1]
      xx1  1
      101  1
  total_module = 2;

  create_module(module, 2, fnot);
    module[2]
      1xx  1
      x11  1
total_module = 3;

Step 3c: print output
    .names v0 m1 m2 m0

do module[1]

i = 0; j = 0;

    f:
    11  1
    fnot:
    x1  1
    g:
    01  1

Step 3a: check trivial
    trivial = FALSE

Step 3b: select expansion
    select_literal = 0
    select_module = 1
    size of the module = 2

i = 0; j = 1;

    f:
    x1  1
    fnot:
    01  1
    g:
    11  1

Step 3a: check trivial
    trivial = FALSE

Step 3b: select expansion
    select_literal = 0
    select_module = 1
    size of the module = 2

i = 0; j = 2;

    f:
    10  1
```

```
   xx  1
   fnot:
   NIL
   g:
   10  1
   xx  1

Step 3a: check trivial
   trivial = TRUE;
   select_module = 6;

Step 3b: select expansion
   select_literal = 2
   select_module = 6
   size of the module = 2

f = compute_f(module[1], 2);
fnot = compute_fnot(module[1], 2);
g = compute_g(f, fnot);

switch(6)
   create_module(module, 3, f);
      module[3]
         10  1
         xx  1
   total_module = 4;

Step 3c: print output
    .names v3 m3 m1

do module[2]

i = 0; j = 0;

   f:
   xx  1
   11  1
   fnot:
   11  1
   g:
   xx  1

Step 3a: check trivial
   trivial = TRUE;
   select_module = 4

Step 3b: select expansion
```

```
   select_literal = 0
   select_module = 4;
   size of module = 1;

i = 0; j = 1;

   f:
   1x  1
   x1  1
   fnot:
   1x  1
   g:
   x1  1

Step 3a: check trivial
   trivial = FALSE;

Step 3b: select expansion
   select_literal = 0
   select_module = 4;
   size of module = 1;

i = 0; j = 2;

   f:
   1x  1
   x1  1

   fnot:
   x1  1

   g:
   x1  1

Step 3a: check trivial
   trivial = FALSE;

Step 3b: select expansion
   select_literal = 0
   select_module = 4;
   size of module = 1;

f = compute_f(module[2], 0);
fnot = compute_fnot(module[2], 0);
g = compute(f, fnot);

switch (4)
```

```
   create_module(module, 4, fnot);
      module[4]
         11  1
   total_module = 5;

Step 3c: print output
   .names v1 m4 m2


do module[3]

i = 0; j =0;

   f:
   1  1
   fnot:
   x  1
   g:
   0  1

Step 3a: check trivial
   trivial = TRUE;
   select_module = 1;

Step 3b: select expansion
   select_literal = 0;
   select_module = 1;
   size of module = 1;

i = 0; j = 1;

   f:
   x  1
   fnot:
   0  1
   g:
   1  1

Step 3a: check trivial
   trivial = TRUE;
   select_module = 1;

Step 3b: select expansion
   select_literal = 0;
   select_module = 1;
   size of module = 1;
```

```
f = compute_f(module[3], 0);
fnot = compute_fnot(module[3], 0);
g = compute_g(f, fnot);

switch(1)
   create_module(module, 5, f);
   module has only one cube with 2 or less variable

Step 3c: print output
    .names v1 v2 m3

do module[4]
it has only 2 variables this module is done

Step 3c: print output
    .names v2 v3 m4
```

The researcher can also extend the SRKDD to an incompletely specified function. The method to reduce the amount of modules and levels is still an open research area.

CHAPTER X

CONCLUSION

The goal of this research is to generate a compact circuit realization as a multi-level circuit. The researcher focuses on minimizing the number of nodes in the network and minimize the length of the path. Two methods are presented to accomplish this goal. One is the permuted tree search method, to select a good variable and an appropriate expansion for each node. Another one is the shared reduce order approach, to reduce the number of nodes and levels.

In this thesis, a tutorial approach is used to explain several Decision Diagrams and the concept to permuted KFDD. An efficient algorithm for Permuted KFDD is also presented. The obtained results are very promising and motivate to further investigate KFDD and other permuted(free) diagram. There is a need to research for all FDD to find better heuristic to obtain circuit realization with the shortest path from input to output. And the RESPER can be generalized to incompletely specified functions.

Based on Table I, it clearly points out that new research areas and potential advantages of combining idea of permuting variables and adding expansion types. It seems that this is the direction of Decision Diagram development: more expansion types and no restriction on the order or the number of repetition of a variable.

FDDs have levels corresponding to input variables and Shannon expansion. And KFDDs have levels corresponding to input varaibles, at every level all nodes are expanded with respect to the same type of expansion: Shannon, DavioI and DavioII. It was proved that essential improvement in both the number of levels and the number of nodes are obtained when the nodes in a level of a diagram have various types of expansions. In this thesis, researcher observed from Table II that when a tree diagram has various types of expansions and input variables at each node will have a major decrease in terms of number of levels and nodes against FDDs and KFDDs.

From the experimental results, it is proved that the permuted tree search method will not generate as a good result as shared reduced order approach combined with some kind of heuristic. However, the permuted tree search method combined with shared reduced order approach will produce a good result in general cases. And the result shows that this approach is relatively better than REMIT and TECHMAP in terms of providing a compact circuit realization in the multi-level circuit.

Further extension to REPER will include mapping to other new celluler FPGAs, especially those from Motorola. Variants of the method can be also created including geometrically increase KFDD, which will improve the operation to inter-link placement and routing.

REFERENCES

[1] Concurrent Logic Inc., *CLi6000 Series Field Programmable Gate Arrays*, 1992.

[2] M Fujita, H. Fujisawa, and N. Kawato, "Evaluation and Improvements of Boolean Comparison Method Based On Binary Decision Diagram," *ICCAD 88,* pp. 2-5, 1988.

[3] L.F. Wu, and M.A. Perkowski, "Minimzation of Permuted Reed-Muller Trees for Cellular Logic Programmable Gate Arrays," *2nd Int. Workshop on FPGAs*, Vienna, Austria, Sept., 1992.

[4] A. Sarabi, and M.A. Perkowski, "Fast Exact and Quasi-Minimal Minimization of Highly Testable Fixed Polarity AND/EXOR Canonical Networks," *29th ACM/IEEE DAC*, pp. 30-35, June 1992.

[5] Ph.W. Besslich, and M.W. Riege, "An Efficient Program for Logic Synthesis of Mod-2 Sum Expressions," *Proc. Euro ASIC'91*, pp. 136-141, Paris, France, 1991.

[6] J.M. Saul, "An Algorithm for the Multi-Level Minimization of Mixed Polarity Reed-Muller Representation," *Proc. IEEE ICCAD'91*, pp. 634-637, Sept. 1991.

[7] I. Schaefer, and M.A. Perkowski, "Synthesis of Multi-Level Multiplexer Circuits for Incompletely Specified Boolean Functions with Mapping to Multiplexer Based FPGAs," *IEEE Trans. on CAD*, pp. 1655-1664, Nov. 1992.

[8] M. Davio, and A. Thayse, *Discrete and Switching Functions*, McGraw Hill, 1978.

[9] A.E.A. Almaini, and M.E. Woodward, "An Approach to the Control Variable Selection Problem for Universal Logic Modules," *Digital Processes*, vol. 3, pp. 189-206, 1977.

[10] R.E. Bryant, "Graph-Based Algorithm for Boolean Function Manipulation," *IEEE Trans. on Comp.*, vol. 38, no.8, pp. 667-691, Aug. 1986.

[11] D. Varma, and E.A. Trachtenberg, "Design Automation Tools for Efficient Implementation of Logic Functions by Decomposition," *IEEE Trans. on CAD*, vol. 8, pp. 901-916, Aug. 1989.

[12] K.M. Butler, D.E. Ross, R. Kapur, and M.R. Mercer, "Heuristics to Compute Variable Orderings for Efficient Manipulation of Ordered Binary Decision Diagrams," *28th ACM/IEEE DAC*, pp. 417-420, 1991.

[13] S.J. Friedman, and K.J. Supowit, "Finding the Optimal Variable Ordering for Binary Decision Diagram", *IEEE Trans. on Comp.*, vol 39, no. 5, pp. 710-713, May, 1990.

[14] S. Minato, N. Ishiura, and S. Yajima, "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation," *Proc. 27th ACM/IEEE DAC*, pp. 52-57, 1990.

[15] U. Kebschull , E. Schubert , and W. Rosenstiel , "Multilevel Logic Synthesis Based on Functional Decision Diagrams," *Proc. IEEE European Design Automation Conference*, pp 43-47, 1992.

[16] D.H. Green , and P.W. Foulk , "Adaptive Logic Trees for Use in Multilevel-Circuit Design", *Electr. Letters*, vol 5, pp. 83-84, 1969.

[17] D.H. Green , and M. Edkins , "Synthesis Procedures for Switching Circuits Representation in Generalised Reed-Muller Form over a Finite Field," *IEE Journal, Comp. and Dig. Techniques*, vol 1, no. 1, pp. 22-35, 1978.

[18] Ingo Schafer, "Orthogonal Expansions for Multilevel Logic Synthesis and the Technology Mapping to FPGAs," *IEE Technical Report*, Portland State University, 1992.

[19] M.A. Perkowski, "The Generalized Orthonormal Expansion of Functions With Multiple-Valued Inputs and Some of its Applications," *Proc. IEEE 22nd ISMVL'92*, pp. 442-450, Sendai, Japan, May 1992.

[20] M.A. Perkowski , and P.D. Johnson , "Canonical Multi-Valued Input Reed-Muller Trees and Forms," *Proc. 3rd NASA Symposium on VLSI Design*, pp. 11.3.1-11.3.13, Moscow, Idaho, October 1991.

[21] N. Song , and M.A. Perkowski , "EORCISM-MV-2: Minimization of Exclusive Sum Of Products Expressions for Multiple-Valued Input Incompletely Specified Functions," *Proc. 23rd ISMVL'93*, pp. 132-137, Dec 1992.

[22] S.B. Akers, "Binary Decision Diagram," *IEEE Trans. on Computers*, vol C-27, no. 6, pp. 509-516, June 1978.

[23] T. Besson , H. Bousouzou , M. Crates , and G. Saucier , "Synthesis on Multiplexer-based Programmable Devices Using (Ordered) Binary Decision Diagrams," *EUROASIC*, pp 8-13, Paris, France, June 1992.

[24] R.K. Brayton, G.D. Hachtel, and A.L. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis," *Proc. of the IEEE*, vol. 78, no. 2, pp. 264-300, Feb. 1990.

[25] M. Fujita, and Y. Matsunaga, "Multi-level Logic Minimization based on Minimal Support and its Application to the Minimization of Look-up Table Type FPGAs," *ICCAD-91*, pp. 560-563, Santa Clara, CA, Nov. 1991.

[26] R. Rudell, and A.L. Sangiovanni-Vincentelli, "Multiple Valued Minimzation for PLA Optimization," *IEEE Trans. on CAD*, vol. 6, no. 5, pp. 727-750, September 1987.

[27] L. Nguyen, M.A. Perkowski, and N.B. Goldstein, "PALMINI - Fast Boolean Minimizer

for Personal Computers," *24th ACM/IEEE DAC*, pp. 615-621, 1987.

[28] R.K. Brayton, R. Rudell, A.L. Sangiovanni-Vincentelli, and A.R. Wang, "MIS: Multi-level Interactive Logic Optimization System," *IEEE Trans. in CAD*, vol. 6, no. t, pp. 1062-1082, November 1989.

[29] P. Bertin, D. Roncin, and J. Vuillemin, "Programmable Active Memories: Performance Measurements," *ACM/SIGDA First International Workshop on Field Programmable Gate Arrays*, pp. 57-59, Berkeley, CA, Feb. 1992.

[30] D. Bostick, G.D. Hachtel, R. Jacoby, M.R. Lightner, P. Moceyunas, C.R. Morrison, and D.Ravenscroft, "The Boulder Optimal Logic Design System," *Proc. ICCAD-87*, pp. 62-65, November 1987.